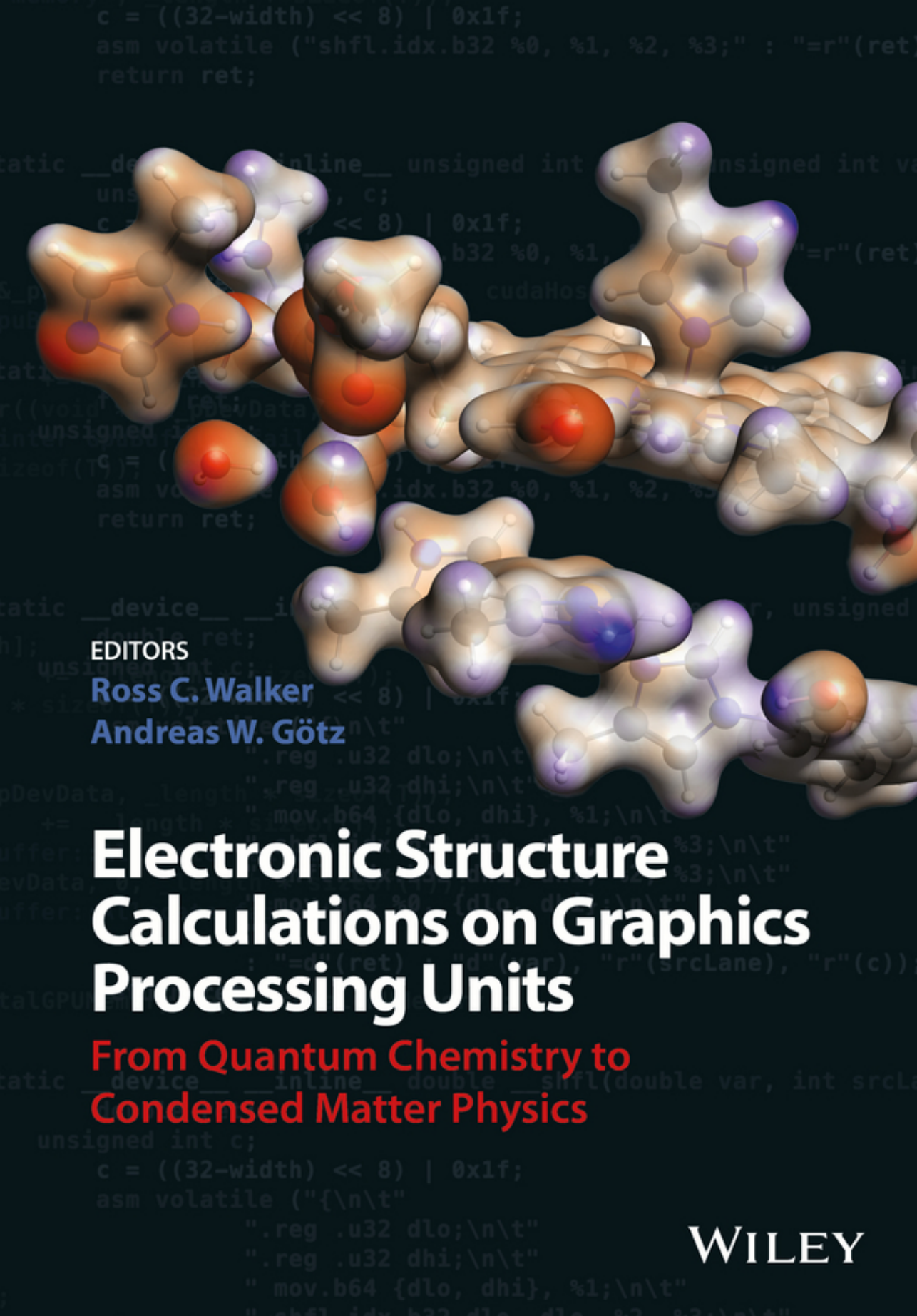


EDITORS

Ross C. Walker

Andreas W. Götz

Electronic Structure Calculations on Graphics Processing Units

From Quantum Chemistry to Condensed Matter Physics

WILEY

Electronic Structure Calculations on Graphics Processing Units

Electronic Structure Calculations on Graphics Processing Units

From Quantum Chemistry to Condensed Matter Physics

Editors

ROSS C. WALKER

San Diego Supercomputer Center and Department of Chemistry and Biochemistry, University of California, San Diego, USA

and

ANDREAS W. GÖTZ

San Diego Supercomputer Center, University of California, San Diego, USA

WILEY

This edition first published 2016

Registered office
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

Library of Congress Cataloging-in-Publication Data applied for.

ISBN: 9781118661789

A catalogue record for this book is available from the British Library.

Cover Image: Courtesy of the Editors

Set in 9/11pt, TimesLTStd by SPi Global, Chennai, India.

1 2016

Contents

List of Contributors

Ani Anciaux-Sedrakian, Mechatronics, Computer Sciences and Applied Mathematics Division, IFP Energies nouvelles, Rueil-Malmaison Cedex, France

Xavier Andrade, Department of Chemistry and Chemical Biology, Harvard University, Cambridge, MA, USA

Edoardo Aprà, William R. Wiley Environmental Molecular Sciences Laboratory, Battelle, Pacific Northwest National Laboratory, Richland, WA, USA

Alán Aspuru-Guzik, Department of Chemistry and Chemical Biology, Harvard University, Cambridge, MA, USA

Jeroen Bédorf, Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

Kiran Bhaskaran-Nair, William R. Wiley Environmental Molecular Sciences Laboratory, Battelle, Pacific Northwest National Laboratory, Richland, WA, USA

Damien Caliste, Université Grenoble Alpes, INAC, Grenoble, France, and CEA, INAC, Grenoble, France

A. Eugene DePrince III, Department of Chemistry and Biochemistry, Florida State University, Tallahassee, FL, USA

Thierry Deutsch, Université Grenoble Alpes, INAC, Grenoble, France, and CEA, INAC, Grenoble, France

Jussi Enkovaara, Department of Applied Physics, Aalto University, Espoo, Finland; CSC – IT Center for Science Ltd, Espoo, Finland

Paul Fleurat-Lessard, Laboratoire de Chimie, Université de Lyon, ENS Lyon, Lyon, France; ICMUB, Université de Bourgogne Franche-Comté, Dijon, France

Luigi Genovese, Université Grenoble Alpes, INAC, Grenoble, France, and CEA, INAC, Grenoble, France

Stefan Goedecker, Institut für Physik, Universität Basel, Basel, Switzerland

Andreas W. Götz, San Diego Supercomputer Center, UCSD, La Jolla, CA, USA

Samuli Hakala, Department of Applied Physics, Aalto University, Espoo, Finland

Jeff R. Hammond, Leadership Computing Facility, Argonne National Laboratory, Argonne, IL, USA

Ville Havu, Department of Applied Physics, Aalto University, Espoo, Finland

Maxwell Hutchinson, Department of Physics, University of Chicago, Chicago, IL, USA

Jürg Hutter, Institute of Physical Chemistry, University of Zürich, Zürich, Switzerland

Adrian Jinich, Department of Chemistry and Chemical Biology, Harvard University, Cambridge, MA, USA

Axel Koslowski, Max-Planck-Institut für Kohlenforschung, Mülheim an der Ruhr, Germany

Karol Kowalski, William R. Wiley Environmental Molecular Sciences Laboratory, Battelle, Pacific Northwest National Laboratory, Richland, WA, USA

Sriram Krishnamoorthy, Computational Sciences and Mathematics Division, Pacific Northwest National Laboratory, Richland, WA, USA

Lin Li, SUNCAT Center for Interface Science and Catalysis, SLAC National Accelerator Laboratory, Menlo Park, CA, USA

Nathan Luehr, Department of Chemistry and the PULSE Institute, Stanford, CA, USA; SLAC National Accelerator Laboratory, Menlo Park, CA, USA

Wenjing Ma, Institute of Software, Chinese Academy of Sciences, Beijing, China

Todd J. Martínez, Department of Chemistry and the PULSE Institute, Stanford, CA, USA; SLAC National Accelerator Laboratory, Menlo Park, CA, USA

Jean-François Méhaut, Université Joseph Fourier – Laboratoire d'Informatique de Grenoble – INRIA, Grenoble, France

Peter Messmer, NVIDIA, Zürich, Switzerland; NVIDIA Co-Design Lab for Hybrid Multicore Computing, Zürich, Switzerland

Perri Needham, San Diego Supercomputer Center, UCSD, La Jolla, CA, USA

Risto M. Nieminen, Department of Applied Physics, Aalto University, Espoo, Finland

Chris O'Grady, SUNCAT Center for Interface Science and Catalysis, SLAC National Accelerator Laboratory, Menlo Park, CA, USA

Roberto Olivares-Amaya, Department of Chemistry, Princeton University, Princeton, NJ, USA

Hans van Schoot, Scientific Computing & Modeling NV, Theoretical Chemistry, Vrije Universiteit, Amsterdam, The Netherlands

Ole Schütt, Department of Materials, ETH Zürich, Zürich, Switzerland

C. David Sherrill, Center for Computational Molecular Science and Technology, Georgia Institute of Technology, Atlanta, GA, USA; School of Chemistry and Biochemistry, Georgia Institute of Technology, Atlanta, GA, USA; School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA, USA

Aaron Sisto, Department of Chemistry and the PULSE Institute, Stanford, CA, USA; SLAC National Accelerator Laboratory, Menlo Park, CA, USA

Dusan Stosic, Department of Computer Science, Federal University of Pernambuco, Recife, Brazil

Sarah Tariq, NVIDIA Corporation, Santa Clara, CA, USA

Walter Thiel, Max-Planck-Institut für Kohlenforschung, Mülheim an der Ruhr, Germany

Antonino Tumeo, Computational Sciences and Mathematics Division, Pacific Northwest National Laboratory, Richland, WA, USA

Joost VandeVondele, Department of Materials, ETH Zürich, Zürich, Switzerland

Brice Videau, Université Joseph Fourier – Laboratoire d'Informatique de Grenoble – INRIA, Grenoble, France

Oreste Villa, Nvidia, Santa Clara, CA, USA

Lucas Visscher, Amsterdam Center for Multiscale Modeling (ACMM), Theoretical Chemistry, VU University Amsterdam, Amsterdam, The Netherlands

Ross C. Walker, San Diego Supercomputer Center, UCSD, La Jolla, CA, USA; Department of Chemistry and Biochemistry, UCSD, CA, USA

Mark A. Watson, Department of Chemistry, Princeton University, Princeton, NJ, USA

Xin Wu, Max-Planck-Institut für Kohlenforschung, Mülheim an der Ruhr, Germany

Jun Yan, SUNCAT Center for Interface Science and Catalysis, SLAC National Accelerator Laboratory, Menlo Park, CA, USA

Preface

The last decade has seen tremendous growth in the use of graphics processing units (GPUs) for numerical simulations, spanning all fields of science. Originally designed for use as coprocessors in graphics applications and visualizations, GPUs have made their way into both mainstream computing platforms and many of the most powerful supercomputers. While substantial effort has gone into the hardware design of GPUs, their potential for scientific computation has only been realized due to the concurrent development of specialized programming approaches and the redesign of the underlying numerical algorithms for massively parallel processors.

Electronic structure calculations are computationally intensive, and the field has a long history of pushing the envelope in high-performance computing. This tradition has continued with the rise of GPUs. Many researchers have invested significant effort in developing software implementations that exploit the computational power of GPUs. This book pays tribute to these developments by collating these efforts into a single reference text.

We have designed this book to provide an introduction to the fast-growing field of electronic structure calculations on massively parallel GPUs. The target audience is graduate students and senior researchers in the fields of theoretical and computational chemistry, condensed matter physics, and materials science, who are looking for an accessible overview of the field, as well as software developers looking for an entry point into GPU and hybrid GPU/CPU programming for electronic structure calculations. To this end, the book provides an overview of GPU computing, a brief introduction to GPU programming, the essential background in electronic structure theory, and the latest examples of code developments and applications for the most widely used electronic structure methods.

We have tried to include all widely used electronic structure methods for which GPU implementations have been developed. The text covers all commonly used basis sets including localized Gaussian- and Slater-type basis functions, plane waves, wavelets, and real-space grid-based approaches. Several chapters expose details on strategies for the calculation of two-electron integrals, exchange-correlation quadrature, Fock matrix formation, solution of the self-consistent field equations, calculation of nuclear gradients to obtain forces, and methods to treat excited states within density functional theory. Other chapters focus on semiempirical methods and correlated wave function methods including density-fitted second-order Møller–Plesset perturbation theory and both iterative and perturbative single- and multireference coupled-cluster methods.

We have enjoyed the steep learning curve that has accompanied the editing of this book, and we trust that you, the reader, will find it an engaging and useful reference.

Ross C. Walker and Andreas W. Götz
August 2015
La Jolla, USA

Acknowledgments

We would like to thank everybody who has made a contribution to this book, either directly or indirectly. This includes our friends and families for their continuous support. Our special thanks go to the dedicated people at Wiley who guided us in our role as editors and worked hard for this book to see the light of day. In particular, we thank our primary contacts Sarah Keegan, Sarah Higginbotham, and Rebecca Ralf. We are also grateful to Dorathy Steve and her team at SPi for copy-editing. This book would never have been possible without the excellent contributions of the many individual authors who contributed to its content. We are grateful that they accepted to write chapters for this book and we would like to thank all for their patience during the editing period.

Glossary

The following provides a details glossary of GPU and GPU programming related terms. It is biased towards NVIDIA GPUs and the CUDA programming model. However, AMD GPUs use similar concepts and hardware implementations. For instance, the equivalent of a CUDA warp on NVIDIA hardware is called a wavefront on AMD hardware. This is not meant to be an exhaustive list of terms but rather is designed to provide the reader with a brief description of the various GPU-related technical terms that appear in this book.

Bandwidth The inverse of the time that is required to transfer one byte of data. Usually measured in GB/s.

Block A set of threads that can share data and communicate during execution on the GPU. Data across blocks cannot be synchronized. A block executes on a single SM. To optimize the performance, the block-size needs to be adjusted to the problem and the hardware (e.g., available shared memory). The number of threads in a block is limited to 1024 on the Kepler architecture. Thread blocks are created and executed in units of a warp; thus, the number of threads should be a multiple of the warp size (32 currently). A thread block has its block ID within its grid.

Cache Fast memory that is used to reduce latency for global memory access. On NVIDIA GPUs with Kepler architecture, the SMs share a cache of 1.5 MB size. This can be considered L2 cache since each SM has an L1 cache that is called shared memory.

Constant memory Fast read-only memory that can be written by the host and accessed by all SMs. Only a limited amount of constant memory is available.

Device The GPU including its processor and memory. The device cannot operate on data that is located on the host.

Global memory The memory that is available on the GPU. Comparable to main memory on the host. Data access to global memory is cached but is slow compared to other memory classes on the GPU due to higher latency. Compared to host memory, the global device memory supports high data bandwidth. On current NVIDIA hardware with Kepler architecture, the data path is 512 bits wide; thus, 16 consecutive 32-bit words can be fetched in a single cycle. As a consequence, there is considerable bandwidth degradation for strided memory access. For instance, a stride-two access will fetch 512 bits but use only half of them. There is less device memory than host memory, at present up to 12 GB on NVIDIA Tesla K40. Global memory can be accessed by the host for data transfers between the host and the device. Global memory is persistent between kernel launches.

Grid A set of blocks that maps to the streaming multiprocessors on the GPU and execute a kernel. The order of execution of the blocks on a GPU is not deterministic. In the Kepler architecture, 16 blocks can be active at the same time in a single multiprocessor.

Host The CPU and its main memory. The host cannot operate on data that is located on the device. A program running on the host can transfer data to/from the device and launch kernels on the device.

Kernel A function that executes in parallel on the GPU. NVIDIA GPUs are programmed as a sequence of kernels that are launched by the host program. By default, a kernel completes execution before the start of the next kernel with an implicit synchronization barrier. Usually, kernels execute a sufficiently large number of thread blocks to occupy all SMs of a GPU. However, the Kepler architecture supports simultaneous execution of multiple independent kernels at the same time. Kernel launches execute multiple threads that are arranged in a grid of blocks.

Latency The time it takes from the issue of a memory operation to the arrival of the first bit. Usually measured in µs.

Latency hiding Techniques to deal with high latency of data transfer between the host and device or access to global device memory. For example, a device memory operation issued by threads in a warp will take very long due to latency on the order of hundreds of clock cycles. CPU architectures make use of a cache memory hierarchy to reduce latency; however, this is not effective on GPUs, which are designed for throughput computing. GPUs instead deal with this latency by using a high degree of multithreading. At a given point in time, up to 64 warps can be active on each multiprocessor in the Kepler architecture. While one warp is waiting for a memory operation to complete, the control unit switches to another warp. Thus, all cores can continue computing if the parallelism on each SM is sufficiently large.

Local memory Slow memory that is located off-chip and has the same latency as global memory. It is used to hold automatic variables for cases in which there is not sufficient register memory available. Variables stored in local memory are private to each thread.

Register memory Very fast on-chip memory faster than shared memory. Used to store local variables that are private to each thread. On the Kepler architecture, a thread can access up to 255 32-bit registers; however, there is only a total of 65,536 32-bit registers on an SM. Ideally, all local variables used by a thread reside in registers on chip. The limited number of registers thus limits the number of concurrent threads. Memory intensive kernels can move data to local memory (this is termed register spillage) with adverse effect on performance due to high latency of local memory.

Register spillage Term used if memory intensive kernels require more storage than is available in registers thus moving data to local memory, which usually has detrimental effect on performance.

Shared memory Shared memory on NVIDIA GPUs is a fast on-chip memory, essentially a programmable L1 cache attached to each SM. It has low latency and high bandwidth with speed that is close to that of registers. On NVIDIA Kepler architecture, the shared memory is 64 KB for each SM and can be configured as 25%, 50% or 75% software managed cache with the remainder as hardware data cache. Data stored in shared memory can be accessed by all threads in the same thread block and persists only for the lifetime of the execution of a block. Since it is a limited resource per SM, its use limits the number of blocks that can be concurrently executed. The host cannot access shared memory.

SIMD Single instruction multiple data. A SIMD processing unit executes single instructions on multiple data. Branching is not possible.

SIMT Single instruction multiple threads. Model for data parallel computing on GPUs. Each core in an SM can execute a sequential thread but all cores in a group called warp execute the same instruction at the same time similar to classical SIMD processors. Branching is possible, but for conditional operations some of the cores in a warp are disabled resulting in no-ops.

Stream Sequence of commands that execute in order. Multiple Streams can be used to execute kernels simultaneously or to overlap kernel execution with memory copies between host and device.

Streaming Multiprocessor (SM) Set of processing cores (ALUs) on a GPU that have access to a common shared memory space. SMs on the NVIDIA GK110 chip, for example, contain 192 single-precision cores and 64 double-precision cores. Groups of 16 cores execute operations of a group of threads in a warp in lockstep. A maximum of 64 warps (2048 threads) can be active at the same time on an SM (see also Latency Hiding). These warps can belong to a maximum of 16 different thread blocks. SMs operate at approximately 1 GHz clock speed, thus at a lower speed than typical CPUs.

Texture memory Read-only memory that can be written by the host. Texture memory resides in device global memory but is cache-optimized for certain read operations, for example, two-dimensional arrays.

Thread (software) In the context of GPU programming, a thread is a sequence of instructions to be executed by a GPU processing element. On a GPU, threads are grouped into blocks and threads within a block are executed in lockstep in sizes of a warp. Each thread thus executes an instance of a kernel. Each thread has thread block and grid ID within its threads block and grid, a program counter, registers, and per-thread private memory available.

Warp Lock-step unit on a GPU. Threads within a warp execute in lock-step, that is in SIMD fashion. However, branching is allowed. Each warp should access a single cache line. A warp always consists of a subset of threads of a block. A warp consists of 32 threads (this number has remained constant so far but is subject to change). On a Kepler SM, a warp takes two cycles to execute one integer or single-precision floating point instruction on each group of 16 cores. At most 4 of the 12 groups of cores in a Kepler SM can execute double-precision instructions concurrently. At most 2 of the 12 groups of cores can concurrently execute intrinsic and transcendental functions.

Abbreviations - Scientific

ACFDT adiabatic connection fluctuation-dissipation theorem

AETRS approximate enforced time-reversal symmetry

AM1 Austin Model 1

AMBER Assisted Model Building with Energy Refinement

AO atomic orbital

BOINC Berkeley Open Infrastructure for Network Computing

BZ Brillouin zone

CC Coupled cluster

CCD Coupled cluster doubles

CCSD Coupled cluster singles doubles

CCSD(T) CCSD with perturbative triples

CD Cholesky Decomposition

CIS Configuration interaction singles

CMS Complete model space

DIIS Direct inversion in the iterative subspace

DF Density Fitting

DFT Density functional theory

ERI Electron-electron repulsion integral

GF Generating functional (moment expansion)

GGA Generalized gradient approximation

GTO Gaussian type orbital

HS Hilbert space

KS Kohn-Sham

LCAO Linear combination of atomic orbitals

LDA Local Density Approximation

LMP2 Local second order Møller-Plesset perturbation theory

MBPT Many-body perturbation theory

MRMBPT Multi-reference many-body perturbation theory

MD Molecular Dynamics

MNDO Modified neglect of diatomic overlap

MO molecular orbital

MP2 second order Møller-Plesset perturbation theory

MRCC Multi-reference Coupled Cluster

MRCI Multi-reference configuration interaction

MRMBPT Multi-reference many-body perturbation theory

NCPP Norm-conserving pseudopotential

NDDO Neglect of diatomic differential overlap

OMx Orthogonalization method x (x=1,2,3)

PAO Projected Atomic Orbital

PM3 Parametric method 3

RLP Reference-level Parallelism

STO Slater type orbital

TDDFT Time-dependent density functional theory

TDHF Time-dependent Hartree-Fock theory

TDKS Time-dependent Kohn-Sham

TCE Tensor Contractor Engine

HF Hartree-Fock

PAW Projector-augmented wave

PWDFT Plane-wave DFT

PQ Pair quantity

RMM-DIIS residual minimization-direct inversion in the iterative subspace

RPA Random phase approximation

SCF Self-consistet field

SRCC Single Reference Coupled Cluster

USPP Ultra-soft pseudopotential

XC exchange-correlation

ZDO zero differential overlap

Abbreviations - Technical

APPML Accelerated Parallel Processing Math Libraries

ARMCI Aggregate Remote Memory Copy Interface

AVX Advanced Vector Extensions

BLAS Basic linear algebra subroutines

CPU Central processing unit

CUBLAS CUDA Basic linear algebra subroutines

DAG Directed acyclic graph

DBCSR Distributed blocked compressed sparse row (software library)

DP Double precision

DRAM Dynamic random access memory

DSL Domain Specific Language

FFTW Fastest Fourier Transform in the West (software library)

FLOPS Floating point operations per second

FPGA Field programmable gate array

GEMM General Matrix Multiply

DGEMM General Matrix Multiply in Double Precision

MGEMM General Matrix Multiply in Mixed Precision

SGEMM General Matrix Multiply in Single Precision

GA Global arrays (programming standard)

GPU Graphics processing unit

GSL GNU Scientific Library

HPC High performance computing

ISA Instruction Set Architecture

LAPACK Linear algebra package

MAGMA Matrix algebra on GPU and multicore architectures (software library)

MPI Message passing interface

MPS Multi-process server (Nvidia)

NUMA Non-Uniform Memory Architecture:w

OpenACC Open accelerators (programming standard)

OpenCL Open computing language (programming standard)

OpenMP Open multi-processing

OS Operating System

PCIe PCI express, Peripheral component interconnect express

PG Processor Group

RAM Random access memory

SFU Special Function Unit

SIMD Single instruction multiple data

SIMT Single instruction multiple threads

SM Streaming multiprocessor

SMP Symmetric multiprocessing

SP Single precision

1

Why Graphics Processing Units

Perri Needham[1]*, Andreas W. Götz*[2] *and Ross C. Walker*[1,2]

[1]*San Diego Supercomputer Center, UCSD, La Jolla, CA, USA*
[2]*Department of Chemistry and Biochemistry, UCSD, La Jolla, CA, USA*

1.1 A Historical Perspective of Parallel Computing

The first *general-purpose electronic* computers capable of storing instructions came into existence in 1950. That is not to say, however, that the use of computers to solve electronic structure problems had not already been considered, or realized. From as early as 1930 scientists used a less advanced form of computation to solve their quantum mechanical problems, albeit a group of assistants simultaneously working on mechanical calculators but an early parallel computing machine nonetheless [1]. It was clear from the beginning that solutions to electronic structure problems could not be carried forward to many-electron systems without the use of some computational device to lessen the mathematical burden. Today's computational scientists rely heavily on the use of parallel electronic computers.

Parallel electronic computers can be broadly classified as having either multiple processing elements in the same machine (shared memory) or multiple machines coupled together to form a cluster/grid of processing elements (distributed memory). These arrangements make it possible to perform calculations concurrently across multiple processing elements, enabling large problems to be broken down into smaller parts that can be solved simultaneously (in parallel).

The first *electronic* computers were primarily designed for and funded by military projects to assist in World War II and the start of the Cold War [2]. The first working *programmable digital computer*, Konrad Zuse's Z3 [3], was an electromechanical device that became operational in 1941 and was used by the German aeronautical research organization. Colossus, developed by the British for cryptanalysis during World War II, was the world's first *programmable electronic* digital computer and was responsible for the decryption of valuable German military intelligence from 1944 onwards. Colossus was a *purpose-built* machine to determine the encryption settings for the German Lorenz cipher and

Electronic Structure Calculations on Graphics Processing Units:
From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

Figure 1.1 *Photograph taken in 1957 at NASA featuring an IBM 704 computer, the first commercially available general-purpose computer with floating-point arithmetic hardware [4]*

read encrypted messages and instructions from paper tape. It was not until 1955, however, that the first *general-purpose* machine to execute floating-point arithmetic operations became commercially available, the IBM 704 (see Figure 1.1).

A common measure of compute performance is floating point operations per second (FLOPS). The IBM 704 was capable of a mere 12,000 floating-point additions per second and required 1500–2000 ft^2 of floor space. Compare this to modern smartphones, which are capable of around 1.5 *GIGA* FLOPS [5] thanks to the invention in 1958 and a subsequent six decades of refinement of the integrated circuit. To put this in perspective, if the floor footprint of an IBM 704 was instead covered with modern-day smartphones laid side by side, the computational capacity of the floor space would grow from 12,000 to around 20,000,000,000,000 FLOPS. This is the equivalent of every person on the planet carrying out roughly 2800 floating point additions per second. Statistics like these make it exceptionally clear just how far computer technology has advanced, and, while mobile internet and games might seem like the apex of the technology's capabilities, it has also opened doorways to computationally explore scientific questions in ways previously believed impossible.

Computers today find their use in many different areas of science and industry, from weather forecasting and film making to genetic research, drug discovery, and nuclear weapon design. Without computers many scientific exploits would not be possible.

While the performance of individual computers continued to advance the thirst for computational power for scientific simulation was such that by the late 1950s discussions had turned to utilizing multiple processors, working in harmony, to address more complex scientific problems. The 1960s saw the birth of parallel computing with the invention of multiprocessor systems. The first recorded example of a commercially available multiprocessor (parallel) computer was Burroughs Corporation's D825, released in 1962, which had four processors that accessed up to 16 memory modules via a cross switch (see Figure 1.2).

Figure 1.2 *Photograph of Burroughs Corporation's D825 parallel computer [6]*

This was followed in the 1970s by the concept of single-instruction multiple-data (SIMD) processor architectures, forming the basis of vector parallel computing. SIMD is an important concept in graphics processing unit (GPU) computing and is discussed in the next chapter.

Parallel computing opened the door to tackling complex scientific problems including modeling electrons in molecular systems through quantum mechanical means (the subject of this book). To give an example, optimizing the geometry of any but the smallest molecular systems using sophisticated electronic structure methods can take days (if not weeks) on a single processor element (compute core). Parallelizing the calculation over multiple compute cores can significantly cut down the required computing time and thus enables a researcher to study complex molecular systems in more practical time frames, achieving insights otherwise thought inaccessible. The use of parallel *electronic* computers in quantum chemistry was pioneered in the early 1980s by the Italian chemist Enrico Clementi and co-workers [7]. The parallel computer consisted of 10 compute nodes, loosely coupled into an array, which was used to calculate the Hartree–Fock (HF) self-consistent field (SCF) energy of a small fragment of DNA represented by 315 basis functions. At the time this was a considerable achievement. However, this was just the start, and by the late 1980s all sorts of parallel programs had been developed for quantum chemistry methods. These included HF methods to calculate the energy and nuclear gradients of a molecular system [8–11], the transformation of two-electron integrals [8, 9, 12], the second-order Møller–Plesset perturbation theory [9, 13], and the configuration interaction method [8]. The development of parallel computing in quantum chemistry was dictated by developments in available technologies. In particular, the advent of application programming interfaces (APIs) such as the message-passing interface (MPI) library [14] made parallel computing much more accessible to quantum chemists, along with developments in hardware technology driving down the cost of parallel computing machines [10].

While finding widespread use in scientific computing until recently, parallel computing was reserved for those with access to high-performance computing (HPC) resources. However, for reasons discussed in the following, all modern computer architectures exploit parallel technology, and effective parallel programming is vital to be able to utilize the computational power of modern devices. Parallel processing is now standard across all devices fitted with modern-day processor architectures. In his 1965 paper [15], Gordon E. Moore first observed that the number of transistors (in principle, directly related to performance) on integrated circuits was doubling every 2 years (see Figure 1.3).

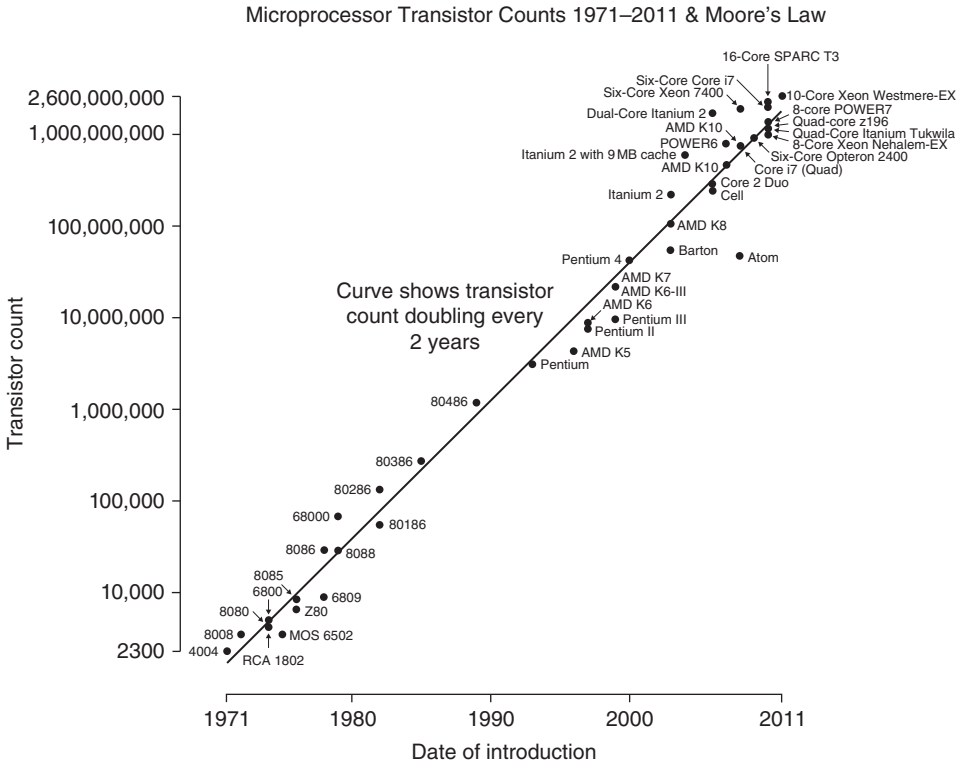


Figure 1.3 *Microprocessor transistor counts 1971–2011. Until recently, the number of transistors on integrated circuits has been following Moore's Law [16], doubling approximately every 2 years*

Since this observation was announced, the semiconductor industry has preserved this trend by ensuring that chip performance doubles every 18 months through improved transistor efficiency and/or quantity. In order to meet these performance goals the semiconductor industry has now improved chip design close to the limits of what is physically possible. The laws of physics dictate the minimum size of a transistor, the rate of heat dissipation, and the speed of light.

> *"The size of transistors is approaching the size of atoms, which is a fundamental barrier" [17].*

At the same time, the clock frequencies cannot be easily increased since both clock frequency and transistor density increase the power density, as illustrated by Figure 1.4. Processors are already operating at a power density that exceeds that of a hot plate and are approaching that of the core of a nuclear reactor.

In order to continue scaling with Moore's Law, but keep power densities manageable, chip manufacturers have taken to increasing the number of cores per processor as opposed to transistors per core. Most processors produced today comprise multiple cores and so are parallel processing machines by definition. In terms of processor performance, this is a tremendous boon to science and industry; however, the increasing number of cores brings with them increased complexity to the programmer in order to fully utilize the available compute power. It is becoming more and more difficult for applications to achieve good scaling with increasing core counts, and hence they fail to exploit this technology. Part of this problem stems from the fact that CPU manufactures have essentially

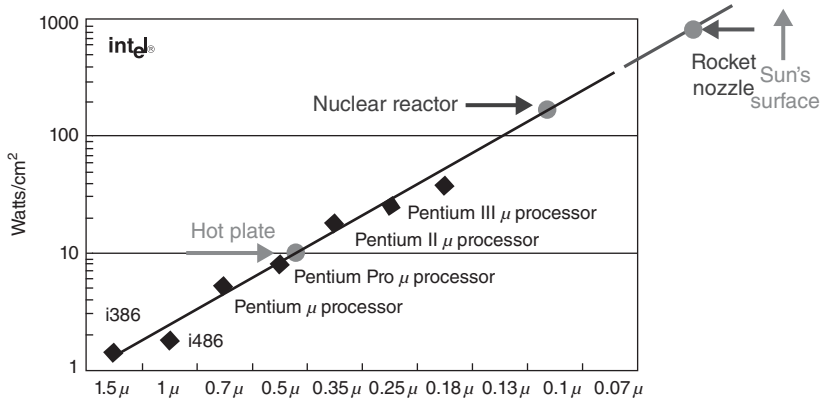


Figure 1.4 *Illustration of the ever-increasing power density within silicon chips, with decreasing gate length. Courtesy Intel Corporation [18]*

taken serial (scalar) architectures and their associated instructions sets and attempted to extend them to a multicore regime. While this is well tolerated for single-digit core counts, once the number of cores hits double digits the complexity of having to keep track of how the data are distributed across multiple independent cores, which requires communication between local cache memories, quickly leads to performance bottlenecks. This complexity has, to date, limited the number of cores and, ultimately, the performance available in traditional CPUs. In the next section we discuss an alternative to multicore programming that utilizes the massively parallel (SIMD) architecture of GPUs.

1.2 The Rise of the GPU

The concept of using a specialized coprocessor to supplement the function of the central processing unit originated in the 1970s with the development of math coprocessors. These would handle floating-point arithmetic, freeing the CPU to perform other tasks. Math coprocessors were common throughout the 1980s and early 1990s, but they were ultimately integrated into the CPU itself. The theme of using a coprocessor to accelerate specific functions continued however, with video accelerators being the most common example. These GPUs would free the CPU from much of the complex geometric math required to render three-dimensional (3D) images, and their development was catalyzed by the computer gaming industry's desire for increasingly realistic graphics.

Beyond the math coprocessor of the 1980s, coprocessors have been routinely used in scientific computation as a means of accelerating mathematically intensive regions of code. More recent examples include ClearSpeed's accelerator boards, which combined hundreds of floating point units on a single PCI card, and field-programmable gate arrays (FPGAs), which can be reconfigured to accelerate a specific computational problem. These accelerator technologies, while exploited by some, have not found widespread use in the general scientific computing arena for a number of reasons, the main one being their cost. However, one coprocessor technology that has succeeded in being adopted by the wider scientific computing community is the GPU. The reasons for this are many, but probably the overriding reason for their widespread adoption is that they are ubiquitous and cheap. GPUs have been an integral part of personal computers for decades. Ever since the introduction of the Voodoo graphics chip by 3DFX in 1996, the entertainment industry has been the major driving force for the development of GPUs in order to meet the demands for increasingly realistic computer games. As a result of the strong demand from the consumer electronics industry, there has been significant industrial investment in the stable, long-term development of GPU

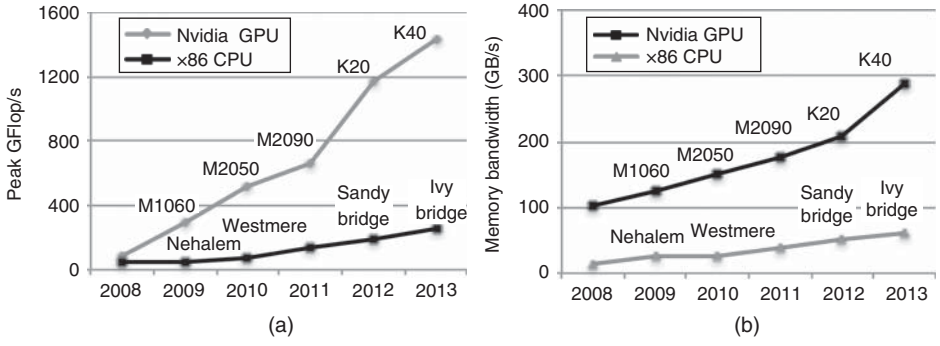


Figure 1.5 *Peak floating-point operations per second (a) and memory bandwidth (b) for Intel CPUs and Nvidia GPUs. Reproduced from [19]*

technology. Nowadays, GPUs have become cheap and ubiquitous and, because their processing power has substantially increased over the years, they have the potential, when utilized efficiently, to significantly outperform CPUs (see Figure 1.5).

GPUs are thus very attractive hardware targets for the acceleration of many scientific applications including the subject of this book, namely electronic structure theory. GPUs with significant computing power can be considered standard equipment in scientific workstations, which means that they either are already available in research labs or can be purchased easily with new equipment. This makes them readily available to researchers for computational experimentation. In addition, many compute clusters have been equipped with large numbers of high-end GPUs for the sole purpose of accelerating scientific applications. At the time of writing, the most powerful supercomputer that is openly accessible to the scientific community draws the majority of its peak computational performance from GPUs.

The nature of GPU hardware, however, originally made their use in general-purpose computing challenging because it required extensive three-dimensional (3D) graphics programming experience. However, as discussed in the following, the development of APIs for general-purpose scientific computing has reduced this complexity such that an extensive range of scientific problems is making use of GPU acceleration in an economically efficient manner. GPU-accelerated computing is now a broad but still advancing area of research and development that is capable of providing scalable supercomputing power at a fraction of the cost and, potentially, a fraction of the software development effort (depending on the application).

GPUs were originally designed for computer graphics, which require repeated evaluation of pixel values and little else. For example, branching is not required in rendering 3D images and therefore in place of sophisticated cache hierarchies, branch prediction hardware, pipeline controllers, and other complex features, GPUs instead have hundreds to thousands of simplistic cores. GPUs are fundamentally parallel, with each core computing as an individual thread, making them ideal for data decompositions with few dependencies between threads. For fine-grained parallel workloads, GPUs can thus provide a substantial amount of computing power. As can be seen from Figure 1.5, they provide, largely by the difference in memory bandwidth, a means to improve the performance of a suitable code far beyond the limits of a CPU.

The scientific community has leveraged this processing power in the form of general-purpose (GP) computation on GPUs, sometimes referred to as GPGPU. GPGPUs were specifically designed to target HPC applications incorporating additional features into GPUs, such as caching technology, and support for double-precision arithmetic. The success of early GPGPU efforts was such that these features are now considered standard in almost all GPUs, and indeed many modern computer games

actually make use of the GPGPU functionality to compute game physics. The traditional approach to computing on a GPU is as an accelerator, or coprocessor, working in conjunction with the CPU. The GPU executes specific functions within the code (known as kernels), which are deemed suitable for parallelization on a GPU. A GPU operates a SIMD instruction set, whereby an instruction is issued to many cores in a single instance. The lightweight nature of a GPU core makes it ideally suited to parallelize compute-intensive code with few data dependencies because, once the instruction is issued, cores cannot diverge along different code paths. If branching occurs on cores issued with the same instruction, some cores will stall until the execution paths converge. This artifact of GPU hardware is what sets them apart from CPU architectures. It enables calculation of multiple data elements in one instance, provided the code design allows it. However, that is not to say that CPUs do not make use of SIMD, but rather GPUs implement it on a much larger scale. For example, the latest GPU in Nvidia's Tesla range, Tesla K40, boasts 2880 GPU cores and has the capacity to launch multiple threads per core, which can be quickly switched in and out with minimal overhead, whereas the latest CPU in the Intel® Core™ product family, Intel® Core™ i7-4930K processor, comprises six cores capable of spawning two threads per core. The CPU makes use of SIMD to carry out multiple operations per thread; in this case, each thread is a 256-bit SIMD instruction in the form of AVX. In principle, the CPU can therefore execute 96 (256-bit/32-bit $\times$ 12 threads) single-precision floating-point operations per clock cycle, although in reality it is effectively half of this since the two threads per core are time-sliced. A GPU therefore can potentially process hundreds, if not thousands, of data elements in a single instance as opposed to a mere 48 on a CPU. However, it cannot be emphasized enough how important the GPU code design is in order to take advantage of this potential performance, often requiring a rethinking of the underlying mathematical algorithms. The purpose of this book is to highlight such rethinking within the framework of electronic structure theory.

1.3 Parallel Computing on Central Processing Units

Before entering into the world of GPU programming it is important to cement some key concepts that lie at the heart of parallel programming. Parallel programming requires substantially more consideration than serial programming. When parallelizing computer code it cannot be presumed that all parallel processing elements have access to all the data in memory or indeed that memory updates by one element will automatically propagate to the others. The locality of data to a processing element is dependent on the setup of the computer system being used and the parallel programming memory model, which is the first concept to be discussed in this section. Following this are the language options available to the programmer that are best suited to a specific programming model and also a discussion of the different types of parallelism to consider when breaking down a problem into smaller parallel problems. It is also beneficial to be aware of the factors that may affect the performance of a parallel program and how to measure the performance and efficiency of a parallel program, which will be the final topics discussed. For a more detailed exposition than presented in the following, we refer the reader to the excellent text book on HPC by Hager and Wellein, which also covers aspects of computer architecture and serial code optimizations in addition to parallelization [20].

1.3.1 Parallel Programming Memory Models

In parallel computing, there are two main memory models: the shared memory model and the distributed memory model. Each model brings with it different implications at both the hardware and software level. In hardware, the *shared memory model* is exactly that – multiple processors share a single address space in memory. The *distributed memory model* refers to segregated memory, where

each processor has its own private address space, be it a physically private memory space or a portion of a larger memory.

In shared memory all processors have access to all addresses in memory making communication between processors potentially very fast. However, problems arise when more than one processor requires access to the same memory address, resulting in potential race conditions, a situation – discussed in more detail later in this chapter – where the result of a computation is a function of which of the threads computes the answer first, and communication overhead in keeping the data stored in different memory caches coherent. In distributed memory all processors have access only to their own memory address space and so communication between processors becomes the bottleneck. If a process requires data that is not stored in its private memory space the process first has to find which processor's private memory the data is stored in before it can be transferred via an interconnect. Consequently, *shared* memory systems are deemed more suited to fewer processor counts, as performance falls off with increasing numbers of processors as a result of explosion of cache coherency overhead. Neither memory model is ideal. In practice, most modern-day parallel computers use a mixture of the two types of memory model. Many scientific computing clusters utilize a hybrid memory model, with shared memory nodes coupled together with an interconnect providing a distributed memory model.

1.3.2 Parallel Programming Languages

An Application Programming Interface (API) fundamentally is a means of communicating with a computer. Communication with a computer is layered and can be easily imagined as a metaphorical onion. At the very core of the onion is *binary*. Zeros and 1's represent the switch states of transistors on a piece of silicon. Binary allows these switch states to represent *real* numbers. The next layer of the onion is *machine code*. A small number of very basic operations instruct the CPU what to do with the *real* numbers. The layer after that is *assembly*, which is similar to *machine code* except with slightly more human-friendly operations enabling manipulation of the *real* numbers with more ease. And so it goes on. The larger the onion, that is, the more the layers of communication, the easier it is for a programmer to express his/her computational problem because with each added layer the language becomes closer and closer to human communication. These language layers are known as APIs and can be deemed *high-level* (outer layers of a large onion) or *lower level* (layers close to the core). The two most widely used APIs for expressing parallel CPU code are MPI (Message Passing Interface) [14] and OpenMP (Open Multi-Processing) [21]. MPI is a library called upon from existing CPU code. The library is language-independent, which means it can be called from code written in any compatible programming language, that is, C, C++, or Fortran. MPI enables point-to-point and collective communication between processors in a system and is primarily designed for use with a distributed memory system, but it can also be used within a shared memory setting. OpenMP is primarily a set of compiler directives that can be called from within C/C++ and Fortran code. OpenMP is designed primarily for shared memory programming, although OpenMP 4.0 supports a "target directive" for running across devices that do not share memory spaces and exhibits a multithreaded paradigm to distribute work across processors. OpenMP is considered a higher level parallel programming API than MPI, as the runtime environment and the compiler do most of the work. The programmer can do as little as specify which sections of code require parallelizing through the use of a directive and let the API do the rest of the work. MPI requires more thought, as the programmer is responsible for keeping track of data in memory and explicitly passing data between processors. However, one benefit from this added degree of difficulty when using MPI is that it forces the programmer to think about data locality and so can lead to more efficient, higher performing code.

Other APIs used in parallel programming are high-performance Fortran (HPF), POSIX threads, and Unified Parallel C (UPC), to name a few; however, they are beyond the scope of this textbook.

1.3.3 Types of Parallelism

One key consideration when parallelizing code is how to distribute the work across processors. The two main types of decomposition are task parallelism and data parallelism. *Task parallelism* involves deconstructing code into separate, independent tasks/calculations able to be computed on different processors concurrently. Each processor is responsible for executing its own set of instructions using the same or different data as other processes. Figure 1.6 shows a simple example of a four-task program. Task B is dependent on Task A, therefore they are executed sequentially. Task C is independent of both A and B, therefore they can be executed concurrently; Task D is dependent on the completion of Tasks B and C, therefore it is executed on their completion.

Data parallelism involves decomposing a dataset across processors, and each processor executing the same instructions but on different data. Data parallelism is synonymous with SIMD architecture. There are many different strategies for data parallelism, for example, *static* where the data are decomposed into chunks and distributed across processors at compile time and each processor carries out all operations on the same chunk of memory initially allocated, and *dynamic* parallelism where a load balancer hands out work on a per-request basis. A more detailed description of data decomposition strategies is beyond the scope of this chapter and so will not be discussed further.

Similar to the parallel memory models, a combination of both types of parallelism can often be the most fruitful where performance is concerned. It can be beneficial to employ both types of parallelism, for example, decomposing code into tasks across multiple compute nodes and further decomposing the data for each task across multiple processors within a compute node. By doing this, multiple levels of parallelism can be exploited, making better use of the available computer architecture.

The ease and extent by which an algorithm can be broken up into separate tasks/calculations is known as the *granularity*. *Fine-grained* parallelism describes algorithms that can be subdivided into many small calculations. These types of algorithm are said to be more easily parallelized but can require more synchronization/communication between parallel processes. On the contrary, *coarse-grained* parallelism is the subdivision of an algorithm into fewer larger tasks/calculations. These types of algorithm do not benefit from as much parallelism, that is, scale to small processor

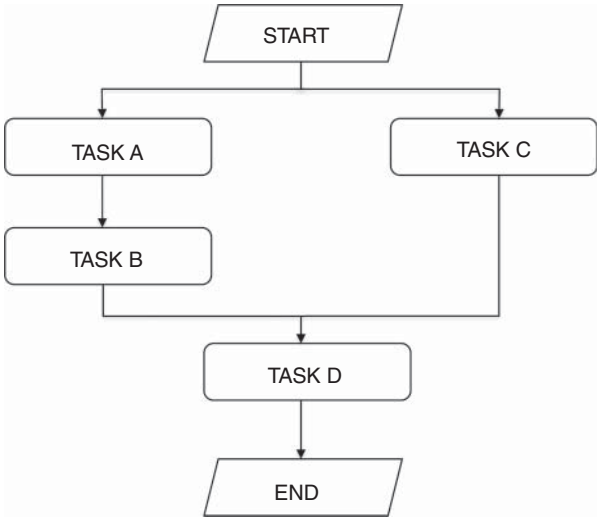


Figure 1.6 *Directed acyclic graph demonstrating the decomposition of four tasks across two processors*

counts, but can benefit from fewer overheads as a result of reduced communication. Algorithms that can be subdivided into many small tasks (similar to fine-grained parallelism) yet do not carry large communication overheads (similar to coarse-grained parallelism) are said to be *embarrassingly* parallel (although the term naturally parallel might be a better description) and are the easiest applications to parallelize.

1.3.4 Parallel Performance Considerations

1.3.4.1 Speedup

Before parallelizing code, it is important to realistically predict the potential for performance improvement. A common measure is speedup (S), which is given by

$$S(n) = \frac{T(1)}{T(n)}, \tag{1.1}$$

where $T(1)$ is the execution time of the original algorithm and $T(n)$ is the execution time of the new algorithm on n processors/cores/threads.

1.3.4.2 Amdahl's Law and Gustafson's Law

Amdahl's and Gustafson's laws provide a guideline for speedup after parallelizing code. Effectively a simplification of a concept, familiar to all chemists, known as a rate-determining step, they show that the serial portion of a parallel program is always the limiting factor in speedup. There are slight subtleties between the two laws; Amdahl presumes a fixed problem size, whereas Gustafson presumes increasing problem size with increasing numbers of processors. Amdahl's Law, named after the computer architect Amdahl [22], was introduced in 1967 and is defined by

$$T(n) = T(1)\left(\beta_A + \frac{1}{n}\left(1 - \beta_A\right)\right), \tag{1.2}$$

$$S(n) = \frac{T(1)}{T(1)\left(\beta_A + \frac{1}{n}\left(1 - \beta_A\right)\right)} = \frac{1}{\beta_A + \frac{1}{n}(1 - \beta_A)}, \tag{1.3}$$

where β_A is the fraction of the algorithm that is non-parallelizable, that is, the fraction of the total run time $T(1)$ that the serial program spends in the non-parallelizable part. As can be seen from Amdahl's equation, as the number n of processors increases, the speedup becomes limited by the serial code fraction β_A. However, Amdahl's Law is rather simplistic, as it makes assumptions such as the number of processes used throughout the parallel portions of the code is a constant; the parallel portion achieves linear speedup (i.e., the speedup is equal to the number of processes); and the parallel portion scales perfectly, to name a few. However, the most important assumption made was that the serial and parallel workloads remain constant when increasing the number of processors. This is true for many scientific problems, particularly those involving integrals over time.

However, some applications are problem-size-bound rather than time-bound, such as weather modeling and graphics rendering, and given more processors, the size of the problem is usually increased to fill the void. Gustafson attempted to remedy this shortcoming in 1988 by defining parallel speedup by a new equation [23], sometimes referred to as "scaled speedup":

$$T(1) = (n - \beta_G(n)(n-1))T(n), \tag{1.4}$$

$$S(n) = n - \beta_G(n)(n-1), \tag{1.5}$$

where β_G is again the non-parallelizable fraction, but now defined as the fraction of the total time $T(n)$ that the parallel program spends in serial sections if run on n processors. The two different definitions

of speedup in terms of processor count n and serial code fraction β can lead to confusion. However, one has to keep in mind that for a given problem size, β_G depends on the number of processors, as indicated in Eqs. (1.4) and (1.5). Equations (1.3) and (1.5) are actually equivalent with the following relation between the different definitions of the serial code fraction, as can be derived from Eqs. (1.2) and (1.4):

$$\beta_A = \frac{1}{1 + \frac{(1-\beta_G)n}{\beta_G}}. \tag{1.6}$$

In the application of Eq. (1.5), one typically assumes that β_G remains constant with increasing processor number. In other words, Gustafson's Law states that exploiting the parallel power of a large multiprocessor system requires a large parallel problem. Gustafson's Law thus allows for expansion of the parallel portion of an algorithm but still makes all the same assumptions as Amdahl's Law. Either way, the serial code will ultimately become the bottleneck as the core count increases. It is thus essential for effective parallel programming that all serial aspects of the compute portion of the code are removed. If a section of the algorithm is serial, even though representing only a tiny fraction of the total compute time, it is necessary to rethink the approach to remove this serial bottleneck if one wishes to scale across the thousands of cores in GPUs. For example, if 99% of code is parallel and it is run on 100 cores with ideal scaling, the remaining 1% that is serial will take over 50% of the total compute time. If one then runs the same calculation on 1000 cores, again assuming ideal scaling, the serial portion now consumes over 90% of the total compute time.

1.3.4.3 Race Conditions

Coding in parallel requires the programmer to consider the possible order of execution for multiple processes at any one time. It is important to be aware of some possible scenarios that may happen during execution with multiple processes.

Scenario 1: Process A relies on the result of a calculation carried out by process B. This is known as a dependency, more specifically a *data dependency*. Process A cannot continue to execute until process B has calculated the result. If it does, the answer will be incorrect since it does not have valid data from B. A solution would be to introduce synchronization between the processes prior to the operation that carries the dependency, or, alternatively, design a new parallel algorithm that eliminates the dependency.

Scenario 2: Process A and process B are both adding a number to the same variable X in memory. Process A and process B both take the current value of X as 5 and begin to add their result to the variable. Process A does $X = 5 + 2$, whereas process B does $X = 5 + 1$. Depending on which process is quicker at putting the new value of X back in memory, the result will be either 7 or 6. However, the desired value of X in memory is actually 8. This is known as a *race condition*. One solution would be to make use of a lock, which ensures that only one process can access any one variable at any time, thereby preventing a race condition by creating mutual exclusion of variable X. The use of a lock is often discouraged however, since it introduces serialization into the code. Various tricks can be used to circumvent this, such as the lock-free, warp-synchronized, approach used by the GPU version of the AMBER MD software discussed later (see Section 1.5.1).

1.3.4.4 Communication Metrics

While it might be possible to perfectly parallelize an algorithm, this does not necessarily mean that the algorithm will be executed at the theoretical limit of the hardware's performance. The reason for this is that all algorithms ultimately require some form of communication between each process (unless embarrassingly parallel). Some might require the reduction of a set of calculations, for example, a summation executed in parallel, while others might require comparison between values or the sharing of variables describing the system's state, for example, the coordinates of a set of atoms, between

processes. There are two major aspects to communication that determine how costly, in terms of time, the sharing of data between processes is. *Bandwidth* is a measure of the amount of data that can be communicated in a given timeframe, a.k.a. bit rate, that is, gigabytes per second. When measuring performance, the actual bandwidth (known as the *throughput*) can be compared with the theoretical bandwidth capability of the hardware to gauge the efficiency of communicating data. *Latency* is another metric to measure the performance of data movements; however, it is more concerned with the time it takes to communicate data rather than the amount of data it can communicate. Throughput and latency essentially measure the same thing: data communication efficiency. In simple terms, latency is the time taken to establish a communication channel, and bandwidth is the amount of data per unit time that can be sent along the channel. Algorithms that rely on large numbers of small packets of data will typically be latency-bound, while algorithms that send large packets of data infrequently will typically be bandwidth-bound. Since the design of parallel hardware typically involves a balancing act between bandwidth and latency, some algorithms will naturally perform better on certain types of hardware than others.

1.4 Parallel Computing on Graphics Processing Units

This section follows much the same order of concepts/theory as the previous section; however, it does so in the context of GPUs. First to be discussed is the memory model of GPUs. Following this will be a discussion of the available APIs and their compatible hardware, code suitability, and, finally, a discussion of scalability, performance and cost effectiveness.

1.4.1 GPU Memory Model

The memory model of a system equipped with GPU coprocessors does not easily fall into either of the distinct categories *shared* or *distributed*, as both the CPU(s) and the GPU(s) have their own memory spaces. The data required by a GPU in order to accelerate offloaded code from the CPU has to be accessible to the GPU. This involves the transferring of data from the CPU to the GPU, similar to the transfer of data from process to process in a *distributed* memory system. Once the data are on the GPU, the memory model is quite different. The main memory on a GPU, referred to as global memory, is accessible and visible to all cores/threads. This reflects the *shared* memory model. However, as a GPU has multiple memory locations (each with varying capacity, bandwidth, and latency depending on the proximity to the compute cores), the shared memory model is not consistent as one steps through the memory hierarchy. The memory hierarchy of a GPU will be discussed in more detail in Chapter 2. Taking advantage of the memory hierarchy is one of the key goals when writing GPU code and is crucial in achieving good performance. A GPU offers a hybrid memory platform giving the programmer control of data locality.

1.4.2 GPU APIs

At present there are two main APIs available to program GPUs: OpenCL and CUDA. A higher level GPU programming option, OpenACC, is also available, which is a collection of directives and programs, logically equivalent to OpenMP, that allows the compiler to analyze sections of code and automatically map loops onto parallel GPU cores. In this textbook, most examples and concepts will be discussed in terms of the CUDA programming paradigm, with some chapters making use of OpenCL. CUDA is freeware-supported, at the time of writing, only by Nvidia GPU hardware, whereas OpenCL is a cross-platform standard available and compatible on many different coprocessor architectures, not just GPUs. CUDA is a series of extensions to the C, C++, and Fortran

programming languages. This has the benefit of reducing the programming effort, as it is not formally required that algorithms be rewritten in order to run on a GPU, only modified to include the CUDA syntax. However, it is worth noting that the majority of algorithms need to be modified in order to execute efficiently on the many-core architecture of a GPU. Therefore, redesigning algorithms may be unavoidable for many applications to get the most performance out of a GPU as highlighted in the following. OpenCL offers a lower level set of extensions to the C programming language, providing the programmer with more control of the hardware. The programming model is very similar to CUDA's, once the terminology is stripped away. For instance, in the CUDA programming model, a *thread* is the sequence of instructions to be executed by a CUDA core, whereas in the OpenCL programming model a thread is termed a *work-item*. There are many pros and cons to both GPU API options; however, a discussion of these is beyond the scope of this textbook, and the choice of API is left to the programmer's preference. For ease of writing, the main focus of this chapter and the next will be CUDA, given its wide use in GPU quantum chemistry codes.

1.4.3 Suitable Code for GPU Acceleration

As GPUs were originally designed for applications involving graphics rendering, the lack of sophisticated hardware limited the number of applications suitable for acceleration using the historical GPU hardware design. The first programmers to attempt general-purpose applications on a GPU had the arduous task of representing their calculations as triangles and polygons. Although modern GPU hardware design has become more forgiving to the general-purpose application, it is still necessary to ensure that an algorithm is suited to the hardware in addition to considering Amdahl's or Gustafson's Law in order to achieve a performance gain.

Usually, only a selection of the entire code will be suitable for GPU execution, as illustrated in Figure 1.7. The rest of the code is executed on the CPU as normal, although the cost of transferring data between CPU and GPU often means that it is desirable to execute as much of the compute-intensive code as possible on the GPU.

Careful consideration of the desired code for porting is required before any development proceeds. Code that has been poorly assessed with respect to the hardware could see no performance improvement or even a performance decrease after porting to GPU, compared with running the entire code on a CPU. One aspect to consider is the SIMT (single-instruction, multiple-thread) architecture of GPUs and how that reflects on code suitability. Threads on a GPU are spawned in blocks (and on Nvidia hardware executed in batches of 32 termed warps), which are logically mapped to the physical GPU

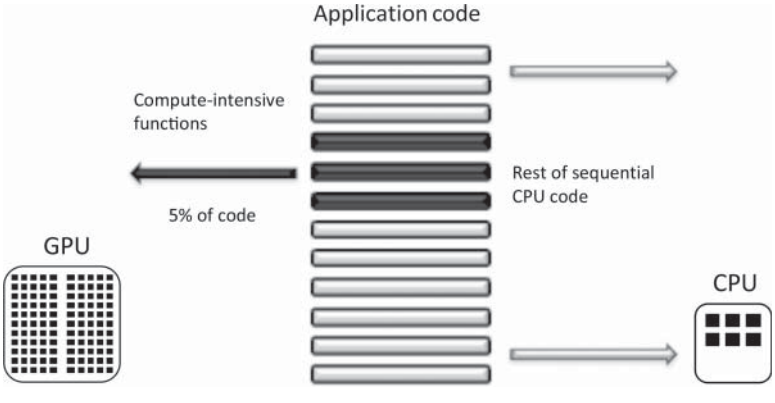


Figure 1.7 *Illustration of a GPU working in conjunction with a CPU as work is offloaded and computed on a GPU concurrently with CPU execution [24]*

cores. The threads are designed to be lightweight and are scheduled with no overhead. The hardware does not execute the threads in a block independently. Instead, all threads execute the same instructions, and conditional branching can thus be detrimental to performance because it holds up an entire block of threads. Because of the nature of the threads working concurrently within a thread block, it is beneficial to reduce the number of idle threads by creating many more threads than there are physical compute cores on the GPU. Communication between threads is also limited and can be a source of poor performance. Therefore, fine-grained parallel problems, such as those that are data-parallel with few dependencies between data elements and few divergent branches, most successfully exploit a GPU's hardware.

Another consideration when assessing a code's suitability for GPU acceleration is memory access latency and bandwidth. Memory access latency is a large overhead when porting to a GPU, as the GPU cannot access memory on the CPU at the same speed as the local GPU memory, and vice versa. Data have to be explicitly transferred, or some form of unified addressing has to be used, both of which introduce overheads. The bandwidth between the GPU and the CPU memory is much lower than the bandwidth between CUDA cores and GPU memory. For example, the theoretical global memory bandwidth for the Nvidia Tesla K40 card is 288 GB/s, and the theoretical unidirectional bandwidth between the CPU and the GPU is only 16 GB/s (PCIe Gen 3×16). Memory access through the PCIe interface also suffers from latencies on the order of several thousand instruction cycles. As a result, transferring data between the CPU and the GPU can cause a bottleneck in performance. Arithmetic-intensive code is required in order to hide the memory access latency with calculations. Well-suited code will have a high degree of arithmetic intensity, a high degree of data reuse, and a small number of memory operations to reduce the overhead of data transfers and memory accesses.

An additional method of hiding the memory access latency is to run code of a large problem size. In order to reduce the start-up costs of allocating and freeing memory when transferring data from CPU to GPU, it is beneficial to send one large data packet as opposed to several smaller data packets. This means that the performance of code with a large problem size will be affected less by the start-up costs of CPU–GPU data transfer, as a proportion of the overall execution time, compared to code with a smaller problem size incurring the same start-up costs.

A desirable feature of a code, where not all compute-intensive regions have been ported to the GPU, is the ability to be able to overlap code execution on both the CPU and the GPU. One aim when parallelizing any code is to reduce the amount of idle processor time. If the CPU is waiting for kernels to be executed on a GPU, and there are CPU-intensive portions, then the performance of the code will suffer as the CPU spends time idling.

1.4.4 Scalability, Performance, and Cost Effectiveness

The GPU's design makes it highly scalable, and in most cases effortlessly scalable, as the creation, destruction, and scheduling of threads are done behind the scenes in hardware. The programmer need not get bogged down with processor counts when planning a decomposition strategy. Algorithms can be decomposed according to the number of data elements, as opposed to the number of processors in a system, making applications effortlessly scalable to thousands of processor cores and tens of thousands of concurrent threads. The programmer need only concern himself/herself with the granularity of the decomposition strategy. The programming model abstracts multiple levels of parallelism from the hardware: that is, on the thread level, fine-grained data parallelism can be achieved, whereas on the kernel level a problem can be divided into more coarse-grained tasks. The ported GPU program will then scale to any number of processors at runtime. Not only does this result in a highly scalable programming/architectural model but it also lessens the difficulty for the programmer in designing scalable algorithms.

The main reason for using a GPU is to boost a program's performance. This can be done using multiple CPUs (as mentioned earlier), however, the scope for performance improvement using a GPU is often far greater for many applications due to the larger memory bandwidth of GPUs. The potential for performance improvement is demonstrated in the next section using two applications that have seen large performance improvements after GPU acceleration (see Section 1.5).

The way the performance of GPU-accelerated programs is assessed is crucial in providing end users with realistic speedups. Obviously it is beneficial to compare the performance of a GPU with a single CPU, but what about multiple CPUs? As the traditional approach to parallelizing code is through multiple CPU cores, it seems only fair that a comparison of these two parallel implementations (GPU vs multicore) should be made in order to assess the real gain from using a GPU. One important factor to include in a performance comparison with multiple CPUs is the cost effectiveness of the improvement. The cost effectiveness of using a GPU depends largely on the type of application and the scalability of the algorithm to many processor counts. Buying time on supercomputers can be costly, and building your own supercomputer is unfeasible for the majority. A GPU puts the processing power of a compute cluster within the reach of anyone willing to spend the time modifying the code. Not only do GPUs provide supercomputing power to desktop computers, but they also do it affordably thanks to their energy efficiency. It is no coincidence that the top 10 computers on the Green500 list (a list that ranks the world's supercomputers by energy efficiency) published in November 2013 contain Nvidia GPUs, as the performance per watt of a GPU is very desirable in comparison with a traditional CPU.

1.5 GPU-Accelerated Applications

Many real-word applications have already experienced massive performance improvements from GPU acceleration. Scientific domains benefiting from GPU technology include the defense industry, oil and gas, oceanography, medical imaging, and computational finance [25, 26]. This section showcases two GPU applications, in wildly different fields, not included in the later chapters of this book, providing additional motivation for parallelizing code on GPUs. The first to be discussed is the classical molecular dynamics (MD) software package Amber, and the second is the rendering and video-editing package Adobe Premiere(R) Pro Creative Cloud (CC).

1.5.1 Amber

Amber [27] is a software suite for molecular dynamics (MD) simulations for biomolecules, which can be used to study how biomolecules behave in solution and in cell membranes. It is used extensively, among others, in the drug discovery to determine the mechanism by which various drugs inhibit enzyme reactivity.

After porting and optimizing in CUDA for Nvidia hardware, the GPU implementation of the PMEMD molecular dynamics engine in the Amber software package is now the fastest simulation engine in the world on commodity hardware. For a simulation of the enzyme dihydrofolate reductase (DHFR), illustrated in Figure 1.8, containing approximately 23,500 atoms, it used to take, in 2009, one of the fastest supercomputers in the world to simulate ∼50 ns in one day. That is one full day of computer simulation for a mere 50 billionth of a second in the life of a biomolecule. Today, 5 years on, with a single desktop containing four Nvidia Tesla Kepler K40 or similar GPUs, an aggregate throughput of over 1 μs/day can be achieved. The GPU code design takes advantage of a patented approach to lock-free computing via the use of *warp-level* parallelism [28], which applies algorithmic

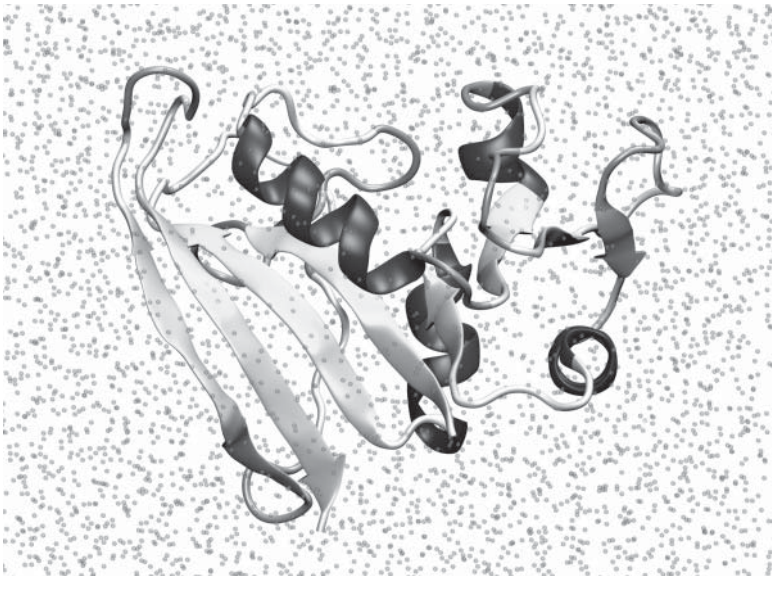


Figure 1.8 *Cartoon representation of the dihydrofolate reductase enzyme in water (23,508 atoms)*

tricks to eliminate the need for locks in parallel pairwise calculations. It also reduces the CPU–GPU data transfer overhead by carrying out the entire simulation on the GPU, communicating back to the CPU just for I/O, and uses PCIe peer-to-peer communication to eliminate CPU and motherboard chipset overhead when running across multiple GPUs. In order to achieve the best performance out of the Nvidia Kepler, Quadro, and GeForce series of GPUs, a mixed precision model, termed SPFP [29], was implemented to capitalize on the single-precision and integer atomic performance of the hardware without sacrificing simulation accuracy. The result is supercomputing capability that is accessible for every researcher at minimal cost – fundamentally advancing the pace of research involving MD simulation.

Example performance data for Amber v14 is shown in Figure 1.9, where the 90,906-atom blood clotting protein, FactorIX, has been simulated on various hardware. Amber 14 running on a single K40 GPU achieves a throughput of almost 40 ns/day, while using four K40 GPUs in tandem it can obtain a throughput on a single simulation of almost 74 ns/day. Compare this to the 5.6 ns/day achievable from running on two Intel Xeon E5-2670 CPUs (16 CPU cores), and the benefit of using GPUs to run Amber becomes obvious. A single GeForce GTX Titan Black (a gaming card that retails at around $1000) obtains over 42 ns/day and is thus capable of over 7× the performance of a single (16 cores) CPU node.

In addition to raw performance, there are also cost efficiency considerations for running Amber 14 on GPU(s), which can be demonstrated using the same FactorIX benchmark. The cost differential – considering for now just the acquisition costs and ignoring the running costs – between a high-end GPU workstation and the equivalent CPU-only cluster required to obtain the same performance is a factor of approximately 45×. This factor can be calculated, for example, by considering a GPU workstation consisting of a single node with four GTX Titan Black cards, costing approximately $7000, and the hardware specification of a CPU-only cluster required to obtain equivalent performance. The GPU workstation can achieve an aggregate throughput on four FactorIX simulations of 168 ns/day (42 ns/day per simulation). To obtain equivalent performance using

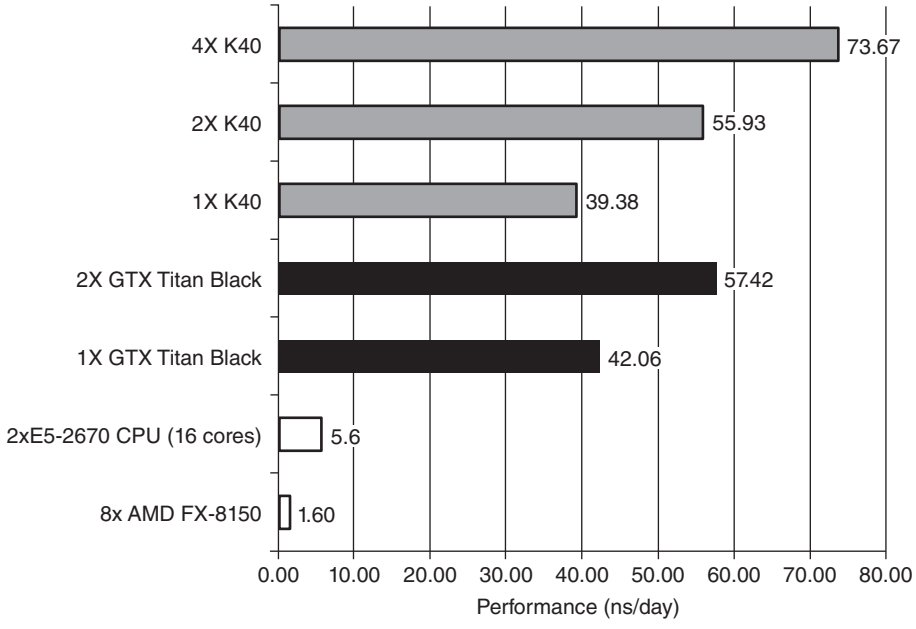


Figure 1.9 *Computational performance of classical molecular dynamics simulations with the CPU and GPU versions of Amber on varying hardware configurations measured in nanosecond per day. The results are for a standard benchmark (FactorIX enzyme in explicit solvent, 90,906 atoms, NVE ensemble, 2 fs time step) [30]*

CPU-based hardware, consisting of dual 8 core CPU nodes, requires a high-end QDR Infiniband interconnect between nodes and a total of 16 nodes per simulation (256 cores), since scaling across multiple nodes is never linear due to communication overhead, for a total cluster size of 64 nodes (for the four individual simulations). The cost of such a cluster, at the time of writing, is approximately $320,000.

Not only does using GPUs for MD simulations have a positive impact on acquisition costs but it also greatly reduces running costs, a substantial fraction of which, for large CPU clusters, is the cost of power. Saving in power also translates into more environmentally friendly computing. As mentioned in Section 1.4, a GPU is more energy efficient than a CPU, helping to promote greener science. To illustrate, compare a single FactorIX simulation running on a single Nvidia GeForce GTX Titan Black GPU against the same simulation running on a dual-socket Intel E5-2670 system. The Titan-Black GPU is rated at 250 W peak power consumption, but this does not include overhead in the node itself, such as case fans, hard drives, and the CPU idle power. Measuring the total node power consumption for the FactorIX simulation gives a power draw of approximately 360 W for a throughput of 42.06 ns/day. This equates to a power consumption of ∼0.20 kW h/ns of simulation. If, instead, one uses the Dual E5-2670 CPUs in this system, then the measured power consumption is approximately 359 W per node, with a single node yielding a throughput of 5.6 ns/day giving a power consumption of ∼1.54 kW h/ns of simulation. In this example, the GPU version of AMBER provides a 7.7× improvement in power efficiency.

In conclusion, the GPU-accelerated Amber MD program enables faster simulations or longer simulation times than on a CPU at greater power efficiencies. It includes the majority of the MD functionality that is available on a CPU but at a fraction of the financial and environmental cost.

1.5.2 Adobe Premier Pro CC

Adobe Premier Pro CC is a GPU-accelerated software suite used primarily to edit videos offering interactive, real-time editing and rendering. Adobe teamed up with Nvidia to create high-speed functionality though the use of Nvidia GPU cards and the CUDA programming platform. Included in the Adobe Premier Pro CC are features such as debayering (process of converting a Bayer pattern color filter array image from a single sensor to a full RGB image), feathered masking (applying a mask to a frame), and master clips effects (allowing cutting, sticking, and reordering of frames). Figure 1.10 shows the performance data for rendering a high-definition (HD) video with the complex Mercury Playback Engine at a resolution of 720p on Nvidia Quadro GPUs, with respect to a Dual Intel Xeon E5 processor (16 cores in total). The cost of the graphics cards range from around $500 for a Quadro K2000, offering nearly a 5× speed-up, to $5000 for a Quadro K6000, capable of rendering over 16× faster.

What is even more impressive about the GPU-accelerated Adobe suite is the speed at which ray-traced 3D scenes can be rendered through their After Effects CC product. By equipping a desktop with a single Quadro K6000, the user can benefit from ∼28× speedup when compared to using a Dual Intel Xeon E5 processor (16 cores in total) alone (see Figure 1.11). If the user happens to have an additional $5000 to spend on computer hardware, his or her 3D ray tracing workflow can be completed nearly 40× faster.

It is no surprise that a graphics rendering package such as the Adobe Premier Pro CC suite can achieve performance improvements of such a magnitude from GPU acceleration, given that the original design of a GPU was for such purposes. However, this serves as a great showcase of the capabilities of GPUs and their ability to accelerate applications in a way that is affordable to the masses. The purpose of this book is to explore in detail the successes that have been achieved and survey the current state of the art in accelerating electronic structure calculations on GPUs.

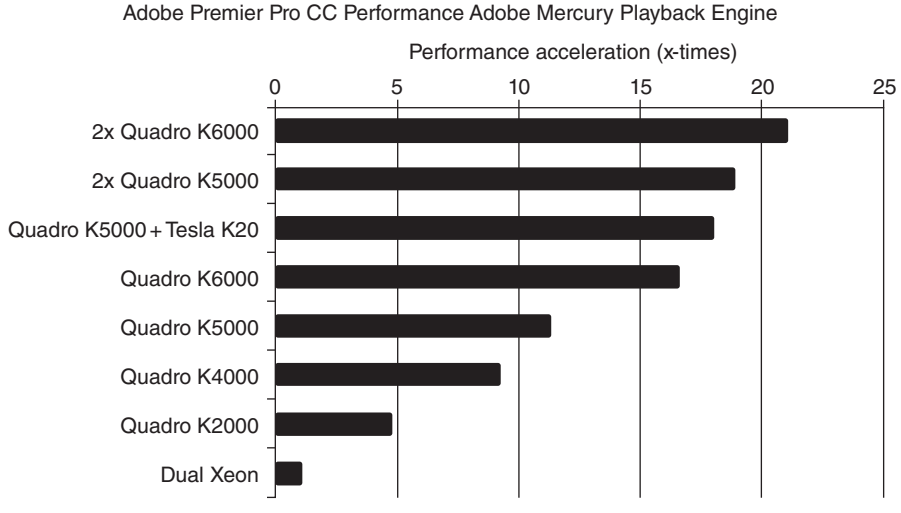


Figure 1.10 *Performance acceleration for Adobe Mercury Playback Engine on GPUs. System configuration: Adobe Premier Pro CC, Windows 7 – 64-bit, Dual Intel Xeon E5 2687 W 3.10 GHz CPUs (16 total cores). Test consists of HD video workflow with complex Mercury Playback Engine effects at 720p resolution. Results based on final output render time comparing noted GPU to CPU [31]*

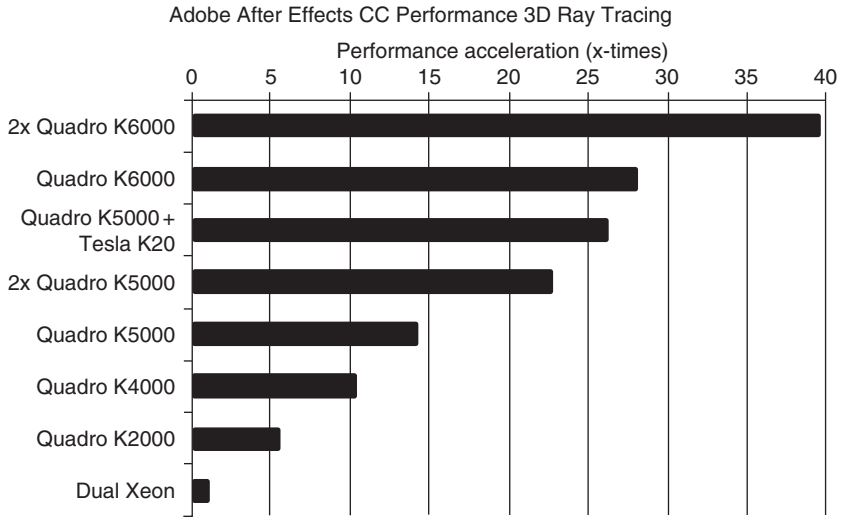


Figure 1.11 *Performance data for Adobe After Effect CC Engine on GPUs. System configuration: Adobe After Effects CC, Windows 7 – 64-bit, Dual Intel Xeon E5 2687 W 3.10 GHz CPUs (16 total cores). Test consists of live After Effect CC scenes with 3D layer, comparing time to render ray-traced 3D scene on noted GPU versus CPU [32]*

References

1. Bolcer, J.D. and Hermann, R.B. (2007) *The Development of Computational Chemistry in the United States*, Reviews in Computational Chemistry, John Wiley & Sons, Inc., pp. 1–63.
2. Rojas, R. and Hashagen, U. (eds) (2002) *The First Computers: History and Architectures*, MIT Press, Cambridge, MA, USA.
3. Rojas, R. (1997) Konrad Zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing*, **19** (2), 5–16.
4. Hahn, M. *IBM Electronic Data Processing Machine 2010.* Available from: `http://grin.hq.nasa.gov/ABSTRACTS/GPN-2000-001881.html` (accessed 7 August 2014).
5. Garg, R. *Exploring the Floating Point Performance of Modern ARM Chips*. Available from: http : // www.anandtech.com / show/6971/exploring-the-floating-point-performance-of-modern-arm-processors (accessed 12 May 2014).
6. Weik, M.H. (1964) *Burroughs D825*, Ballistic Research Laboratories, Aberdeen Proving Ground, Maryland.
7. Clementi, E., Corongiu, G., Chin, J.D.S., and Domingo, L. (1984) Parallelism in quantum-chemistry – Hydrogen-bond study in DNA-base pairs as an example. *International Journal of Quantum Chemistry*, **18**, 601–618.
8. Guest, M., Harrison, R., van Lenthe, J., and van Corler, L.H. (1987) Computational chemistry on the FPS-X64 scientific computers. *Theoretica Chimica Acta*, **71** (2–3), 117–148.
9. Colvin, M., Whiteside, R., and Schaefer, H.F. III (1989) Quantum chemical methods for massively parallel computers, in *Methods in Computational Chemistry*, vol. **3** (ed S. Wilson), Plenum, NY, p. 167.

10. Janssen, C.L. and Nielsen, I.M.B. (2008) *Parallel Computing in Quantum Chemistry*, CRC Press, Taylor & Francis Group, Boca Raton.
11. Dupuis, M. and Watts, J.D. (1987) Parallel computation of molecular-energy gradients on the loosely coupled array of processors (Lcap). *Theoretica Chimica Acta*, **71** (2–3), 91–103.
12. Whiteside, R.A., Binkley, J.S., Colvin, M.E., and Schaefer, H.F. III (1987) Parallel algorithms for quantum chemistry. 1. Integral transformations on a hypercube multiprocessor. *Journal of Chemical Physics*, **86** (4), 2185–2193.
13. Watts, J.D. and Dupuis, M. (1988) Parallel computation of the Moller–Plesset 2nd-order contribution to the electronic correlation-energy. *Journal of Computational Chemistry*, **9** (2), 158–170.
14. *MPI: A Message-Passing Interface Standard Version 3.0, Message Passing Interface Forum*, September 21, 2012. Available from: `http://www.mpi-forum.org` (accessed 12 September 2014).
15. Moore, G.E. (1998) Cramming more components onto integrated circuits (Reprinted from Electronics, pp. 114–117, April 19, 1965). *Proceedings of the IEEE*, **86** (1), 82–85.
16. Simon, W.G.. *Transistor Count and Moore's Law 2011*. Available from: `http://commons.wikimedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg` (accessed 12 August 2014).
17. Dubash, M. (2005) *Moore's Law is Dead, says Gordon Moore*. Available from: `http://news.techworld.com/operating-systems/3477/moores-law-is-dead-says-gordon-moore/` (accessed 12 August 2014).
18. Mandy, P. (2011) *Microprocessor Power Impacts*, Intel.
19. Xu, D., Williamson, M.J., and Walker, R.C. (2010) Advancements in molecular dynamics simulations of biomolecules on graphical processing units. *Annual Reports in Computational Chemistry*, Vol. **6**, 2–19.
20. Hager, G. and Wellein, G. (2011) *Introduction to High Performance Computing for Scientists and Engineers*, CRC Press, Taylor & Francis Group, Boca Raton.
21. *The OpenMP API Specification for Parallel Programming*. Available from: `http://openmp.org/wp/openmp-specifications/` (accessed 13 September 2014).
22. Amdahl, G.M. (1967) *Validity of Single-Processor Approach to Achieving Large-Scale Computing Capability*. Proceedings of AFIPS Conference, Reston (VA), pp. 483–485.
23. Gustafson, J.L. (1988) Reevaluating Amdahl's law. *Communications of the ACM*, **31** (5), 532–533.
24. Nvidia Corporation (2014) *What is GPU Computing?*. Available from: `http://www.nvidia.com/object/what-is-gpu-computing.html` (accessed 12 August 2014).
25. Hwu, W.-M.W. (ed.) (2011) *GPU Computing Gems – Emerald Edition*, Morgan Kaufmann, Burlington, MA, USA.
26. Hwu, W.-M.W. (ed.) (2012) *GPU Computing Gems – Jade Edtion*, Morgan Kaufmann, Waltham, MA, USA.
27. Case, D.A., Babin, V., Berryman, J.T., *et al.* 2014 *AMBER 14. University of California, San Francisco*. Available from: `http://ambermd.org` (accessed 17 October 2014).
28. Le Grand, S. (2012) *Computer-Readable Medium, Method and Computing Device for N-body Computations Using Parallel Computation Systems*. Google Patents.
29. Le Grand, S., Götz, A.W., and Walker, R.C. (2013) SPFP: Speed without compromise – A mixed precision model for GPU accelerated molecular dynamics simulations. *Computer Physics Communications*, **184** (2), 374–380.

30. Walker, R.C. (2014) *Amber 14 NVIDIA GPU Acceleration Support: Benchmarks*. Available from: `http://ambermd.org/gpus/benchmarks.htm` (accessed 12 August 2014).
31. Nvidia Corporation (2014) *Adobe Premier Pro CC*. Available from: `http://www.nvidia.com/object/adobe-premiere-pro-cc.html` (accessed 17 August 2014).
32. Nvidia Corporation (2014) *Adobe After Effects CC*. Available from: `http://www.nvidia.com/object/adobe-premiere-pro-cc.html` (accessed 17 August 2014).

2

GPUs: Hardware to Software

Perri Needham[1]*, Andreas W. Götz*[1] *and Ross C. Walker*[1,2]

[1]*San Diego Supercomputer Center, UCSD, La Jolla, CA, USA*
[2]*Department of Chemistry and Biochemistry, UCSD, La Jolla, CA, USA*

When programming GPUs, a thorough understanding of how the programming model is abstracted from the hardware is imperative to achieving good performance. This chapter discusses the architectural design of GPUs, programming models and how they map to hardware, and basic GPU programming concepts. This is followed by an overview of the currently available GPU-accelerated software libraries as well as design features of modern GPUs that simplify their programming and make attaining good performance easier.

It is beyond the scope of this book to teach details of how to program GPUs. For a grounding in GPU programming, we recommend reviewing the reference manuals and programming guides (along with associated training textbooks) for the various platforms available. Good introductions are given in the following text books: *CUDA By example – An introduction to general-purpose GPU programming* by J. Sanders & E. Kandrot and *Programming Massively Parallel Processors – A hands-on approach* by D. B. Kirk & W. M. W. Hwu.

At the time of writing, there are multiple different approaches to programming GPUs with CUDA [1, 2], OpenCL [3] and OpenACC [4] being the most popular. In the later chapters of this book, examples are given of the use of all these approaches. However, covering all of these methods and the differences in GPU hardware between manufacturers (e.g., AMD and Nvidia) is beyond the scope of this book. The underlying considerations about parallel constructs, diverse memory models, and CPU-to-GPU communication are the same regardless of which programming model or GPU platform is used. For the purposes of providing a concise introduction to GPU programming in a way that can be easily followed by someone who is not familiar with many-core programming, we will confine our discussion here to the most common approach to GPU programming. At the time of writing, this means the use of the CUDA programming language on Nvidia GPU hardware.

Electronic Structure Calculations on Graphics Processing Units:
From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

2.1 Basic GPU Terminology

Before going into the details on GPU architecture and programming, it is important to first clarify some of the terminology used throughout this chapter. For a more thorough list of terminology, refer to the glossary at the beginning of this book.

Host – the CPU that the GPU is coupled to.
Device – the GPU coprocessor that is coupled to a CPU.
Cores – the processing elements of a GPU (or CPU) that execute a sequence of instructions (*thread*).
SM or *streaming multiprocessor* – responsible for running GPU kernels and comprises a subset of the total GPU cores, registers for each residing core, a shared memory, texture cache, constant cache, L1 cache, and warp schedulers. A GPU is organized into multiple streaming multiprocessors (SMs).
Thread – a set of instructions to be executed sequentially by a GPU (or CPU) processing element (compute core).
Warp – a set of 32 SIMD threads (as currently implemented in CUDA-enabled architecture) that execute the same instructions simultaneously in hardware. Instructions are issued to and scheduled for execution on an entire warp of threads at a time, which step through the instructions in lockstep.
Kernel – a procedure that is executed on the SMs of a GPU, typically by thread counts in the region of tens of thousands in parallel. Instructions of a kernel are scheduled in groups of a warp in a single-instruction multiple-thread (SIMT) manner.

2.2 Architecture of GPUs

As illustrated in Figure 2.1, GPU architecture differs from CPU architecture in that in place of a few processing cores with sophisticated hardware, that is, multilevel caching technologies, prefetching, and branch prediction, there are thousands of simplistic compute cores that operate in lock step. This so-called SIMT model is reminiscent of traditional SIMD architectures in CPUs, but

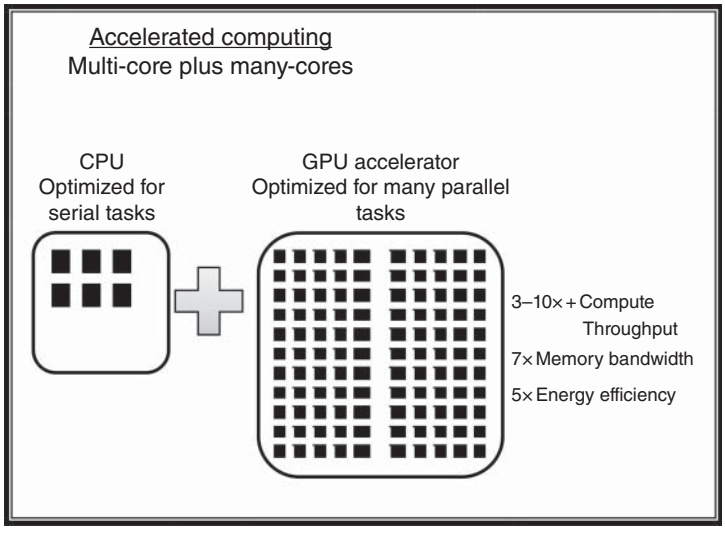


Figure 2.1 *General overview of CPU–GPU hardware configuration. Picture adaptation courtesy of Nvidia Corporation*

allows for branch divergence between threads, making programming easier. This architecture makes GPUs optimal for many data-parallel tasks, as they offer substantially higher compute throughput as well as very high memory bandwidth by virtue of the ability to vectorize loads/stores from/to memory.

2.2.1 General Nvidia Hardware Features

A GPU *device* is connected to a *host* via the peripheral component interconnect express (PCIe) bus, and data are passed between host memory and device memory as illustrated in Figure 2.2. The GPU is partitioned into streaming multiprocessors (SMs) where each SM contains GPU cores, double-precision unit(s), special function units, registers, and a multithreaded instruction unit for scheduling warps. A warp is an important concept when optimizing GPU code. At the time of writing, a warp represents a group of 32 threads that are scheduled to execute together and operate in lock step. The concept of a warp will be discussed in more detail shortly. Each SM operates by a SIMT paradigm, whereby the SM issues one set of instructions to a warp of threads for concurrent execution on different data elements.

In addition to the global memory on the device, there are four other types of on-chip memory accessible to each GPU SM. Each thread has access to local memory, which is part of the dynamic random access memory (DRAM) and is private to each thread. Each SM also has access to a small amount (48 KB per SM in Nvidia K40 GPUs [1]) of high-speed shared memory accessible to the threads spawned on that SM and them alone. Constant memory and texture memory are artifacts of programming for graphical applications and provide small read-only memory locations with fast complementary caches. It is important to keep in mind that global memory accesses while providing high bandwidth incur long latencies on the order of several hundred clock cycles. This also holds for local memory, which is used by memory-intensive kernels if the amount of register memory required by a thread exceeds that available (255×32 bit values on Nvidia K40 GPUs [1]). Some modern GPUs are also equipped with a small amount of L1 and L2 cache, which is used to speed up memory accesses to global/local memory.

2.2.2 Warp Scheduling

CUDA-enabled GPUs execute instructions in a SIMT manner, whereby a block of threads is grouped into *warps* by a scheduler. At the time of writing, a *warp* comprises 32 threads. All threads of a warp start at the same instruction to execute in parallel with the other threads in a warp. The warp scheduler

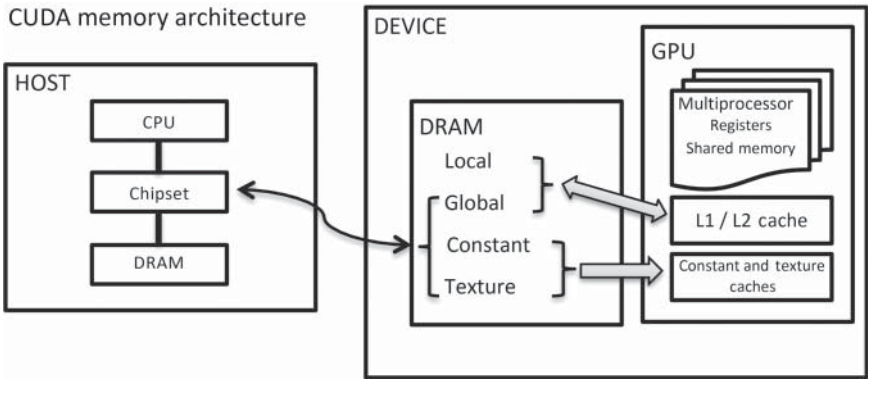


Figure 2.2 *Illustration highlighting the movement of data between memory locations on the host and the device. Picture adaptation courtesy of Nvidia Corporation*

issues one common instruction to be executed by all threads in a warp at a time. Ensuring blocks of threads are a multiple of 32 (a warp) prevents underpopulation of warps and therefore idle cores. The SIMT offers not only thread-level parallelism but also instruction-level parallelism, ensuring that instructions are pipelined to the cores in a constant stream by scheduling the next instruction to an active warp of threads. Threads are able to branch and diverge away from other threads, as each thread keeps track of its own instruction address and register state; however, divergent branching will cause all other threads in a warp to be idled (masked to null operations, no-ops) until the divergent branch is complete and threads are back on the same execution path. For this reason, a GPU is used most effectively when there are no divergent branches, that is, conditional statements, in a warp of threads and hence no idle cores waiting for threads to catch up.

2.2.3 Evolution of Nvidia Hardware through the Generations

The evolution of Nvidia GPU hardware can be tracked by *compute capability* (c.c.), which is a major and minor revision number linked to core architecture design. GPUs with the same major revision number are of the same hardware generation. Minor revision numbers reflect incremental improvements to the core architecture, that is, new features. For a complete explanation of the differences between compute capabilities, refer to the *CUDA Programming Guide* [1].

The architectural design of the latest generation Nvidia hardware (Kepler – compute capability 3.x) is shown in diagrammatic form in Figure 2.3. Each SM is equipped with registers, shared memory (SMEM), read-only and L1 instruction and data cache. All SMs have access to an L2 cache, which is responsible for reducing the latency of global memory accesses.

2.3 CUDA Programming Model

In the CUDA programming model, functions within the code are offloaded to the GPU as *kernels*. The workload of each kernel is decomposed into *N* independent workloads, which are then executed

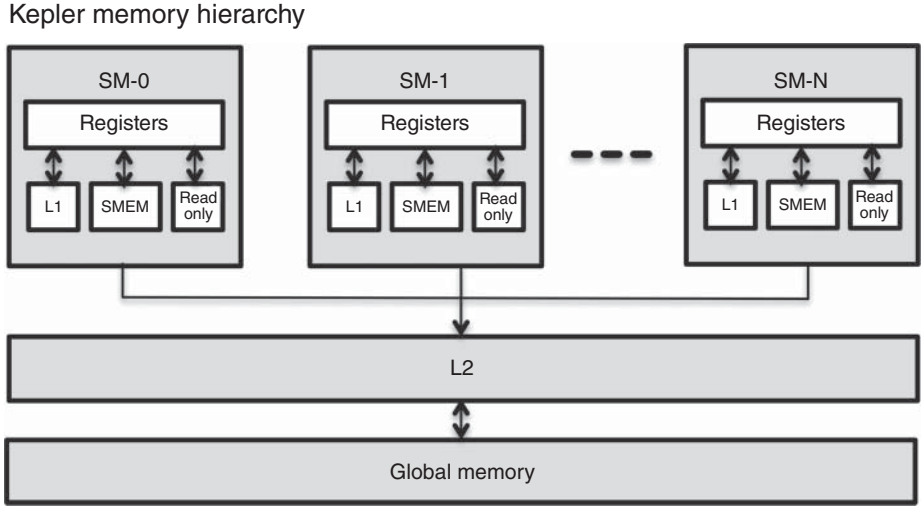


Figure 2.3 *Nvidia Kepler architecture memory model. Global memory and L2 cache can be accessed by all SMs. L1 cache and shared memory (SMEM) are accessible to threads running on the same SM. Picture adaptation courtesy of Nvidia*

in parallel by *N* threads, where *N* is typically of the order of several thousands. Threads and memory locations are structured into hierarchies that map logically to the underlying hardware. In order to extract the most performance out of a GPU, a solid understanding of the programming model is needed. In the following description, the CUDA programming model is logically sectioned into kernels, thread hierarchy, memory hierarchy, and warp scheduling.

2.3.1 Kernels

A *kernel* is a function to be executed in parallel by *N* threads on a GPU. A pre-existing function call in C, C++, or Fortran code is easily modified to execute on a GPU. The syntax is the same across different programming languages. To launch a kernel in C for CUDA a declaration specifier *__global__* marks the function to be executed in parallel on a GPU along with specialist *execution configuration* syntax in the kernel call to spawn threads, *⋘dimGrid,dimBlock⋙* (see Figure 2.4).

The *execution configuration* syntax (in triple angle brackets) specifies the number of threads to be launched, including information on how the threads will be mapped to the hardware. The first variable in the execution configuration defines the number of blocks, and the second variable the number of threads in a block. In the example of Figure 2.4, one block with *N* threads would be launched. We will explain this further after discussing the thread hierarchy in the following.

An important characteristic of a kernel call is its asynchronous nature. When the host's execution path hits a kernel call, the runtime library spawns the requested number of threads on the device for execution of the device function. While this is going on, control is immediately passed back to the host. The host is free to continue on its execution path and will only wait for the device to complete kernel execution if explicit synchronization is requested. Asynchronous and synchronous execution will be discussed in Section 2.4.

2.3.2 Thread Hierarchy

Threads are organized into blocks of threads within a grid of threadblocks (Figure 2.5).

```
//Kernel definition
__global__ void Kernel_add(float* A, float* B, float* C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}

int main()
{
...
//Kernel invocation with N threads
Kernel_add <<< 1, N >>> (A, B, C);
...
}
```

Figure 2.4 *Example of C for CUDA kernel code and kernel launch code. The declaration specifier __global__ marks the kernel for execution on the GPU, while the execution configuration (grid and block size) is contained in triple angle brackets*

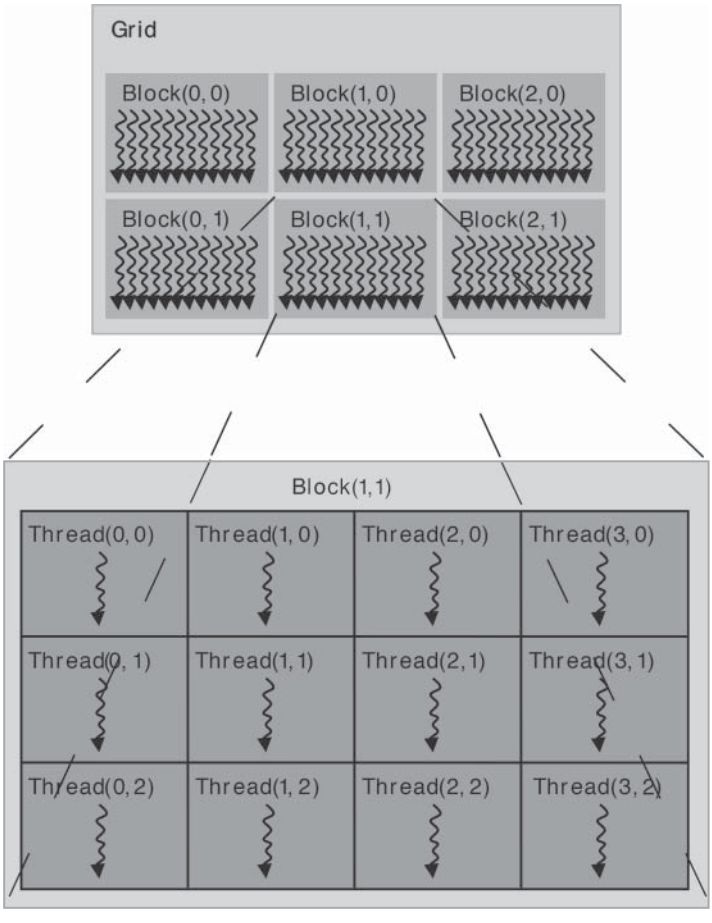


Figure 2.5 *Thread hierarchy. Blocks of threads are configured into a grid of thread blocks that logically map to the underlying GPU hardware with blocks being executed on individual SMs. Picture adaptation courtesy of Nvidia [1]*

The number of threads spawned at launch time is usually dictated by the problem size or number of processors, but many more threads can be created than there are compute cores in hardware, and this is in fact necessary to achieve good performance. This configuration allows data structures, such as matrices and vectors, to be easily mapped to hardware. It also makes the code amenable to different hardware configurations with different numbers of SMs and compute cores and thus future-proof.

Blocks of threads can have up to three dimensions, as can a grid of threadblocks. This enables each element of a three-dimensional array to be mapped to an individual thread either in a block or in a grid, promoting fine-grained data parallelism. The size and shape of a block of threads and/or a grid of threadblocks can be specified at execution configuration using the intrinsic type **dim3**. For example, one would specify a two-dimensional block with *ndimx* by *ndimy* threads using the syntax: *dim3 threadsPerBlock(ndimx, ndimy)*.

Threads within a thread block can be easily indexed in device code by a convenient three-component vector built into the CUDA programming language – *threadIdx*. For example, for

two-dimensional thread blocks (see Figure 2.5), each thread can access its thread indices within a block as $tx = threadIdx.x$ for the x-dimension and as $ty = threadIdx.y$ for its y-dimension. A similar three-component vector is also supplied for indexing blocks in a grid and is used in exactly the same manner – *blockIdx*. These built-in vectors allow for local indexing of threads in a block and can be easily used to calculate a thread's ID *tid* within a block as

$$tid = tx + ty \times ndimx + tz \times ndimx \times ndimy \tag{2.1}$$

where *tx*, *ty*, and *tz* are the thread index in the x-, y-, and z-dimension, respectively, and *ndimx*, *ndimy*, and *ndimz* are the number of threads in the respective dimension of a block, also referred to as the extent of a dimension.

To determine the global index of a thread in a grid is also straightforward and just requires the block ID in the corresponding dimension:

$$global\_x = blockIdx.x \times ndimx + threadIdx.x \tag{2.2}$$

and is equivalent for x- and y-dimension. Using Thread(0, 0) in Block(1, 1) from Figure 2.5 as an example, the global indices are $global\_x = 1 \times 4 + 0 = 4$ and $global\_y = 1 \times 3 + 0 = 3$. These variables can then be used to calculate the thread's global thread ID by switching the extent of a block in any given dimension with the extent of a grid in any given dimension in Eq. (2.1). So, it is possible to identify a thread in four ways: local index, local ID, global index, and global ID.

There are restrictions on the total number of threads in a block of threads. Threads within the same thread block are executed on the same SM. Communication between threads in the same block is made possible through the use of shared memory (which is the equivalent of an L1 cache) and synchronization statements. However, the requirement that all threads in a block have to be on the same SM does create a restriction on the number of threads in a block. Currently, a maximum of 1024 threads per block are allowed due to the resources available to each SM [1]. Since threads in a thread block are executing in lockstep in units of a warp, the block size should be a multiple of the warp size to avoid threads being masked to null operations (no-ops) and thus wasting resources.

2.3.3 Memory Hierarchy

To expand on what was introduced above, there are five separate memory spaces that CUDA GPU threads have access to: global memory, constant memory, texture memory, shared memory, and local memory. These memory spaces differ in their size and locality, that is, whether they are accessible by the host or not and whether all threads can access them, as shown in Table 2.1. Memory access latencies differ by several orders of magnitude, which needs to be accounted for when programming. The programmer has control over managing global, constant, texture, and shared memory; however, data are allocated to local memory by the compiler. Memory is managed through calls to the CUDA runtime library. Memory transfers between the host and the device are performed in the host code. These transfers are rather slow with latencies on the order of several thousand instruction cycles.

Local memory is managed by adjusting the number of registers being used per core at compile time. It is private to each thread and therefore lasts the lifetime of the thread. Shared memory is managed in device code and is private to each block of threads; therefore it has the same lifetime of the thread block. Threads within the same block can communicate through the use of shared memory and synchronization statements. Constant memory and texture memory are managed in host code; they are read-only memory spaces and both have a lifetime of the application. Global memory is allocated and deallocated by the host, can be accessed by all threads, and has the lifetime of the application unless it is explicitly deallocated in the host code.

Table 2.1 *Memory hierarchy in the CUDA programming model detailing the location, lifetime, and privacy of the five memory spaces*

Memory	Location	Lifetime	Privacy
Global	Device memory accessed via L2 cache	Application/ deallocation	All threads
Constant (read-only)	Device memory accessed via constant cache	Application/ deallocation	All threads
Texture (read-only)	Device memory accessed via texture cache	Application/ deallocation	All threads
Shared	Shared memory on each SM	Kernel	Private to each block of threads
Local	Device memory accessed via L2 cache	Thread	Private to each thread

2.4 Programming and Optimization Concepts

GPU programming adds new concepts to the fundamentals of parallel programming. GPU programming can be broken down into two stages: porting and optimization. The first task for the programmer should be to get the CPU code running on a GPU and giving the correct results, although the effort required in later stages can be reduced if the code is not just naively ported but, instead, the algorithm and approach is carefully tailored to match the GPU architecture. This initial porting process results in GPU portions of the program. Once the GPU code has been validated, the code needs to be optimized in order to fully utilize a GPU's hardware and squeeze as much performance out as possible. There are three key strategies for improving code performance on GPU architectures that need to be considered during porting and optimization:

Increasing parallel efficiency – achieving the maximum parallel execution in order to fully utilize the GPU architecture by ensuring a program exposes as much parallelism as possible.

Improving memory throughput – making the best use of the different memory locations on a GPU in order to reduce or hide memory access latency.

Improving instruction throughput – feeding the cores with a constant flow of instructions in order to reduce the amount of time cores spend idle.

In this section, GPU programming concepts are introduced, and an explanation of how they can be managed in order to influence performance, with respect to the three strategies outlined above, is given.

2.4.1 Latency: Memory Access

Memory access latency varies from one memory location to another. The highest latency memory access is data transfers between the host and the device, which is on the order of thousands of clock cycles. On the GPU global memory and local memory are the slowest memory locations, as they reside in device memory (DRAM) that has the highest latency of several hundred clock cycles (and lowest bandwidth). Constant memory and texture memory also reside in device memory but as they come armed with their own cache, accesses can be as fast as constant/texture cache latency, which can be considerably quicker than global device memory accesses. As shared memory is on-chip, it is the fastest programmer-managed memory location. There is also an on-chip L1 cache and L2 cache on devices of compute capability 2.x and higher, which offer lower latency/higher bandwidth memory accesses [1]. The latency/bandwidth of these memory locations varies from architecture to architecture, so exact numbers are not given here.

In order to maximize *memory throughput*, it is important not only to reduce the amount of host–device data transfers and utilize as much on-chip low latency/high bandwidth memory as possible but also to organize/access data in memory accordingly, as explained in the following.

To reduce host–device data transfers, one can try porting intermediate CPU code between GPU kernels. This can sometimes reduce data transfers between the host and device, in which case, even if porting the CPU code does not impact performance the reduction in data transfers might. Other methods of reducing data transfer overheads are using page-locked host memory and mapped page-locked memory.

Effective utilization of on-chip memory can mean transferring data from global memory to shared memory. For example, if an array is to be used multiple times by a block of threads, it may benefit performance if the threads load the array into shared memory – each thread fetches a portion of the array from global memory to the shared memory space. Thread synchronization must occur after data fetch to ensure that all threads have finished and hence the entire array is in shared memory. The threads are then free to operate on the shared memory array, benefiting from lower latency. However, this is worthwhile only if the elements of the array are to be used multiple times, as the performance of the initial transfer of data to shared memory is still limited by the latency/bandwidth of device memory. Once threads have completed working on the array, it can be transferred back to device memory in the same way. It may be necessary to synchronize threads prior to the transfer to ensure that all threads have finished updating the array. Read-only data can also be placed in constant or texture memory spaces to gain from their lower latency/higher bandwidth cache.

2.4.2 Coalescing Device Memory Accesses

To understand the concept of coalesced device memory accesses, one must first understand how data are accessed in memory. A data transaction can be of size 32, 64, or 128 bytes. The size of a data transaction depends on the amount of data requested, the location of the data in memory, and the compute capability of the device. Memory addresses of a single data transaction can span a maximum of 128 consecutive bytes of device memory. When a warp (or half-warp) of threads requests data from device memory, the request is *coalesced* into as few transactions as possible so as to increase the throughput of the data access. It is beneficial to performance to assist in reducing the number of memory transactions per data request as much as possible. The ability of a device to coalesce memory accesses varies between the different compute capabilities. The rules on coalescing memory accesses become more relaxed for devices of higher compute capabilities. Details of the coalescing behavior of each architecture can be found in the CUDA programming guide [1]. Coalescing memory requests into as few data transactions as possible can dramatically improve the performance of a GPU program. Aligning memory accesses and organizing data in memory accordingly can ensure that device memory reads/writes are coalesced.

2.4.3 Shared Memory Bank Conflicts

The efficiency of shared memory is reliant on there being no bank conflicts. Banks are equally sized sections of shared memory, which are accessed simultaneously by the threads in a half warp. Bank conflicts are where more than one memory address request points to the same bank of shared memory. In this eventuality, the memory accesses are serialized, with significant effects on performance. An exception is if more than one thread of a warp requests data in the same memory location in a shared memory bank, which results in a broadcast of some form dependent on the compute capability of the device. The memory banks are organized into words that vary in size depending on the compute capability of the device. Figure 2.6 shows a half-warp of threads ideally accessing 16 consecutive banks in shared memory.

Reducing bank conflicts is the key to achieving high *shared memory throughput* and taking advantage of the on-chip low latency/high bandwidth.

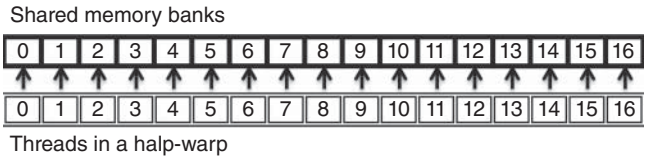


Figure 2.6 *Half-warp of threads accessing 16 shared memory banks*

2.4.4 Latency: Issuing Instructions to Warps

The number of clock cycles a warp scheduler takes to issue an instruction to a warp varies depending on a device's compute capability and the instruction being issued. This is known as the *latency*. In the interest of keeping each streaming multiprocessor as busy as possible, that is, utilizing a GPU to its fullest by *increasing parallel efficiency*, the latency involved in issuing an instruction should be hidden. To achieve this, all warp schedulers must issue an instruction to a warp on every clock cycle of the latency period, completely hiding the latency period. This relies on their warps being ready to execute on every clock cycle, creating a direct link between the number of warps and *parallel efficiency*. The number of warps in a kernel is controlled through the number of threads and blocks declared by the execution configuration on kernel launch and are measured by *occupancy*.

2.4.5 Occupancy

The ratio of the number of *active* warps to the *maximum* number of warps on an SM is known as the *occupancy*. Calculating the occupancy offers the programmer some insight into the *parallel efficiency* of a given execution configuration, that is, whether or not a GPU is being used to its maximum. Achieving the optimum occupancy is a balancing act since one needs to ensure that each thread in a block has enough resources, preventing register spillages, that is, data overflowing from registers into high latency/low bandwidth device (local) memory, while keeping the multiprocessor busy enough to *maximize instruction throughput* through thread-level parallelism. It is not always beneficial to have too high an occupancy ratio. The optimal level of occupancy can be dependent on whether a kernel is memory bandwidth-bound. The more bandwidth bound a kernel, the more likely a higher occupancy will benefit, as increasing the number of active warps on a multiprocessor gives the scheduler the opportunity to hide expensive device (global) memory accesses behind computation. Too low an occupancy ratio for a bandwidth-bound kernel will result in poor performance. On the other hand, if a kernel is not bandwidth-bound because it has a high compute to data ratio, a high occupancy ratio could have an adverse effect on performance due to more register spills, divergent branches, and additional instructions. Factors that may affect the number of active warps on a multiprocessor are the following:

- Amount of shared memory required per block of threads – shared memory is a limited on-chip resource, so only as many warps can be active on a multiprocessor as there is shared memory available.
- Register usage – as with shared memory, there are only a limited number of registers on each SM. This has a daisy-chain effect on the number of warps able to be scheduled at any one time. The compiler attempts to find a balance between the number of active warps on an SM and the amount of register usage/spillage. It is possible to control register usage by the compiler flag `maxrregcount` or through launch bounds (hints to the compiler to assist in optimal register usage).

- Number of threads in a block – an SM is limited to 16 active blocks of threads at any one time (Kepler architecture) [1]; therefore having a small number of threads per block may be detrimental to performance, as it could result in a low occupancy ratio and the total number of threads on a SM not saturating the hardware.

As each application is unique, there is no exact execution configuration that will produce the optimal occupancy for every code. An occupancy calculator is available from Nvidia [5], which can aid in choosing the right configuration for a particular code. Experimentation with varying numbers of threads per block, blocks per grid, maximum register counts, and amount of shared memory allocation is recommended to fine-tune the performance of a kernel/application.

2.4.6 Synchronous and Asynchronous Execution

Synchronization of threads can be crucial in preventing race conditions and out-of-order memory reads/writes and in maintaining the accuracy of a kernel. However, it can also harm performance by reducing *instruction throughput* and *parallel efficiency* as threads become idle. There are two main types of synchronization:

__syncthreads() – a function call in device code that causes all threads in a block of threads to synchronize.

cudaDeviceSynchronize() – a function call in host code that causes all threads on the device to synchronize.

The *device code synchronization* step (__syncthreads()) might be used in the case where two threads in a block of threads need to communicate. The *host code synchronization* step (cudaDeviceSynchronize()) might be used if there is a dependency between two data elements that is seen by two different blocks of threads. It is in the interest of performance that as few synchronization functions are used as possible, particularly the global synchronization function in host code. An algorithm should be mapped to the CUDA programming model so that threads needing to communicate with each other reside in the same block where possible to reduce the overheads of placing data back in the high-latency/low-bandwidth global memory between kernel calls and the overhead of kernel invocation after global synchronization.

Asynchronous execution can be employed to help expose as much parallelism as possible. At the application level, asynchronous functions or streams (streams are discussed below) can be used to maximize concurrent host–device execution, for instance, copying data to the device asynchronously enabling the CPU to overlap computation with data transfers. At the device level, asynchronous functions can be used to squeeze as much parallelism as possible out of an SM by feeding it with multiple kernels, giving the warp scheduler plenty of options when issuing the next instruction to an active warp. Invocation of multiple kernels executing concurrently on a GPU is possible only on devices of compute capability 2.x and higher [1].

2.4.7 Stream Programming and Batching

A *stream* is an in-order sequence of instructions to be executed on a GPU. Multiple streams of instructions can be created. The order in which different streams execute is not sequential, enabling asynchronous concurrent execution of kernels and functions. On devices of compute capability 2.x and higher, multiple kernels can be launched concurrently through the use of streams [6]. If two streams are launched, the warp scheduler has the option of issuing the next instruction from stream 1 if there is idle hardware while executing stream 0. The reason why hardware might be idle when executing stream 0 could be that some threads in the stream have finished executing the kernel and are waiting for the other threads to "catch-up," or the number of threads launched in stream 0 did

not saturate the hardware. Streams can be used to *batch* small, independent operations so as to fully utilize the hardware. This adds an additional tool for the scheduler to hide latency, which can help in *maximizing instruction throughput* and *increasing parallel efficiency*.

Batching is a method of bundling together operations to create one larger operation from two or more smaller operations. For example, if one has five matrices, each of size 5×5, finding the inverse of each matrix is an $\mathcal{O}(N^3)$ problem. This is in the microsecond timescale and will not provide enough work for a GPU to hide the start-up costs of invoking several smaller kernels and fetching data from memory. Batching all the independent, small matrix inverse operations into one larger matrix inverse maximizes GPU utilization so as to yield better performance.

The *CUDA Programming Guide* [1] and *Best Practices Guide* [7] are excellent resources for discovering programming and optimization conventions that will improve the performance of a GPU application. The extent by which performance is improved when applying any of the above techniques or concepts is application-dependent. It is worth measuring the performance of a GPU application/kernel in order to deduce the limiting factor, that is, bandwidth-bound or compute-bound. An optimization used to improve memory throughput, for example, moving data to on-chip shared memory, may not service a compute-bound application and can result in much programming effort for little performance reward. Tools such as the CUDA Profiler [8] can assist in identifying the bottlenecks in a code and should be used alongside metrics such as theoretical bandwidth and speedup to assess the efficiency of a GPU implementation.

2.5 Software Libraries for GPUs

As general-purpose GPU technology matures, an increasing number of GPU-accelerated software libraries and functions are becoming available to the programmer. These allow an application to benefit from GPU acceleration by calling functions in the libraries that have already been optimized for the target architecture, reducing the effort for the programmer. Scientific codes that already make use of libraries such as BLAS [9–11] and LAPACK [12] for linear algebra can easily call a GPU-accelerated version of a routine by linking to the accelerated library and modifying the arguments as required. There are dozens of GPU-accelerated libraries available at `https://developer.NVIDIA.com/gpu-accelerated-libraries`. Some of the more useful libraries for computational chemistry applications are the following:

- cuBLAS [13] – a completely accelerated, freely available version of the Basic Linear Algebra Subroutines (BLAS) library offering very high performance on Nvidia GPUs. The Nvidia CUDA BLAS library (cuBLAS) supports all data types, that is, single precision, double precision, complex, and double complex, concurrent streams, multiple GPUs, and batching of some operations such as matrix–matrix multiplications. cuBLAS offers *batch mode*, which enables a programmer to specify that the solutions to multiple small independent matrix operations be provided simultaneously.
- cuFFT [14] – Fast Fourier transforms are frequently used in many scientific applications and are a means of converting a function from real (time) to reciprocal space (frequency) and vice versa. The Nvidia CUDA Fast Fourier Transform library (cuFFT) offers freely available, GPU-accelerated FFT functions capable of performing up to 10× faster through the use of a divide-and-conquer decomposition strategy.
- cuSPARSE [15] – The Nvidia CUDA Sparse Matrix library (cuSPARSE) offers freely available, GPU-accelerated, sparse matrix, basic linear algebra subroutines capable of improving performance significantly as compared to popular CPU based implementations.
- CULA [16] – CULA is a CUDA-enabled GPU-accelerated linear algebra software library that is distributed as CULA Basic, CULA Premium, or CULA Commercial. Each distribution offers

different levels of functionality, features, and support at varying prices starting at free for CULA Basic (limited to single-precision data types). The programmer can choose between using a lower level version of a specific function, offering more control to the programmer at the price of higher programming effort, or a higher level version, whereby the function call takes care of all the GPU requirements behind the scenes.
- MAGMA [17] – an open-source software library from the developers of LAPACK and ScaLAPACK offering GPU-accelerated linear algebra routines for heterogeneous systems. It is capable of multiple-precision arithmetic and supports hybrid many-/multi-core CPUs and multiple Intel Xeon Phi accelerator cards or GPUs. This library interfaces to current LAPACK and BLAS packages and standards for ease of programming.

2.6 Special Features of CUDA-Enabled GPUs

GPU technology is still an emerging field in the world of HPC with a steep growth curve, the benefit of which is fast-paced innovation and improvement of features. Some of the more recent GPU features available at the time of writing are Hyper-Q technology, Multi-Process Service (MPS), Unified Memory, and NVLink.

2.6.1 Hyper-Q

Hyper-Q technology allows kernels to be launched on a single GPU from multiple CPU processes at any one time by increasing the number of "connections" to a GPU from 1 (compute capability 2.x and lower) to 32 (compute capability 3.x and above). This feature is designed to enhance the performance of CUDA streams and enable the use of multiple MPI processes. Programs that use streams can often be limited by something called *false sharing*. False sharing is where the scheduler underestimates the number of independent instructions able to be executed concurrently as a consequence of multiple independent streams being queued into a single pipeline. The multiple "connections" of Hyper-Q aid in reducing the false sharing behavior by allowing streams to remain in separate pipelines so that the scheduler is more likely to recognize independent instructions for concurrent execution.

2.6.2 MPS

MPS is a client–server runtime implementation of the CUDA API, which enables multiple MPI processes to execute kernels and carry out data transfers on a single GPU through the use of Hyper-Q technology. The server

- Enables multiple MPI processes to launch kernels on a GPU, reducing idle hardware.
- Reduces storage and scheduling resource usage by allocating a single copy of data on a GPU accessible to all MPI processes.
- Extends the lifetime of scheduling resources of a single MPI process to eliminate the overhead of swapping resources every time a new MPI process is scheduled.

2.6.3 Unified Memory

Unified memory is a feature of CUDA 6, which eliminates the need to explicitly allocate data on a GPU and transfer it between the CPU and GPU. A single pointer can be used by the CPU and the GPU to address shared data, as the hardware carries out all the data migration behind the scenes, making GPU programming much simpler, as separate GPU data arrays do not need to be created. This is done through a pool of managed memory shared by a CPU and GPU (Figure 2.7).

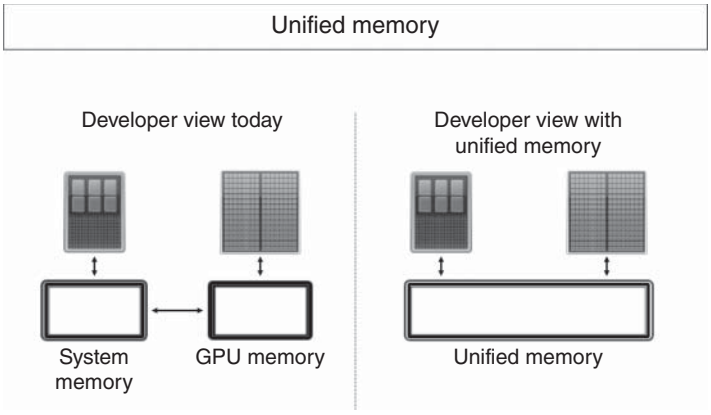


Figure 2.7 *Visual representation of unified memory. Host and device memory is presented as a single address space. This makes programming easier, but since memory copies still happen behind the scenes, performance can be lower compared to explicitly managed memory. Picture adaptation courtesy of Nvidia Corporation*

Unified memory removes some of the complexity from programming and can be as quick as local data accesses on a GPU. It is worth noting, however, that the use of asynchronous memory copies and programming at a lower level is more than likely faster than using unified memory. Unified memory is designed to open up GPU programming to a wider audience by simplifying code development.

2.6.4 NVLink

The final special feature to be discussed is NVLink, which is the first interconnect designed specifically for GPUs. At the time of writing, NVLink was not yet available, having been announced only in March 2014. NVLink promises a high-speed interconnect between CPU and GPU, allowing 5–12 times the bandwidth capabilities of the fastest interconnect currently on the market, and enables GPU–GPU connection, making data sharing between multiple GPUs faster and more efficient. One of the largest bottlenecks in a GPU application is the rate at which data can be transferred between the host and the device. This advancement in GPU technology will hopefully reduce the data transfer bottleneck and thus improve data throughput.

References

1. NVIDIA (2014) *CUDA C Programming Guide 2007–2014*. Available from: `http://docs.nvidia.com/cuda/cuda-c-programming-guide` (accessed 01 September 2015).
2. NVIDIA (2014) *CUDA Runtime API: API Reference Manual 2014*. Available from: `http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf` (accessed 01 September 2015).
3. Khronos (2014) *The OpenCL Specification 2014*. Available from: `https://http://www.khronos.org/registry/cl/` (accessed 01 September 2015).
4. OpenACC (2014) *The OpenACCTM Application Programming Interface 2013*. Available from: `http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf` (accessed 01 September 2015).

5. NVIDIA (2014) *CUDA Occupancy Calculator Spreadsheet*. Available from: `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls` (accessed 01 September 2015).
6. Rennich, S. (2014) *CUDA C/C++ Streams and Concurrency: NVIDIA; 2011*. Available from: http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf (accessed 01 September 2015).
7. NVIDIA (2014) *CUDA: Best Practices Guide 2007–2014*. Available from: `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html` (accessed 01 September 2015).
8. NVIDIA (2014) *CUDA: Profiler User's Guide 2014*. Available from: `http://docs.nvidia.com/cuda/profiler-users-guide/index.html` (accessed 01 September 2015).
9. Dongarra, J. (2002) Basic linear algebra subprograms technical forum standard. *International Journal of High Performance Applications and Supercomputing*, **16** (1), 1–111.
10. Dongarra, J. (2002) Basic linear algebra subprograms technical forum standard. *International Journal of High Performance Applications and Supercomputing*, **16** (2), 115–199.
11. Blackford, L.S., Daniel, J., Dongarra, J. *et al.* (2002) An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, **28** (2), 135–151.
12. Anderson, E., Bai, Z., Bischof, C. *et al.* (1999) *LAPACK User's Guide*, 3rd edn, Society for Industrial and Applied Mathematics, Philadelphia, PA.
13. NVIDIA (2014) *CUBLAS Library User Guide 2014*. Available from: `http://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf` (accessed 01 September 2015).
14. NVIDIA (2014) *CUFFT User Guide 2014*. Available from: `http://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf` (accessed 01 September 2015).
15. NVIDIA (2014) *CUSPARSE Library 2014*. Available from: `http://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf` (accessed 01 September 2015).
16. Humphrey, J.R., Price, D.K., Spagnoli, K.E., Paolini, A.L., Kelmelis, E.J. 2010 CULA: Hybrid GPU Accelerated Linear Algebra Routines. SPIE Defense and Security Symposium (DSS).
17. Lab IC (2014) *Software Distribution of MAGMA Version 1.4 2013*. Available from: `http://icl.cs.utk.edu/magma/` (accessed 01 September 2015).

3

Overview of Electronic Structure Methods

Andreas W. Götz
San Diego Supercomputer Center, UCSD, La Jolla, CA, USA

This chapter provides a concise overview of electronic structure methods that are widely used for applications in chemistry, physics, biology, and materials science. The discussion includes Hartree–Fock and density functional theory, semiempirical methods, and wave-function-based electron correlation methods. We present the essential theoretical background of each method, discuss common choices of basis sets that are used to discretize the equations, and point out strengths and weaknesses of the different approaches. The computational effort and its scaling with problem size strongly depends on the electronic structure method, and we highlight the computational bottlenecks with special emphasis on implementations on graphics processing units in later sections of this book.

3.1 Introduction

It is hard to overstate the importance of electronic structure theory for all branches of science that are concerned with matter at the nanometer scale since it is the interaction between the electrons that determines the properties of the matter that surrounds us, from isolated molecules to materials. As a consequence, electronic structure calculations are widely used in chemistry, physics, biology, and materials science for the analysis and prediction of the structure and thermodynamical properties of molecules and solids as well as their interaction with electromagnetic fields and radiation [1–5]. Because of the significant advances in theoretical methods, numerical approximations, as well as computer hardware, electronic structure methods are now routinely employed both in basic science and in applied research in numerous industries. The value of electronic structure methods was recognized in 1998 with award of the Nobel Prize in chemistry to Kohn "for his development of the

Electronic Structure Calculations on Graphics Processing Units:
From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

density-functional theory" [6] and John A. Pople "for his development of computational methods in quantum chemistry" [7].

At the basis of electronic structure methods lies the Born–Oppenheimer approximation, which separates the motion of electrons and atomic nuclei. The task is then reduced to describing the motions of the electrons for a given configuration of nuclei, that is, to solve the electronic Schrödinger equation and find the electronic wave function for the system under consideration. If relativistic effects are important, then the Dirac equation needs to be solved in place of the Schrödinger equation. However, in this book we will not discuss the Dirac equation or other methods and approximations to take relativistic effects into account [8]. This chapter gives an overview of the most common theoretical approaches to solve the electronic Schrödinger equation. Later chapters of this book explain in detail how graphics processing units (GPUs) are used in practical implementations of these methods.

3.1.1 Computational Complexity

For many-electron systems, analytical solutions to the Schrödinger equation are not known, and one has to resort to numerical approximations. The same complexity that precludes the exact analytical solution of the Schrödinger equation, however, also results in highly unfavorable scaling of computational effort and the required resources such as random access memory or disk storage. For example, the computational demand of exact calculations with the full configuration interaction (CI) method grows exponentially with the size of the system. Approximations that represent a reasonable compromise between efficiency and accuracy are thus needed to carry out electronic structure calculations on sizable molecules. This chapter presents the theoretical background of the most important approximations. Additional details and references can be found in standard quantum chemistry and electronic structure theory text books [9–11].

A hierarchy of well-controlled approximations that do not sacrifice the predictive power of the parameter-free nature of quantum mechanical calculations but exhibit polynomial rather than exponential scaling is routinely available [10]. Among these are popular wave function-based quantum chemical methods such as Møller–Plesset perturbation theory [12] and coupled cluster theory [13, 14], both of which are extensively discussed in terms of massively parallel implementations on GPUs in later chapters of this book. These so-called *ab initio* methods (*ab initio* because they do not rely on parameterizations) exhibit a scaling of computational complexity of at least $\mathcal{O}(N^5)$, where N denotes system size, meaning that applications to many interesting problems are still out of reach.

Partly owing to this steep scaling of *ab initio* wave function-based methods, density functional theory (DFT) [15–17] is established as the most popular electronic structure theory, which, in its Kohn–Sham formulation, replaces the complicated many-electron Schrödinger equation with a set of self-consistent one-electron equations with a formal scaling of computational effort of $\mathcal{O}(N^3)$, or $\mathcal{O}(N^4)$ if exact exchange is included. Many different implementations of density functional methods exist, usually focusing on either finite molecular systems or condensed phase systems and solids. Several examples for these cases that exploit GPUs are discussed in later chapters.

Finally, semiempirical quantum chemistry methods [18] and density functional tight binding (DFTB) [19, 20] introduce additional approximations and thus have to rely on extensive parameterizations. As a consequence, they are computationally extremely efficient and exhibit low quadratic scaling for computing the Hamiltonian.

Much effort has been devoted to improving the parameterizations, and thus fidelity of these approximate electronic structure methods, as well as to reduce both the prefactor and the effective scaling of *ab initio* and density functional methods. Because of the "near-sightedness" of matter [21, 22], in principle, linear scaling with system size is achievable, at least for insulating materials with a sufficiently large bandgap or gap between highest occupied and lowest unoccupied molecular orbitals (MOs). In combination with the advancements in computer hardware that we have witnessed

over the last decades, not least the development of massively parallel GPU processors, the future is thus bright for electronic structure theory and its application to an ever-increasing range of fields.

3.1.2 Application Fields, from Structures to Spectroscopy

The electronic wave function depends parametrically on the position of the nuclei, and in the Born–Oppenheimer approximation the nuclei move on the potential energy surfaces, which are the solutions of the electronic Schrödinger equation. Thus, it is possible to use electronic structure methods to find equilibrium molecular structures, which are stationary points on this surface. Hence, one of the most common applications is to map reaction mechanisms that connect these stationary points through thermal or photochemical processes [11]. Since both the electronic ground state and electronically excited states are accessible within most of the aforementioned electronic structure methods, it is possible to model many important processes that involve transitions between these electronic states. Application fields include spectroscopy and processes that involve charge transfer and separation such as biological light harvesting or artificial photosynthesis and solar cells [4, 5].

Among the many spectra that are nowadays routinely accessible are vibrational infrared (IR) and Raman spectra, nuclear magnetic resonance (NMR) spectra including chemical shifts and J coupling constants, electronic paramagnetic resonance (EPR) spectra, and electronic spectra including absorption and emission spectra, to name a few [4, 5, 11, 23, 24]. Importantly, electronic structure calculations make it not only possible to predict properties of molecules and materials but also enable the interpretation and understanding in terms of structural features of molecules and materials that lead to desired properties. This enables a rational design of molecules and materials with specific properties, such as improved homogeneous or heterogeneous catalysts, enzymes for biocatalysis, drugs, semiconductors, fuel cells, or donor–acceptor materials for light harvesting in solar cells.

3.1.3 Chapter Overview

This chapter gives a concise overview of the methods that are most widely used for electronic structure calculations, with a bias toward approaches that are common in molecular quantum chemistry as opposed to computational solid-state physics. It covers the background and relevant details for all approaches that are included in the following chapters of the book but omits other important approaches such as quantum Monte Carlo (QMC) [25–27], explicitly correlated R12/F12 Methods [28], or many-body Green's function methods for electronic excitations [29], for which there are no GPU ports at the time of writing or which could not be covered in the remainder of the book.

Our presentation starts with Hartree–Fock (HF) and DFT methods. Although these methods are conceptually different, the working equations are essentially identical, and most practical implementations use in fact the same code structure for both. DFT is undoubtedly the work horse of electronic structure calculations both in chemistry and materials science due to its favorable combination of low computational cost and good accuracy. While HF solutions to the electronic structure problem in general do not lead to quantitatively useful results, HF theory is important as the starting point for accurate wave function-based electron correlation methods, which we cover later in the chapter. We also summarize common ways of numerically representing the electron density and wave function, using either analytical basis functions or discrete numerical representations. The choice of the basis naturally has a significant effect on the required numerical methods and approaches that are used to achieve good computational scaling and, in turn, the algorithms that can be used for efficient implementations on GPUs. We also cover semiempirical approaches, which, as mentioned previously, make approximations that render these methods computationally efficient compared to HF or DFT. We use atomic units throughout this chapter.

3.2 Hartree–Fock Theory

As mentioned, electronic structure methods aim at solving the Schrödinger equation for many-electron systems within the Born–Oppenheimer approximation:

$$\hat{H}\Psi = E\Psi, \tag{3.1}$$

where $\hat{H}$ is the nonrelativistic, time-independent electronic Hamiltonian, and Ψ the corresponding wave function. The Hamiltonian includes the kinetic energy operator and describes the interaction among all elementary particles (electrons and nuclei) and external electromagnetic fields. If the Hamiltonian contains fields that are time-dependent, then the time-dependent Schrödinger equation has to be solved.

HF theory [9–11] is based on the assumption that the electronic wave function can, to a good approximation, be described in terms of a single Slater determinant Φ, that is, an antisymmetrized product of orthogonal molecular spin orbitals or single-particle wave functions ψ_i. Importantly, this wave function ansatz satisfies the Pauli exclusion principle. For brevity of discussion, we will assume the so-called restricted (closed shell) Hartree–Fock (RHF) theory, in which each orbital ψ_i is doubly occupied with a pair of spin-up and spin-down electrons that share the same probability density in real space. Extension to open-shell systems with an odd number of electrons or otherwise unpaired electrons and nonzero spin density is straightforward within either the restricted open-shell Hartree–Fock (ROHF) or unrestricted Hartree–Fock (UHF) formalism.

Assuming a Slater determinant as wave function, the expectation value of the energy becomes [9–11]

$$E_{\mathrm{RHF}} = 2\sum_{i}^{\mathrm{occ}}\langle i|\hat{h}|i\rangle + 2\sum_{ij}^{\mathrm{occ}}(ii|jj) - \sum_{ij}^{\mathrm{occ}}(ij|ij), \tag{3.2}$$

where we have introduced the one-electron operator

$$\hat{h} = -\frac{1}{2}\nabla^2 + v_{\mathrm{ext}}(\mathbf{r}) \tag{3.3}$$

that contains the kinetic energy operator $-1/2\nabla^2$ and the external potential v_{ext} due to Coulomb interaction of the electrons with the nuclei and other external fields. We use following shorthand notation for the one-electron integrals:

$$\langle i|\hat{h}|i\rangle = \int d\mathbf{r}\ \psi_i(\mathbf{r})\hat{h}\psi_i(\mathbf{r}), \tag{3.4}$$

and define the shorthand notation for the two-electron repulsion integrals (ERIs) via

$$(ij|kl) = \int d\mathbf{r}d\mathbf{r}'\,\frac{\psi_i(\mathbf{r})\psi_j(\mathbf{r})\psi_k(\mathbf{r}')\psi_l(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}. \tag{3.5}$$

The summations in the equations above run over all $n/2$ doubly occupied orbitals of an n-electron system. The first term on the right-hand side of Eq. (3.2) contains the kinetic energy of the electrons and the interaction energy between the electrons and the nuclei and other external electric fields. The second term is the classical Coulomb repulsion among all electrons, while the third term is the exchange energy, which is due to the antisymmetry of the wave function.

Minimizing the HF energy (3.2) with respect to variations in the orbitals under the constraint that the orbitals remain orthonormal leads to the single-particle Hartree–Fock equations, which replace the complicated many-body Schrödinger equation:

$$\hat{F}\psi_i = \epsilon_i\psi_i. \tag{3.6}$$

$\hat{F}$ is the closed-shell Fock operator,

$$\hat{F} = \hat{h} + \sum_{j}^{\text{occ}} \left(2\hat{J}_j - \hat{K}_j\right), \tag{3.7}$$

where we have defined the Coulomb and exchange operators

$$\hat{J}_j\psi_i(\mathbf{r}) = \int d\mathbf{r}'\psi_j(\mathbf{r}')\psi_j(\mathbf{r}')\psi_i(\mathbf{r}), \tag{3.8}$$

$$\hat{K}_j\psi_i(\mathbf{r}) = \int d\mathbf{r}'\psi_j(\mathbf{r}')\psi_i(\mathbf{r}')\psi_j(\mathbf{r}). \tag{3.9}$$

The eigenvalues ϵ_i of the HF equations (3.6) are Lagrange multipliers that guarantee orthonormality of the orbitals ψ_i and can be interpreted as molecular orbital energies. The $n/2$ orbitals with the lowest eigenvalues are the orbitals that are occupied with n electrons and thus represent the ground state wave function, while the unoccupied or virtual orbitals spanning the remainder of the Hilbert space do not have a direct meaning but are of relevance for correlated wave function-based electronic structure methods and methods that describe excited electronic states.

Hartree–Fock theory is a mean-field theory since the solutions to the HF equations, the single-particle orbitals ψ_i, describe the motion of a single electron (or pair of electrons in the case of RHF theory) in the field of all other electrons. The interaction with all other electrons is encoded in the Coulomb and exchange operators, both of which, and thus also the Fock operator, depend on the orbitals ψ_i that are the solutions of the HF equations. As a consequence, the HF equations have to be solved iteratively until self-consistency is reached, that is, until the orbitals that define the Fock operator are identical to the solutions of the Fock equations. For this reason, the HF equations are also often referred to as self-consistent field (SCF) equations.

3.2.1 Basis Set Representation

For practical applications, a numerical representation of the orbitals ψ_i is required, and a variety of choices is possible and in use, as discussed in greater detail in the following and throughout some of the later chapters of this book. In brief, plane waves (see Chapters 7 and 9) are commonly employed for calculations under periodic boundary conditions, but other choices are possible, including wavelets (see Chapter 6) and real-space discretizations on a numerical grid (see Chapter 9). While numerical grids can also be used for finite systems (see Chapter 10), it is most common for finite molecular systems to expand the orbitals using localized, atom-centered analytical basis functions that are Gaussian functions (see Chapter 4) or, less frequently, Slater type functions (see Chapter 5). In a basis set representation the molecular orbitals are expanded as

$$\psi_i(\mathbf{r}) = \sum_{\mu} c_{\mu i}\phi_\mu(\mathbf{r}), \tag{3.10}$$

where $c_{\mu i}$ are the molecular orbital expansion coefficients and $\{\phi_\mu\}$ is the basis set. Within this representation, the HF equations turn into a set of nonlinear, general eigenvalue equations, which can be written in matrix form as

$$\mathbf{FC} = \mathbf{SC}\epsilon, \tag{3.11}$$

where $\mathbf{F}$ is the matrix representation of the Fock operator in the expansion basis, $\mathbf{C}$ collects the expansion coefficients for all orbitals in its columns, ϵ is the diagonal matrix of orbital eigenvalues, and the overlap matrix $\mathbf{S}$ is the metric of the basis set that arises in the general case of non-orthogonal basis functions and is unity otherwise. These equations are now amenable for efficient implementation in computer programs.

In line with the definition of the Fock operator $\hat{F}$ in Eq. (3.7), the Fock matrix elements are given as a sum of one-electron contributions (kinetic energy and external potential) and two-electron Coulomb and exchange contributions:

$$\mathbf{F} = \mathbf{h} + \mathbf{J} - \frac{1}{2}\mathbf{K}, \tag{3.12}$$

$$h_{\mu\nu} = \left\langle \mu \left| -\frac{1}{2}\nabla^2 + v_{\text{ext}} \right| \nu \right\rangle, \tag{3.13}$$

$$J_{\mu\nu} = \sum_{\kappa\lambda} P_{\kappa\lambda} (\mu\nu|\kappa\lambda), \tag{3.14}$$

$$K_{\mu\nu} = \sum_{\kappa\lambda} P_{\kappa\lambda} (\mu\kappa|\nu\lambda), \tag{3.15}$$

where

$$P_{\mu\nu} = 2 \sum_{i}^{\text{occ}} c_{\mu i} c_{\nu i} \tag{3.16}$$

are elements of the density matrix that can be used to express the electron density in terms of the basis functions

$$\rho(\mathbf{r}) = \sum_{\mu\nu} P_{\mu\nu} \phi_\mu(\mathbf{r}) \phi_\nu(\mathbf{r}). \tag{3.17}$$

The notation for the one-electron integrals and ERIs has been defined in Eqs. (3.4) and (3.5), but now the integrals are over basis functions and not MOs.

There are two major computational bottlenecks in HF calculations. These are the evaluation of the Fock matrix elements as defined in Eqs. (3.12)–(3.15), and solution of the SCF equations from Eq. (3.11). The latter requires diagonalization of the Fock matrix, which scales cubically with system size and eventually dominates the computational cost for very large calculations. In this case, alternative ways to solve the SCF equations that do not rely on matrix diagonalization must be employed [30, 31].

3.2.2 Two-Electron Repulsion Integrals

The computational cost for calculating the Fock matrix elements is dominated by the evaluation of the ERIs $(\mu\nu|\kappa\lambda)$, which are required for the Coulomb and exchange contributions, see Eqs. (3.14) and (3.15). The ERIs are four-index quantities

$$(\mu\nu|\kappa\lambda) = \int d\mathbf{r} d\mathbf{r}' \frac{\phi_\mu(\mathbf{r}) \phi_\nu(\mathbf{r}) \phi_\kappa(\mathbf{r}') \phi_\lambda(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} \tag{3.18}$$

and formally there are $\mathcal{O}(N^4)$ ERIs that need to be evaluated during an HF calculation. Because of the sheer number of ERIs, storage in memory is not an option. Instead, so-called direct SCF methods recompute the ERIs during each iteration of the SCF cycle and contract them with the density matrix elements to directly obtain the corresponding Coulomb and exchange contributions to the Fock matrix [9–11].

For large, spatially extended systems, most of the ERIs are zero or negligible, which can be efficiently exploited to reduce the number of ERIs that have to be computed. This leads to an effective scaling of $\mathcal{O}(N^2)$ for most practical calculations. The reason for this asymptotic scaling with local, atom-centered basis sets consisting of Gaussian-type functions (GTFs), Slater-type functions (STFs), or numerical atomic orbitals is easy to understand. These basis functions decay to zero at large distances from their origin. As a consequence, an overlap density $\rho_{\mu\nu} = \phi_\mu \phi_\nu$, that is, the product of two

basis functions, will always be negligible for pairs of basis functions whose origins are sufficiently far away from each other. Any ERI with a negligibly small overlap density can then safely be discarded from the calculation. This is typically exploited quantitatively by using the Schwarz inequality

$$(\mu\nu|\kappa\lambda) \leq (\mu\nu|\mu\nu)^{1/2}(\kappa\lambda|\kappa\lambda)^{1/2}, \tag{3.19}$$

which provides an upper bound for the magnitude of ERIs and has proven successful and reliable for integral prescreening in electronic structure calculations. The ERIs required for prescreening using the Schwarz inequality are two-index quantities and thus not numerous and can be conveniently precomputed before solving the SCF equations. As can be seen from Eqs. (3.14) and (3.15), the Fock matrix contribution of an ERI will also be negligible if the corresponding density matrix elements are sufficiently small. Direct SCF implementations exploit thus the sparsity of the density matrix in combination with Schwarz integral screening.

While each ERI in principle contributes to both the Coulomb and exchange contributions to the Fock matrix, it is in fact beneficial to separate the calculation of the Coulomb matrix elements $J_{\mu\nu}$ and the exchange matrix elements $K_{\mu\nu}$ and to exploit the different nature of the Coulomb and exchange interactions. This leads to linear scaling methods [22, 32], that is, implementations that show effective $\mathcal{O}(N)$ scaling in the asymptotic limit of large systems. Important progress with respect to the Coulomb problem has been achieved, for example, by generalization of the fast multipole method to Gaussian charge distributions (overlap densities as defined above), which was shown to lead to linear scaling [33, 34]. For nonmetallic systems with a large gap between highest occupied and lowest unoccupied molecular orbitals, the density matrix decays exponentially. This has led to the development of $\mathcal{O}(N)$ methods that exploit the fast decaying nature of the exchange interaction [35, 36]. Using these algorithmic advances for linear scaling evaluation of the Fock matrix, SCF calculations with hundreds to thousands of atoms have become possible.

The importance of the algorithmic advances that enable low-scaling SCF calculations cannot be understated. Nevertheless, many systems of interest simply do not fall into the linear scaling regime. For this reason, and in order to reduce the prefactor of linear scaling computations of the Fock matrix, efficient algorithms and software implementations to compute the ERIs will always be required. Tremendous effort has been made over the last decades to develop highly efficient CPU-based algorithms that require a minimum of floating-point operations for the calculation of ERIs with Gaussian basis sets. In Chapter 4 of this book, Luehr, Sisto, and Martínez demonstrate how GPU architectures can, instead, be efficiently exploited to compute Gaussian basis set ERIs in a massively parallel manner.

3.2.3 Diagonalization

As mentioned previously, the matrix eigenvalue equation (3.11) is usually solved by diagonalization of the Fock matrix. For small matrices, this is an efficient approach since in this case (with exception of semiempirical methods) the numerical work to compute the Fock matrix is significantly larger than its diagonalization. However, algorithms for the diagonalization of dense matrices scale as $\mathcal{O}(N^3)$ and thus invariably will start to dominate the computational effort of an SCF calculation if efficient prescreening techniques and linear scaling algorithms are employed for the calculation of the Fock matrix. In addition, it is almost trivial to make use of parallel processing when computing contributions to the Fock matrix, while this is much more difficult for matrix diagonalization. As a result, methods have been developed to replace the diagonalization step by procedures with a more favorable scaling behavior for the update of the orbitals, for example, based on direct density matrix optimizations with conjugate gradient algorithms [30, 31]. In Chapter 8, Schütt *et al*. demonstrate how GPUs can be exploited in algorithms for orbital updates that achieve linear scaling computational effort for large systems using sparse matrix algebra.

3.3 Density Functional Theory

The roots of DFT [15, 16, 37–40] can be traced back to work by Thomas and Fermi in the late 1920s who used models for the electronic structure of atoms that depend only on the electron density [41, 42]. Similarly, the Hartree–Fock–Slater or Xα method [43] replaces the nonlocal, orbital-dependent exchange term of the Hartree–Fock method by the approximate local exchange potential of Dirac [44], which is very simple and given by $\rho^{1/3}$. The basis for modern DFT, however, is the Hohenberg–Kohn theorems [45], later generalized by Levy, which prove that all properties of a many-electron system are functionals of the ground-state electron density. This means that, in place of solving the complicated many-electron Schrödinger equation, the ground-state energy can be obtained by minimizing the functional of the total electronic energy with respect to variations in the electron density. Unfortunately, this energy functional is not known, and it has proven difficult to develop functionals of the electron density that are sufficiently accurate for practical applications. This holds, in particular, for the density functional of the kinetic energy, a problem that already plagued the Thomas–Fermi approach. Thus, while the Hohenberg–Kohn theorems give a sound justification for DFT, the importance of this theory would not have risen above that of the Thomas–Fermi model if it was not for the work of Kohn and Sham.

3.3.1 Kohn–Sham Theory

Kohn and Sham had the ingenious idea to obtain the real, interacting electron density from an auxiliary system of noninteracting electrons with an electron density that is identical to that of the real system. The wave function for such a system of noninteracting electrons is a Slater determinant built from the eigenfunctions ψ_i of a single-particle Hamiltonian (similar to HF theory). In Kohn–Sham (KS) theory [46], the total electronic energy is given as

$$E[\rho] = T_s[\rho] + \int d\mathbf{r}\rho(\mathbf{r})v_{\text{ext}}(\mathbf{r}) + \frac{1}{2}\int d\mathbf{r}\, d\mathbf{r}' \frac{\rho(\mathbf{r})\rho(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|} + E_{\text{xc}}[\rho], \tag{3.20}$$

where the various terms represent, in order, the kinetic energy of the KS system of noninteracting electrons, the interaction between the electrons and the external potential of nuclei and other fields, the Hartree energy arising from the Coulomb interaction of the electron density, and the remainder of the total energy, which is referred to as exchange-correlation (xc) energy E_{xc}. The electron density is given in terms of the occupied orbitals

$$\rho(\mathbf{r}) = 2\sum_{i}^{\text{occ}} |\psi_i(\mathbf{r})|^2, \tag{3.21}$$

where, as before in the section on HF theory, we have assumed a closed-shell n electron system with $n/2$ doubly occupied orbitals. The functional of the kinetic energy of noninteracting electrons is given as

$$T_s[\rho] = 2\sum_{i}^{\text{occ}} \left\langle \psi_i \left| -\frac{1}{2}\nabla^2 \right| \psi_i \right\rangle. \tag{3.22}$$

This noninteracting kinetic energy is different from the exact kinetic energy of the real system; however, it is a convenient and fairly good approximation. The remainder of the exact kinetic energy is taken into account as part of the xc energy E_{xc}.

Variational minimization of this total energy functional with respect to the electron density under the constraint that the orbitals remain orthonormal leads to the single-particle KS equations

$$\hat{F}^{\text{KS}}\psi_i = \epsilon_i\psi_i \tag{3.23}$$

with the KS single-particle Hamiltonian

$$\hat{F}^{\mathrm{KS}} = -\frac{1}{2}\nabla^2 + v_{\mathrm{ext}}(\mathbf{r}) + v_{\mathrm{H}}(\mathbf{r}) + v_{\mathrm{xc}}(\mathbf{r}). \tag{3.24}$$

Here,

$$v_{\mathrm{H}}(\mathbf{r}) = \int d\mathbf{r}' \frac{\rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} \tag{3.25}$$

and

$$v_{\mathrm{xc}}(\mathbf{r}) = \frac{\delta E_{\mathrm{xc}}[\rho]}{\delta \rho(\mathbf{r})} \tag{3.26}$$

are the Hartree potential and the xc potential, respectively. The KS Hamiltonian looks similar to the Fock operator in HF theory, but the xc potential is a local potential, whereas the exchange operator in HF theory is orbital-dependent and nonlocal. The single-particle KS equations (3.23) look deceptively simple, but the full many-body nature is reflected in the fact of our incomplete knowledge of the xc energy density functional.

In basis set representation the KS Hamiltonian becomes

$$\mathbf{F}^{\mathrm{KS}} = \mathbf{h} + \mathbf{J} + \mathbf{V}_{\mathrm{xc}}, \tag{3.27}$$

where the one-electron matrix elements $h_{\mu\nu}$ and Coulomb matrix elements $J_{\mu\nu}$ are the same as in HF theory (see Eqs. (3.13) and (3.14)), and the matrix elements of the xc potential are

$$V_{\mathrm{xc},\mu\nu} = \langle \mu | v_{\mathrm{xc}} | \nu \rangle. \tag{3.28}$$

Thus, it is straightforward to implement DFT into an HF program by replacing the HF exchange energy and operator with the xc energy and potential.

3.3.2 Exchange-Correlation Functionals

For the KS method to be successful, good approximations to the exchange-correlation energy E_{xc} are required [47]. For many years, the local density approximation (LDA) was the most widely used scheme. Within the LDA, the xc energy is expressed as

$$E_{\mathrm{xc}}^{\mathrm{LDA}}[\rho] = \int d\mathbf{r} \epsilon_{\mathrm{xc}}(\rho(\mathbf{r})), \tag{3.29}$$

where ϵ_{xc} is the xc energy density of a homogeneous electron gas, which is accurately known from QMC calculations [48]. The xc functional can be divided into exchange and correlation contributions according to

$$E_{\mathrm{xc}}^{\mathrm{LDA}}[\rho] = E_x^{\mathrm{LDA}}[\rho] + E_c^{\mathrm{LDA}}[\rho]. \tag{3.30}$$

In the case of LDA, the Dirac exchange energy functional is used for E_x. Several expressions exist for the correlation contribution, with popular versions due to Vosko, Wilk, and Nusair (VWN) [49] or Perdew and Wang (PW92) [50]. For practical purposes, all LDA functionals are nearly equivalent. The LDA approximation works well in the limit of slowly varying densities, and became popular with condensed-matter physicists who found that LDA gave good descriptions of bulk solids and surfaces, but it was never widely adopted by quantum chemists.

The situation changed with introduction of the generalized gradient approximation (GGA), which supplements the LDA functional with a term that depends on the gradient of the electron density:

$$E_{\mathrm{xc}}^{\mathrm{GGA}}[\rho] = \int d\mathbf{r} \epsilon_{\mathrm{xc}}(\rho(\mathbf{r}), \nabla\rho(\mathbf{r})). \tag{3.31}$$

The introduction of GGA corrections to the LDA made DFT popular among chemists since GGA functionals like BLYP, consisting of the exchange contribution by Becke (B88)[51] and the correlation contribution by Lee, Yang and, Parr (LYP) [52], provided sufficient accuracy for quantitative analysis of chemical bonds. Many other GGA functionals are available, and new ones continue to appear. Some of these functionals contain empirical parameters that are optimized to reproduce reference data from experiments or high-level calculations, while others are obtained from physical constraints such that they incorporate key features of the –exchange-correlation energy.

Several newer density functionals depend on additional variables beyond the electron density and its gradient. Meta-GGA functionals add dependence on the kinetic energy density τ of the KS system:

$$E_{\mathrm{xc}}^{\mathrm{MGGA}}[\rho] = \int d\mathbf{r}\epsilon_{\mathrm{xc}}(\rho(\mathbf{r}), \nabla\rho(\mathbf{r}), \tau(\mathbf{r})), \tag{3.32}$$

$$\tau(\mathbf{r}) = \sum_{i}^{\mathrm{occ}} |\nabla\psi_i(\mathbf{r})|^2. \tag{3.33}$$

Meta-GGAs are examples of functionals that are implicit density functionals, since the KS orbitals ψ_i implicitly depend on the electron density.

A rather important development was the introduction of hybrid xc functionals in 1993 by Becke [53]. In hybrid functionals, a portion of the explicitly density-dependent DFT exchange is replaced by the orbital-dependent HF exact-exchange energy:

$$E_{\mathrm{xc}}^{\mathrm{hybrid}} = aE_x^{\mathrm{exact}} + (1-a)E_x^{\mathrm{DFT}} + E_c^{\mathrm{DFT}}. \tag{3.34}$$

A mixing coefficient of $a \approx 1/4$ was found to work well in many cases, and this choice is also backed by theoretical considerations. This value is used in the nonempirical PBE0 [54] functional, which is successfully employed for applications to ground- and excited-state properties. The importance of hybrid functionals is also reflected in the fact that the B3LYP [55] hybrid functional has been the most widely used density functional in computational chemistry over the last two decades [39, 40].

Employing exact exchange leads to a partial cancellation of the self-interaction error inherent to local or semilocal density functionals and improves the description of charge transfer and Rydberg excitations in time-dependent density functional theory (TDDFT) schemes [40]. It is worth pointing out that the exact exchange part of standard hybrid functionals is usually implemented using the nonlocal, orbital-dependent HF potential and as such is not the KS exact exchange. The local KS exact exchange potential, instead, can be obtained with the so-called optimized effective potential methods, with important consequences in particular for unoccupied KS orbitals [56]. However, the occupied orbitals obtained with standard implementations of hybrid functionals are close to true KS orbitals.

More recently, range-separated hybrid functionals, also termed long-range corrected functionals, have been introduced. In these functionals, for example, CAM-B3LYP [57], the exact exchange is screened as a function of the interelectronic distance. As a consequence, long-range charge transfer excitations for which GGAs or regular hybrid functionals fail are well described.

For obvious reasons, all density functionals discussed so far are not able to correctly describe long-range dispersion that decays as $-C_6/R^6$. Instead, the asymptotic interactions decay exponentially. To correct for this deficiency, *ad hoc* empirical dispersion corrections in form of pair potentials have been introduced. These empirical corrections work rather well. Numerical results obtained, for example, with Grimme's DFT-D methodology [58] for binding energies in weakly interacting systems are excellent. Several nonempirical dispersion models that do not rely on London-type pairwise corrections have also been developed, including those by Tkatchenko and Scheffler [59], and applications of DFT dispersion methods are rapidly growing.

3.3.3 Exchange-Correlation Quadrature

In general, the dependence of the xc functionals on the electron density ρ, its gradient $\nabla\rho$, and the kinetic energy density τ is very complicated. As a consequence, it is impossible to analytically solve the xc integrals required to obtain the xc energy E_{xc} and the matrix elements of the xc potential $V_{\text{xc},\mu\nu}$. Instead, a numerical integration grid is employed to integrate the xc potential v_{xc} and the xc energy density ϵ_{xc}. This is a set of points $\mathbf{r}_i$ and nonnegative weights ω_i such that

$$\int d\mathbf{r} f(\mathbf{r}) \approx \sum_i \omega_i f(\mathbf{r}_i). \tag{3.35}$$

The xc energy is thus obtained from numerical quadrature as

$$E_{\text{xc}}^{\text{MGGA}} \approx \sum_i \omega_i \epsilon_{\text{xc}}^{\text{MGGA}}(\rho(\mathbf{r}_i), \nabla\rho(\mathbf{r}_i), \tau(\mathbf{r}_i)). \tag{3.36}$$

The integrand is usually partitioned over atomic points using a weight scheme with further decomposition into radial and angular components of each atomic contribution [60, 61]. The formal scaling of the computational cost for the setup of an integration grid with such a weight scheme is $\mathcal{O}(N^3)$. The numerical quadrature itself scales as $\mathcal{O}(N^3)$ because the number of atomic-based grid points grows linearly with system size N and, as can be seen from Eq. (3.17), contributions need to be evaluated for each pair of basis functions. In practice, linear scaling quadrature and construction of the integration grid are possible using appropriate screening techniques that take the sparseness of the density matrix and local nature of basis functions into account [62]. Nevertheless, numerical quadrature constitutes a major computational bottleneck. Algorithms for efficient numerical xc quadrature on GPUs are discussed in Chapters 4 and 5.

If Slater-type basis functions are employed, three- and four-center ERIs of Eq. (3.18) cannot be computed analytically. In this case, the electron density is expanded in an auxiliary basis, which is sometimes referred to as density fitting. As explained in Chapter 5, the Hartree energy and matrix elements of the Hartree potential are then computed by numerical quadrature of the Hartree potential [63] within this auxiliary basis. HF exchange integrals with STFs can also be evaluated from a combination of density fitting and numerical quadrature on the xc integration grid [64].

3.4 Basis Sets

The choice of basis $\{\phi_\mu\}$ to expand the orbitals ψ_i and electron density ρ is of paramount importance, because this determines both the numerical accuracy and computational cost of electronic structure methods. Atom-centered local basis functions are most popular for nonperiodic systems such as molecules or metal clusters, while plane-wave basis sets are mostly used for electronic structure calculations of periodic systems including metals, silicates, zeolithes, and molecular crystals. Alternatively, it is possible to use real-space representations on a discrete grid both for molecular and periodic systems.

3.4.1 Slater-Type Functions

Atom-centered Slater-type functions (STFs) have the functional form

$$\phi_{\zeta nlm}^{\text{STF}}(\mathbf{r}) = N r^{n-1} e^{-\zeta r} Y_l^m(\theta, \varphi), \tag{3.37}$$

where n is the principal quantum number, Y_l^m is a spherical harmonic of the angular momentum quantum number l and magnetic quantum number m, and N is a normalization constant. The radial

part is an exponential function, which makes STFs the natural solutions to the atomic Schrödinger equation for a single electron, displaying both a cusp at the nucleus and, more importantly, the correct asymptotic decay. The exponential dependence also guarantees a fairly rapid convergence of orbitals in molecular simulations with increasing number of basis functions. This is in contrast to GTFs, which require a larger set of basis functions to represent the orbitals and the electron density with the same quality.

Basis sets that contain the minimum number of functions required to describe all electrons of neutral atoms are called minimal basis sets or single zeta basis sets. Such basis sets are useful at best for qualitative purposes. To increase the flexibility of the basis set and achieve a good representation of molecular orbitals, the number of basis functions needs to be multiplied, leading to double-zeta, triple-zeta, quadruple-zeta basis sets, and so on. In addition, polarization functions, that is, additional basis functions with higher angular momentum quantum numbers, are required to appropriately describe the deformation of the atomic electron densities upon bond formation and due to electron–electron interactions. Specially tailored basis sets may be required to compute certain molecular properties. For instance, computation of NMR J coupling constants requires a highly accurate representation of the electron density in vicinity of the nuclei. High-quality Slater basis sets for DFT calculations are available [65].

A major disadvantage of STFs is that no analytical solutions to the three- and four-center ERIs of Eq. (3.18) are known. They are thus primarily used for atomic and diatomic systems when high accuracy is desired. STFs are also used in semiempirical methods where all three- and four-center integrals are neglected. DFT methods that are based on STFs circumvent the need to explicitly compute these ERIs using density-fitting approximations as outlined in the section on auxiliary basis sets and explained in more detail in Chapter 5.

3.4.2 Gaussian-Type Functions

Atom-centered Gaussian-type functions (GTFs) are the most widely used basis functions for quantum chemistry applications [66]. Their functional form is given as

$$\phi^{\mathrm{GTF}}_{\zeta nlm}(\mathbf{r}) = N r^{(2n-2-l)} e^{-\zeta r^2} Y^m_l(\theta, \varphi) \tag{3.38}$$

or, in the case of Cartesian GTFs, as

$$\phi^{\mathrm{GTF}}_{\zeta l_x l_y l_z}(\mathbf{r}) = N x^{l_x} y^{l_y} z^{l_z} e^{-\zeta r^2}. \tag{3.39}$$

Because of the dependence on r^2 in the exponential of the radial part, Gaussian basis sets do not display a cusp at the nucleus and decay too rapidly far from the nucleus, thus poorly representing the tail of the wave function. This is usually compensated for by using a large number of basis functions with a range of different exponents ζ. In order to keep the computational cost lower, several GTFs are frequently contracted according to

$$\phi^{\mathrm{CGTF}}_{nlm}(\mathbf{r}) = \sum_i a_i \phi^{\mathrm{GTF}}_{\zeta_i nlm}(\mathbf{r}) \tag{3.40}$$

into a single basis function with fixed coefficents a_i, which better represents the radial solutions of the atomic Schrödinger equation. A very large number of high-quality basis sets is available for a wide range of applications with DFT and *ab initio* wave function methods.

The popularity of Gaussian basis sets is due to the Gaussian product Theorem [67] according to which the product of two GTFs yields a different GTF with origin between the original GTFs. If we define

$$G_{\alpha A}(\mathbf{r}) = e^{-\alpha |\mathbf{r}-\mathbf{R}_A|} \tag{3.41}$$

for a spherical Gaussian with origin at $\mathbf{R}_A$, then

$$G_{\alpha A}(\mathbf{r})G_{\beta B}(\mathbf{r}) = K_{AB}G_{\gamma C}(\mathbf{r}) \tag{3.42}$$

with

$$\begin{aligned} \gamma &= \alpha + \beta, \\ \mathbf{R}_C &= (\alpha\mathbf{R}_A + \beta\mathbf{R}_B)\gamma^{-1}, \\ K_{AB} &= e^{-\alpha\beta\gamma^{-1}|\mathbf{R}_A - \mathbf{R}_B|^2}. \end{aligned} \tag{3.43}$$

As a consequence, at most two-center ERIs have to be evaluated since using the Gaussian product theorem any three- or four-center ERIs can be expressed in terms of a two-center ERI. Many different algorithms have been developed to efficiently calculate integrals using Gaussian basis sets, and Chapter 4 deals with algorithms and implementations for GPUs.

3.4.3 Plane Waves

For electronic structure calculations of periodic systems, the most widely used approach is to employ a plane wave basis, that is, to represent the orbitals, the electron density, and other quantities in terms of their Fourier expansions. According to Bloch's theorem, the KS orbitals of a system with periodic external potential can be written as a product of a cell-periodic part and a wave-like part with periodicity of the underlying lattice:

$$\psi_{nk}(\mathbf{r}) = e^{i\mathbf{k}\cdot\mathbf{r}}u_{n\mathbf{k}}(\mathbf{r}), \tag{3.44}$$

where $\mathbf{k}$ is a Brillouin zone sample or k-point [17, 68]. Taking into account its periodicity, the wave-like part is expanded as a set of plane waves

$$u_{n\mathbf{k}}(\mathbf{r}) = \frac{1}{\sqrt{\Omega}}\sum_j c_{jn\mathbf{k}}e^{i\mathbf{g}_j\cdot\mathbf{r}}, \tag{3.45}$$

where $\mathbf{g}_j$ are vectors of the reciprocal lattice, and Ω is the unit cell volume. The number of basis functions is controlled by a single parameter, namely the energy cutoff E_{cut}. Only expansion coefficients $c_{jn\mathbf{k}}$ related to plane waves with a kinetic energy $1/2(\mathbf{k}+\mathbf{g}_j) < E_{\text{cut}}$ are retained. This allows easy and systematic control of the quality of a plane wave basis set.

An inherent advantage of plane waves is that they are independent of the atomic positions and thus do not suffer from basis set superposition errors like localized basis sets. On the downside, because of their implicit periodicity, they are not well suited for applications to nonperiodic systems. However, electronic structure calculations of isolated molecules are possible by placing the molecule at the center of a periodic supercell that is sufficiently large for the interactions of molecules in neighboring cells to be negligible. Plane wave basis sets are usually used in combination with Pseudopotentials [69, 70] or the projector-augmented wave (PAW) method [71] such that the plane waves have to describe only the valence orbitals and valence electron density. Expanding core orbitals and the features of other orbitals close to the atomic nuclei would otherwise require a very large number of plane waves, that is, a very large energy cutoff. Despite this, many functions are required to expand the orbitals, which renders the solution of the KS equations via full diagonalization impractical. Instead, iterative schemes such as conjugate gradient or Davidson algorithms [72] are employed during which the full matrix representation of the Hamiltonian is never built. Chapter 7 of this book deals with corresponding techniques and their implementation on GPUs.

3.4.4 Representations on a Numerical Grid

In place of localized basis functions or plane waves, electronic structure calculations can also employ real-space discretizations on a numerical grid [73–75]. In this case, physical quantities such as the orbitals and the electron density are explicitly represented via their numerical values at the grid points, while operators such as the Laplacian of the kinetic energy are approximated via finite differences. The simplest approach is to use uniform grids [74], while nonuniform grids can be employed to reduce the number of points in regions of space that require lower resolution. Similar to plane wave methods, grid-based implementations usually employ pseudopotentials or related approaches such as the PAW method [71] to circumvent the problem of representing the nuclear Coulomb potential and the complicated shape of the core orbitals near the nuclei.

An advantage of numerical grids is that the discretization error can be controlled systematically by increasing the number of grid points. This is harder to achieve with localized basis sets based on GTFs or STFs, where the choice of basis typically requires considerable experience by the user. Real-space grids obviously are well suited for parallelization, and GPU implementations of grid-based ground- and excited-state DFT methods are discussed in Chapters 9 and 10.

3.4.5 Auxiliary Basis Sets

As mentioned previously, three- and four-center ERIs cannot be computed if STFs are employed as the orbital basis set. In this case, an auxiliary basis set $\{\eta_\alpha\}$ of STFs is used to expand the electron density [63] according to

$$\rho(\mathbf{r}) \approx \sum_\alpha d_\alpha \eta_\alpha(\mathbf{r}). \tag{3.46}$$

The Hartree potential of Eq. (3.25) can then be analytically computed on the points of a numerical integration grid and the corresponding contributions to the energy and matrix elements of the KS potential obtained by numerical quadrature, as explained in more detail in Chapter 5.

Without going into great details, such auxiliary basis expansions are widely used also with Gaussian basis sets and for HF theory and wave function-based electron correlation methods [76, 77]. These methods are sometimes referred to as density fitting or resolution of the identity (RI) methods. In brief, the four-index ERIs of Eq. (3.18) can, in general, be approximated as

$$(\mu\nu|\kappa\lambda) \approx \sum_{\alpha\beta} (\mu\nu|\alpha)[J^{-1}]_{\alpha\beta}(\beta|\kappa\lambda), \tag{3.47}$$

where the two-index quantity $[J^{-1}]$ is the inverse of the Coulomb metric evaluated in the auxiliary basis

$$J_{\alpha\beta} = \int d\mathbf{r}\, d\mathbf{r}' \frac{\eta_\alpha(\mathbf{r})\eta_\beta(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}. \tag{3.48}$$

The three-index ERIs $(\mu\nu|\alpha)$ thus project the basis function products $\phi_\mu\phi_\nu$ onto the space spanned by the auxiliary basis set $\{\eta_\alpha\}$. A variety of other closely related approaches to approximate the four-index ERIs through three-index intermediates are available, including the pseudospectral [78, 79] and Cholesky decomposition [76, 80, 81] techniques.

The auxiliary basis sets used in density fitting have to be optimized both for the corresponding orbital basis set and for the specific application. For instance, the requirements to the auxiliary basis set in DFT methods are different than those in *ab initio* wave function methods. For DFT, only the electron density ρ needs to be well approximated, while in the latter case the auxiliary basis needs to represent more complicated functions such as the orbital products $\psi_i\psi_a$ between occupied and virtual molecular orbitals.

Significant speedups are obtained using density-fitting approaches in HF and DFT calculations for small to medium sized molecules in which linear scaling techniques are not yet efficient. Similarly,

the computational cost of wave function-based methods can be significantly reduced. For instance, the computational cost of second order Møller–Plesset perturbation theory (MP2) is dominated by the $\mathcal{O}(N^5)$ integral transformation from atomic into molecular orbital basis. This four-index transformation can be avoided by density fitting methods [82], which, instead, require transformation of three-index quantities, leading to an effective computational effort of $\mathcal{O}(N^4)$. Chapter 12 of this book presents approaches to accelerate the required linear algebra operations on GPUs.

3.5 Semiempirical Methods

Semiempirical electronic structure methods [18] are SCF methods that derive from Hartree–Fock theory by introducing several approximations that render them computationally extremely efficient. These approximations were initially introduced several decades ago because of lack of computing power. However, semiempirical methods are still in wide use nowadays, in particular for large-scale simulations or long-timescale molecular dynamics simulations that are out of reach for *ab initio* wave function theory or DFT methods. Semiempirical models are efficient since they use valence-only minimal basis sets and neglect many of the integrals that contribute to the Fock matrix. To compensate for these approximations, the remaining integrals are parameterized to achieve good results for geometries, bond enthalpies, and other small-molecule properties. By way of construction, semiempirical methods thus include the correlation energy and relativistic effects through their parameterizations and can achieve accuracies that far surpass Hartree–Fock theory and rival correlated wave function methods if applied to problems that fall within the scope of their parameterization. While semiempirical models have been parameterized for transition metals, most reliable parameterizations are applicable only to main group elements. A common procedure to improve the accuracy of simulations is to train parameter sets that accurately represent specific aspects of molecular systems, for example, barriers of a specific reaction type [83]. This, of course, means that one needs to be very careful when using these parameterizations since they are valid only for simulations that fall within the scope of the reference models that were used to derive the parameters.

3.5.1 Neglect of Diatomic Differential Overlap

The most popular semiempirical methods in use nowadays are based on the modified neglect of diatomic overlap (MNDO) [84] model. As in HF theory, the wave function is a single Slater determinant of molecular orbitals ψ_i, but restricted to valence electrons and expanded in terms of a minimal basis of STFs $\{\phi_\mu\}$, see Eqs. (3.10) and (3.37). MNDO is restricted to s- and p-type Slater functions, that is, an l value of 0 or 1, respectively. This effectively limits the elements that can be accurately described to the first and second row of the periodic table. An extension to d-type functions is available with the MNDO/d [85, 86] method, which enables simulations with a wider range of elements.

The decisive simplification in these semiempirical methods is the neglect of diatomic differential overlap (NDDO) approximation [87]

$$\phi_{\mu_A}\phi_{\nu_B} := \delta_{AB}\phi_{\mu_A}\phi_{\nu_B}, \tag{3.49}$$

where the subscript A indicates the atom on which a basis function is located. Although in reality this is not the case, the NDDO approximation implies that we are working in an orthogonal basis, and hence the overlap matrix becomes a unit matrix

$$\mathbf{S} = \mathbf{1}, \tag{3.50}$$

and the SCF equations reduce to a simple eigenvalue problem

$$\mathbf{FC} = \mathbf{C}\epsilon. \tag{3.51}$$

3.5.2 Fock Matrix Elements

As a consequence of the NDDO approximation, the Fock matrix elements contain at most two-center ERIs and are given as

$$F_{\mu_A\nu_A} = h_{\mu_A\nu_A} + \sum_{\kappa_A\lambda_A} P_{\kappa_A\lambda_A}\left[(\mu_A\nu_A|\kappa_A\lambda_A) - \frac{1}{2}(\mu_A\kappa_A|\nu_A\lambda_A)\right] + \sum_B \sum_{\kappa_B\lambda_B} P_{\kappa_B\lambda_B}(\mu_A\nu_A|\kappa_B\lambda_B), \tag{3.52a}$$

$$F_{\mu_A\nu_B} = h_{\mu_A\nu_B} - \frac{1}{2}\sum_{\kappa_A\lambda_B} P_{\kappa_A\lambda_B}(\mu_A\kappa_A|\nu_B\lambda_B), \tag{3.52b}$$

where $h_{\mu\nu}$ are elements of the one-electron core Hamiltonian and the density matrix elements $P_{\mu\nu}$ and ERIs $(\mu\nu|\kappa\lambda)$ are as defined before.

The one-center core Hamiltonian matrix elements are given as

$$h_{\mu_A\nu_A} = \delta_{\mu_A\nu_A} U_{\mu_A\mu_A} + \sum_{B\neq A} V^B_{\mu_A\nu_A}. \tag{3.53}$$

Here, $U_{\mu\mu}$ parametrizes the one-center matrix elements of the kinetic energy operator and the interaction with the atomic core, that is, $\langle\mu_A| - (1/2)\nabla^2 + V_A|\mu_A\rangle$. The interaction with cores on other atoms, that is, $\langle\mu_A|V_B|\nu_A\rangle$, is given by $V^B_{\mu_A\nu_A}$ and is expressed in terms of two-center ERIs

$$V^B_{\mu_A\nu_A} = -Z_B(\mu_A\nu_A|s_Bs_B), \tag{3.54}$$

where Z_B is the charge of the core of atom B. By core we mean an atom including its nucleus and all its electrons up to the valence shell, that is, all electrons that are not treated explicitly by semiempirical methods.

The two-center core Hamiltonian matrix elements are given as

$$h_{\mu_A\nu_B} = \beta_{\mu_A\nu_B} S_{\mu_A\nu_B}, \tag{3.55}$$

where β_{AB} are called resonance integrals, which essentially parametrize the two-center matrix elements of the kinetic energy operator and the interaction with the cores of all atoms, $\langle\mu_A| - (1/2)\nabla^2 + \sum_C V_C|\nu_B\rangle$, while the distance dependence is given via the overlap integrals $S_{\mu_A\nu_B} = \langle\mu_A|\nu_B\rangle$, which are evaluated analytically. This seems inconsistent with the NDDO approximation, Eq. (3.49), according to which all basis functions are orthogonal and hence the overlap matrix is unity. However, the resonance integrals of Eq. (3.55) are necessary to obtain useful results. Indeed, it can be shown that otherwise no covalent bonding would arise in this model.

In MNDO, the resonance integrals are expressed in terms of atomic parameters (as opposed to diatomic or pair parameters, which would depend on specific atom pairs):

$$\beta_{\mu_A\nu_B} = \frac{\beta_{\mu_A} + \beta_{\nu_B}}{2}, \tag{3.56}$$

where β_{μ_A} is a parameter characteristic of the Slater function ϕ_{μ_A} on atom A.

3.5.3 Two-Electron Repulsion Integrals

What remains is to specify the way in which the ERIs of Eqs. (3.52a), (3.52b), and (3.54) are evaluated. None of the ERIs is evaluated analytically, which is another reason for the computational efficiency of semiempirical methods.

The one-center ERIs are parameterized with numerical values that are much smaller than the analytical values. It is commonly assumed that this implicitly accounts for electron correlation. One usually distinguishes Coulomb and exchange integrals:

$$(\mu_A \mu_A | \nu_A \nu_A) = g_{\mu\nu}, \tag{3.57a}$$

$$(\mu_A \nu_A | \mu_A \nu_A) = h_{\mu\nu}. \tag{3.57b}$$

All other one-center ERIs are zero due to symmetry.

The two-center ERIs $(\mu_A \nu_A | \kappa_B \lambda_B)$ represent the interaction energy between the charge distributions $\phi_{\mu_A}\phi_{\nu_A}$ and $\phi_{\kappa_B}\phi_{\lambda_B}$. These charge distributions are expanded [84–86] in terms of multipole moments M_{lm} with order l and orientation m, which are represented by an appropriate configuration of 2^l point charges of magnitude $1/2^l$ separated by a distance D_l. The two-center ERIs are then evaluated in terms of semiempirical, classical multipole–multipole interactions between these multipole moments

$$(\mu_A \nu_A | \kappa_B \lambda_B) = \sum_{l_1 l_2} \sum_{m=-l_{\min}}^{l_{\min}} [M^A_{l_1 m}, M^B_{l_2 m}], \tag{3.58}$$

where $l_{\min}$ is the lower of the two multipole moments l_1 and l_2 in the summation and, using the Klopman–Ohno formula, with

$$[M^A_{l_1 m}, M^B_{l_2 m}] = \frac{1}{2^{l_1+l_2}} c^A_{l_1 m} c^B_{l_2 m} \sum_i^{2^{l_1}} \sum_j^{2^{l_2}} [R^2_{ij} + (\rho^A_{l_1} + \rho^B_{l_2})^2]^{-1/2}. \tag{3.59}$$

Here, R_{ij} is the distance between the point charges. The additive terms $\rho^A_{l_1}$ and $\rho^B_{l_2}$ are chosen such that, at vanishing interatomic distance ($R_{AB} \to 0$), the ERIs reduce to the corresponding one-center integrals in the monoatomic case. The factors $c^A_{l_1 m}$ and $c^B_{l_2 m}$ ensure that at infinite separation ($R_{AB} \to \infty$) the classical limit of the multipole interactions (which is exact for nonoverlapping charge distributions) is recovered. Details can be found in Refs [85, 86].

3.5.4 Energy and Core Repulsion

The total electronic energy is given as

$$E_{\text{el}} = \frac{1}{2} \sum_{\mu\nu} P_{\mu\nu}(h_{\mu\nu} + F_{\mu\nu}) \tag{3.60}$$

and the total energy is

$$E_{\text{tot}} = E_{\text{el}} + \sum_{A<B} E^{\text{core}}_{AB}, \tag{3.61}$$

where E^{core}_{AB} is the repulsion energy between the cores of an atom pair AB.

Aiming at a balance between electrostatic attractions and repulsions within a molecule, the core repulsion, like the core-electron attraction of Eq. (3.54), is treated in terms of the corresponding two-center ERIs

$$E^{\text{core}}_{AB} = Z_A Z_B (s_A s_A | s_B s_B)[1 + f_{AB}(R_{AB})], \tag{3.62}$$

where $R_{AB} = |\mathbf{R}_A - \mathbf{R}_B|$ is the interatomic distance. The function f_{AB} is an effective atom-pair potential that attempts to compensate for errors introduced by the treatment of the core–electron attraction

and core repulsion according to Eqs. (3.54) and (3.62). This term essentially tries to account for Pauli repulsion, that is, prevents atoms from getting too close to each other. It therefore is repulsive at short distances and vanishes in the limit of infinite interatomic distance.

3.5.5 Models Beyond MNDO

Most successful semiempirical models are based on the NDDO approximation and are derived from the MNDO model. For example, the popular AM1 [88] and PM3 [89] models differ only in the parameters and the form that has been chosen for the core repulsion in Eq. (3.62). Another difference between MNDO and AM1/PM3 is that the former uses identical exponents for s- and p-type basis functions while the latter lifts this restriction.

More recently, orthogonalization corrections have been introduced to rectify some of the problems that plague NDDO approaches. These semiempirical methods of the OM*x* family [90–92] account for the lack of Pauli exchange repulsion in the Fock matrix, leading to numerical results that are generally superior to AM1 or PM3. Common to all semiempirical methods is that the calculation of the Fock matrix requires little computational work such that practically all simulations are dominated by the Fock matrix diagonalization. Chapter 11 of this book discusses in detail how GPUs can be exploited to accelerate the computationally most demanding linear algebra operations, taking as example the OM3 method.

3.6 Density Functional Tight Binding

DFTB is an approximate method based on the Kohn–Sham formulation of DFT. It is derived from a Taylor series expansion of the KS–DFT total energy [19, 20]. Similar to semiempirical quantum chemistry methods, DFTB employs a valence-only minimal basis set and neglects a variety of integrals. This, and the fact that the integrals are tabulated, leads to a computationally very efficient method. As is the case for semiempirical methods, the computational bottleneck is the diagonalization of the Hamiltonian matrix. While DFTB is not discussed in the remainder of this book, we include a short overview here for the sake of completeness.

In DFTB, the electron density is written in terms of a reference electron density ρ^0, which perturbed by some density fluctuation, $\rho = \rho^0 + \delta\rho$. The electronic energy up to third order in the density fluctuation can then be written as

$$\begin{aligned} E^{\mathrm{DFTB3}}[\rho^0 + \delta\rho] =& 2\sum_i \langle\psi_i|\hat{H}[\rho^0]|\psi_i\rangle + E_{\mathrm{xc}}[\rho^0] \\ &- \frac{1}{2}\int d\mathbf{r}d\mathbf{r}'\,\frac{\rho^0(\mathbf{r})\rho^0(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|} - \int d\mathbf{r}\; v_{\mathrm{xc}}[\rho^0]\rho^0(\mathbf{r}) \\ &+ \frac{1}{2}\int d\mathbf{r}\; d\mathbf{r}'\left(\frac{1}{|\mathbf{r}-\mathbf{r}'|} + \left.\frac{\delta^2 E_{\mathrm{xc}}[\rho]}{\delta\rho(\mathbf{r})\delta\rho(\mathbf{r}')}\right|_{\rho^0}\right)\delta\rho(\mathbf{r})\delta\rho(\mathbf{r}') \\ &+ \frac{1}{6}\int d\mathbf{r}\; d\mathbf{r}'\; d\mathbf{r}''\left.\frac{\delta^3 E_{\mathrm{xc}}[\rho]}{\delta\rho(\mathbf{r})\delta\rho(\mathbf{r}')\delta\rho(\mathbf{r}'')}\right|_{\rho^0}\delta\rho(\mathbf{r})\delta\rho(\mathbf{r}')\delta\rho(\mathbf{r}''). \end{aligned} \tag{3.63}$$

The models that are obtained from this expansion are termed DFTB, DFTB2, and DFTB3, depending on the terms that are retained. DFTB2 also used to be referred to as self-consistent-charge DFTB (SCC-DFTB), since the density fluctuation $\delta\rho$ has to be obtained from an SCF procedure.

The density fluctuation is expanded in terms of atomic contributions, which in turn are expressed via a multipole expansion of which only the monopole term is kept. The atomic charge density

fluctuations δq_A are thus expressed in terms of the Mulliken charges Δq_A. With several other approximations in place, the DFTB3 energy is finally written as [20, 93]

$$E^{\mathrm{DFTB3}} = \sum_{\mu\nu} P_{\mu\nu} H^0_{\mu\nu} + \frac{1}{2} \sum_{AB} \Delta q_A \Delta q_B \gamma^h_{AB} + \frac{1}{3} \sum_{AB} \Delta q_A^2 \Delta q_B \Gamma_{AB} + \frac{1}{2} \sum_{AB} V^{\mathrm{rep}}_{AB}. \tag{3.64}$$

The KS orbitals are expanded in a valence-only minimal basis set $\{\phi_\mu\}$ of Slater-type confined atomic Orbitals [94]. The expansion coefficients are determined from the KS equations, with Hamiltonian matrix elements given as

$$H^{\mathrm{DFTB3}}_{\mu\nu} = H^0_{\mu\nu} + S_{\mu\nu} \sum_C \Delta q_C \left[\frac{1}{2} (\gamma^h_{AC} + \gamma^h_{BC}) + \frac{1}{3} (\Delta q_A \Gamma_{AC} + \Delta q_B \Gamma_{BC}) + \frac{\Delta q_C}{6} (\Gamma_{AC} + \Gamma_{BC}) \right]. \tag{3.65}$$

In above equations, the tight-binding matrix elements neglect three-center elements

$$H^0_{\mu\nu} = \left\langle \mu \left| -\frac{1}{2}\nabla^2 + V^{\mathrm{eff}}_A + V^{\mathrm{eff}}_B \right| \nu \right\rangle, \mu \in \{A\}, \nu \in \{B\}, \tag{3.66}$$

and the effective potentials V^{eff}_A are parameterized for each atom type. The Hamiltonian matrix elements $H^0_{\mu\nu}$ and the overlap matrix elements $S_{\mu\nu}$ are precalculated and tabulated for relevant pairwise interatomic distances. The short-range pairwise repulsive potentials V^{rep}_{AB} are typically parameterized by comparison to reference DFT data, but they can also be fitted to empirical data [20]. The analytical function γ_{AB} parameterizes the interaction between the charge density fluctuations δq_A and δq_B, which reduces to the standard Coulomb interaction between the partial charges Δq_A and Δq_B in the limit of large separation. At short range, it describes the effective on-site electron–electron interaction, which evaluates to the Hubbard parameter U_A (or atomic chemical hardness) and thus implicitly takes the xc contribution to the second-order term into account [20]. The superscript h indicates a reparameterization of this term specifically for DFTB3, as opposed to DFTB2. Finally, the third-order terms that are parameterized with the function Γ_{AB} describe the change in chemical hardness of an atom depending on its charge and thus contain the chemical hardness derivative U^d_A as parameter [20].

Since DFTB is an approximation to DFT, it also inherits many of its limitations, including the fact that it does not account for long-range dispersion forces. However, parameterizations have been developed to include these effects via empirical pair potentials [19, 20, 95]. Similar to semiempirical methods, DFTB parameters are not available for the complete periodic table and are not transferable for all types of applications. Several specific parameterizations are available for organic or biological molecules or for applications in materials science [20]. These have been successfully used to model a broad range of systems ranging from molecules to solids. Because of the approximations outlined above, the computational efficiency of DFTB is comparable to that of semiempirical methods, enabling large-scale simulations that are out of reach with DFT.

3.7 Wave Function-Based Electron Correlation Methods

In Hartree–Fock theory, the wave function is approximated as a single Slater determinant Φ, effectively replacing the exact many-electron interaction with an average interaction that does not describe the instantaneous Coulomb repulsion between electrons. This is a remarkably good approximation

but, in general, not useful for quantitative calculations and can even lead to qualitatively incorrect results.

The error that is inherent to the HF approximation is quantified as correlation energy, which is defined as the difference between the HF energy and the exact nonrelativistic energy. Since a Slater determinant satisfies the antisymmetry of the wave function, HF theory accounts for Fermi correlation, that is, correlation between the spin coordinates of the electrons, which gives rise to the HF exchange term. The motion of the electrons is also correlated through instantaneous Coulomb repulsions, which is the Coulomb correlation that is missing in HF theory. Wave function-based electron correlation methods, or *ab initio* methods, are based on HF theory and improve upon the HF mean-field approximation by adding many-electron corrections to recover the missing electron correlation in a systematic way [9–11].

Coulomb correlation can be introduced by expanding the wavefunction as a linear combination of Slater determinants:

$$\Psi = a_0\Phi + \sum_i a_i\Phi_i. \tag{3.67}$$

In general, the leading term in this expansion is the HF determinant, that is, the coefficient a_0 is close to 1. The other determinants can be represented in terms of excited HF determinants, in which occupied orbitals of the HF determinant are replaced with virtual (unoccupied) orbitals. Electron correlation methods differ in how they calculate the coefficients for the excited determinants.

In the configuration interaction (CI) method, the coefficients are determined by variational minimization. If all excited determinants are included, this is referred to as full CI, which is the exact solution of the nonrelativistic Schrödinger equation of a many-electron system for a given basis set. Since the number of excited determinants grows factorially with the size of the basis set, the expansion is usually truncated at a certain level of excitations. The resulting methods are then called CISD (single and double excitations), CISDT (including triple excitations), and so forth. CISD scales as $\mathcal{O}(N^6)$, while CISDT scales as $\mathcal{O}(N^8)$, rendering these methods quickly untractable. The CI approach has several drawbacks and has largely been replaced by very successful coupled-cluster methods. A major drawback of CI methods is that they are not size-extensive and as a result recover a diminishing fraction of the electron correlation with increasing molecular size.

Multi-reference methods have to be employed in cases in which a single Slater determinant leads to a qualitatively wrong description of the electronic ground state. The inadequacy of a single Slater determinant to describe an electronic state is usually referred to as static correlation as opposed to dynamical correlation. This is an artificial but useful division of the correlation energy. In the case of static correlation, the major part of the correlation energy can be captured by adding only a few determinants. In multi-configuration self-consistent field (MCSCF) methods, such a limited CI expansion is used to take static correlation into account while variationally optimizing both orbital and CI coefficients. The most widely used multi-reference method is the complete active space self-consistent field (CASSCF) method [96, 97]. Perturbation theory or any other electron correlation method can be used in combination with a multi-reference wave function to capture dynamical correlation effects, leading, for example, to the CASPT2 [98] method, which uses many-body perturbation theory with a CASSCF reference wave function. The main drawback of MCSCF methods is that the selection of the configurations to be included in the reference wave function is not straightforward and requires considerable expertise on side of the user.

In what follows, we assume that a single Slater determinant Φ provides a good zeroth-order description of the electronic ground state.

3.7.1 Møller–Plesset Perturbation Theory

One of the most popular and computationally least expensive *ab initio* electronic structure methods is the second-order Møller–Plesset perturbation theory (MP2) [12]. For compounds that do not contain transition metals, MP2 equilibrium geometries are of comparable quality to DFT. However, MP2 includes long-range correlation effects such as dispersion, which density functionals are not able to capture.

The basis for many-body perturbation theory is a subdivision of the electronic Hamiltonian into a reference part $\hat{H}_0$ for which the solutions are known and a perturbation operator $\hat{V}$. In case of Møller–Plesset perturbation theory, the reference is chosen to be the sum of Fock operators from Hartree–Fock theory:

$$\hat{H} = \sum_i \hat{F}_i + \hat{V}. \tag{3.68}$$

With this partitioning, the perturbation operator is the difference between the exact electron–electron interactions and the HF mean-field potential. The zero-order wave function is the HF determinant, and it is easy to show that the zero-order energy plus first-order correction yields the HF energy, $E_{\mathrm{HF}} = E_0 + E^{(1)}$. The first perturbation correction beyond the HF solution that contributes to the correlation energy is thus given by the second-order energy [9–11]. It involves a sum over doubly excited determinants, which for a closed-shell system can be expressed as

$$E^{(2)} = \sum_{ijab} \frac{(ia|jb)[2(ia|jb) - (ib|ja)]}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b}, \tag{3.69}$$

where i, j and a, b are occupied and virtual (unoccupied) spatial molecular orbitals, respectively, and ϵ_i are the orbital energies. The molecular orbital integrals are obtained via integral transformation as

$$(ia|jb) = \sum_{\mu\nu\kappa\lambda} c_{\mu i} c_{\nu a} c_{\kappa j} c_{\lambda b} (\mu\nu|\kappa\lambda), \tag{3.70}$$

where the ERIs $(\mu\nu|\kappa\lambda)$ in atomic orbital basis have been defined in Eq. (3.18).

The computational cost of canonical MP2 calculations is dominated by the $\mathcal{O}(N^5)$ integral transformation of Eq. (3.70). As outlined in the section on basis sets, density fitting or RI approximations, that is, expansion of molecular orbital products into an auxiliary basis, are routinely employed to reduce the scaling of this integral transformation to $\mathcal{O}(N^4)$ without sacrificing accuracy. Chapter 12 describes approaches to accelerate RI–MP2 calculations by performing the computationally intensive matrix multiplications on GPUs.

3.7.2 Coupled Cluster Theory

In coupled cluster theory [13, 14], the wave function is written in form of an exponential ansatz:

$$\Psi = e^{\hat{T}}\Phi, \tag{3.71}$$

where the cluster operator $\hat{T}$ is given by

$$\hat{T} = \sum_n \hat{T}_n. \tag{3.72}$$

The cluster operators $\hat{T}_n$ generate all nth excited Slater determinants

$$\hat{T}_1 = \sum_{ia} t_i^a a_a^\dagger a_i, \tag{3.73}$$
$$\hat{T}_2 = \sum_{ij,ab} t_{ij}^{ab} a_a^\dagger a_b^\dagger a_j a_i,$$

where a_i and $a_a^\dagger$ are annihilation and creation operators, respectively, and t_i^a and t_{ij}^{ab} are the cluster amplites that need to be determined.

Using the coupled cluster wave function from Eq. (3.71), the Schrödinger equation can be written as

$$\hat{H}e^{\hat{T}}\Phi = Ee^{\hat{T}}\Phi. \tag{3.74}$$

In order to derive working equations, the Schrödinger equation is pre-multiplied by the inverse of the exponential operator $e^{-\hat{T}}$. Left projection by the reference wave function leads to an expression for the energy:

$$E = \langle\Phi|e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Phi\rangle, \tag{3.75}$$

while left projection by the excited determinants leads to expressions for the determination of the cluster amplitudes. The single and double excitation amplitudes, for example, are obtained from

$$\langle\Phi_i^a|e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Phi\rangle = 0, \tag{3.76}$$
$$\langle\Phi_{ij}^{ab}|e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Phi\rangle = 0.$$

The result is a set of nonlinear equations that need to be solved in an iterative manner, for example, using the Jacobi method.

Practical applications truncate the cluster operator at a given level, most commonly at double excitations, which leads to the coupled cluster singles doubles (CCSD) method [99]. The CCSD equations involve tensor contractions that scale up to $\mathcal{O}(N^6)$ with system size. Taking advantage of the relationship between coupled cluster theory and many-body perturbation theory, it is possible to construct perturbation-based corrections to account for higher excitation contributions. Undoubtedly the most popular method is CCSD(T) [100], which adds corrections for triple excitations to CCSD and scales as $\mathcal{O}(N^7)$ with system size. CCSD(T) is currently considered as the gold standard of quantum chemistry because of its high accuracy for a wide range of problems. However, because of the steep computational scaling with system size, its application range remains limited to rather small molecules. Efficient algorithms and software implementations that optimally exploit parallel computer hardware are thus essential to extend the reach of coupled cluster calculations.

More details on coupled cluster theory, the required tensor contractions, and strategies for optimization of GPU implementations are described in the last two chapters of this book. Chapter 13 deals with iterative CCD and CCSD methods including density fitting approximations, while the focus of Chapter 14 is noniterative triples corrections for both single- and multi-reference wave functions.

Acknowledgments

Support by the National Science Foundation (award CHE-1416571) is gratefully acknowledged.

References

1. Clary, D.C. (2006) Quantum chemistry of complex systems. *Science*, **314**, 265–266.
2. Carter, E.A. (2008) Challenges in modeling materials properties without experimental input. *Science*, **321**, 800–803.
3. Reiher, M. (ed.) (2007) *Atomistic Approaches in Modern Biology*, Topics in Current Chemistry, Springer-Verlag, Heidelberg.
4. Grunenberg, J. (ed.) (2010) *Computational Spectroscopy*, Wiley-VCH Verlag GmbH, Weinheim.
5. Comba, P. (ed.) (2011) *Modeling of Molecular Properties*, Wiley-VCH, Weinheim, Germany.
6. Kohn, W. (1999) Nobel Lecture: electronic structure of matter – wave functions and density functionals. *Rev. Mod. Phys.*, **71**, 1253–1266.
7. Pople, J.A. (1999) Nobel lecture: quantum chemical models. *Rev. Mod. Phys.*, **71**, 1267–1274.
8. Reiher, M. and Wolf, A. (2009) *Relativistic Quantum Chemistry*, Wiley-VCH Verlag GmbH, Weinheim.
9. Szabo, A. and Ostlund, N.S. (1996) *Modern Quantum Chemistry*, Dover Publications, New York.
10. Helgaker, T., Jørgensen, P. and Olsen, J. (2000) *Molecular Electronic-Structure Theory*, John Wiley & Sons, Ltd, West Sussex.
11. Jensen, F. (2007) *Introduction to Computational Chemistry*, John Wiley & Sons, Ltd, Chichester.
12. Møller, C. and Plesset, M.S. (1934) Note on an approximation treatment for many-electron systems. *Phys. Rev.*, **46**, 618–622.
13. Čížek, J. (1966) On the correlation problem in atomic and molecular systems. Calculation of wavefunction components in Ursell type expansion using quantum field theoretical methods. *J. Chem. Phys.*, **45**, 4256.
14. Crawford, T. and Schaefer, H. III (2000) An introduction to coupled cluster theory for computational chemists, in *Reviews of Computational Chemistry*, vol. **14**, Chapter 2 (eds K. Lipkowitz and D. Boyd), VCH Publishers, New York, pp. 33–136.
15. Parr, R.G. and Yang, W. (1989) *Density-Functional Theory of Atoms and Molecules*, Oxford University Press, Oxford.
16. Koch, W. and Holthausen, M.C. (2000) *A Chemist's Guide to Density Functional Theory*, Wiley-VCH Verlag GmbH, Weinheim.
17. Martin, R.M. (2004) *Electronic Structure: Basic Theory and Practical Methods*, Cambridge University Press, Cambridge.
18. Thiel, W. (2014) Semiempirical quantum-chemical methods. *WIREs Comput. Mol. Sci.*, **4**, 145–157.
19. Seifert, G. and Joswig, J.O. (2012) Density-functional tight binding - an approximate density-functional theory method. *WIREs Comput. Mol. Sci.*, **2**, 456–465.
20. Elstner, M. and Seifert, G. (2014) Density functional tight binding. *Philos. Trans. R. Soc. London, Ser. A*, **372**, 20120483.
21. Kohn, W. (1996) Density functional and density matrix method scaling linearly with the number of atoms. *Phys. Rev. Lett.*, **76**, 3168–3171.
22. Goedecker, S. (1999) Linear scaling electronic structure methods. *Rev. Mod. Phys.*, **71**, 1085–1123.
23. Autschbach, J. and Ziegler, T. (2003) Double perturbation theory: a powerful tool in computational coordination chemistry. *Coord. Chem. Rev.*, **238–239**, 83–126.

24. Neese, F. (2009) Prediction of molecular properties and molecular spectroscopy with density functional theory: from fundamental theory to exchange-coupling. *Coord. Chem. Rev.*, **253**, 526–563.
25. Ceperley, D. and Alder, B. (1986) Quantum Monte Carlo. *Science*, **231**, 555–560.
26. Foulkes, W.M.C., Mitas, L., Needs, R.J. and Rajagopal, G. (2001) Quantum Monte Carlo simulations of solids. *Rev. Mod. Phys.*, **73**, 33–83.
27. Lüchow, A. (2011) Quantum Monte Carlo methods. *WIREs Comput. Mol. Sci.*, **1**, 388–402.
28. Kong, L., Bischoff, F.A. and Valeev, E.F. (2012) Explicitly correlated R12/F12 methods for electronic structure. *Chem. Rev.*, **112**, 75–107.
29. Onida, G., Reining, L. and Rubio, A. (2002) Electronic excitations: density-functional versus many-body Green's-function approaches. *Rev. Mod. Phys.*, **74**, 601–659.
30. Challacombe, M. (1999) A simplified density matrix minimization for linear scaling self-consistent field theory. *J. Chem. Phys.*, **110**, 2332–2342.
31. Salek, P., Høs, S., Thøgersen, L., Jørgensen, P., Manninen, P., Olsen, J. and Jansík, B. (2007) Linear-scaling implementation of molecular electronic self-consistent field theory. *J. Chem. Phys.*, **126**, 114110.
32. Ochsenfeld, C., Kussmann, J. and Lambrecht, D.S. (2007) Linear-scaling methods in quantum chemistry, in *Reviews in Computational Chemistry*, vol. **23** (eds K.B. Lipkowitz and T.R. Cundari), John Wiley & Sons, Inc., New York, pp. 1–82.
33. White, C., Johnson, B., Gill, P. and Head-Gordon, M. (1994) The continuous fast multipole method. *Chem. Phys. Lett.*, **230**, 8–16.
34. Strain, M.C., Scuseria, G.E. and Frisch, M.J. (1996) Achieving linear scaling for the electronic quantum Coulomb problem. *Science*, **271**, 51.
35. Schwegler, E. and Challacombe, M. (1996) Linear scaling computation of the Hartree-Fock exchange matrix. *J. Chem. Phys.*, **105**, 2726–2734.
36. Ochsenfeld, C., White, C.A. and Head-Gordon, M. (1998) Linear and sublinear scaling formation of Hartree-Fock-type exchange matrices. *J. Chem. Phys.*, **109**, 1663–1669.
37. Fiolhais, C., Nogueira, F. and Marques, M.A.L. (2003) *A Primer in Density Functional Theory*, Lecture Notes in Physics, Springer-Verlag, Berlin.
38. Perdew, J.P. and Ruzsinszky, A. (2010) Fourteen easy lessons in density functional theory. *Int. J. Quantum Chem.*, **110**, 2801–2807.
39. Burke, K. (2012) Perspective on density functional theory. *J. Chem. Phys.*, **136**, 150901.
40. Becke, A.D. (2014) Perspective: fifty years of density-functional theory in chemical physics. *J. Chem. Phys.*, **140**, 18A301.
41. Thomas, L.H. (1927) The calculation of atomic fields. *Proc. Cambridge Philos. Soc.*, **23**, 542.
42. Fermi, E. (1927) Un metodo statistico per la determinazione di alcune proprietá dell' atomo. *Rend. Accad. Lincei*, **6**, 602.
43. Slater, J.C. (1951) A simplification of the Hartree-Fock method. *Phys. Rev.*, **81**, 385–390.
44. Dirac, P.A.M. (1930) Note on exchange phenomena in the Thomas atom. *Proc. Cambridge Philos. Soc.*, **26**, 376.
45. Hohenberg, P. and Kohn, W. (1964) Inhomogeneous electron gas. *Phys. Rev.*, **136**, B864–B871.
46. Kohn, W. and Sham, L. (1965) Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, **140**, A1133–A1138.
47. Perdew, J.P. and Schmidt, K. (2001) Jacob's ladder of density functional approximations for the exchange-correlation energy. *AIP Conf. Proc.*, **577**, 1–20.
48. Ceperley, D.M. and Alder, B.J. (1980) Ground state of the electron gas by a stochastic method. *Phys. Rev. Lett.*, **45**, 566–569.
49. Vosko, S., Wilk, L. and Nusair, M. (1980) Accurate spin-dependent electron liquid correlation energies for local spin density calculations: a critical analysis. *Can. J. Phys.*, **58**, 1200.

50. Perdew, J.P. and Wang, Y. (1992) Accurate and simple analytic representation of the electron-gas correlation energy. *Phys. Rev. B*, **45**, 13244–13249.
51. Becke, A.D. (1988) Density-functional exchange-energy approximation with correct asymptotic behavior. *Phys. Rev. A*, **38**, 3098–3100.
52. Lee, C., Yang, W. and Parr, R.G. (1988) Development of the Colle-Salvetti correlation-energy formula into a functional of the electron density. *Phys. Rev. B*, **37**, 785–789.
53. Becke, A.D. (1993) Density-functional thermochemistry. III. The role of exact exchange. *J. Chem. Phys.*, **98**, 5648–5652.
54. Adamo, C. and Barone, V. (1999) Toward reliable density functional methods without adjustable parameters: the PBE0 model. *J. Chem. Phys.*, **110**, 6158–6170.
55. Stephens, P., Devlin, F.J., Chabalowski, C.F. and Frisch, M.J. (1994) Ab-initio calculation of vibrational absorption and circular-dichroism spectra using density-functional force-fields. *J. Phys. Chem.*, **98**, 11623–11627.
56. Görling, A., Ipatov, A., Götz, A.W. and Heßelmann, A. (2010) Density-Functional theory with orbital-dependent functionals: exact-exchange Kohn-Sham and density-functional response methods. *Z. Phys. Chem.*, **224**, 325–342.
57. Yanai, T., Tew, D.P. and Handy, N.C. (2004) A new hybrid exchange-correlation functional using the Coulomb-attenuating method (CAM-B3LYP). *Chem. Phys. Lett.*, **393**, 51–57.
58. Grimme, S., Antony, J., Ehrlich, S. and Krieg, H. (2010) A consistent and accurate Ab initio parametrization of density functional dispersion correction (DFT-D) for the 94 elements H-Pu. *J. Chem. Phys.*, **132**, 154104.
59. Tkatchenko, A. and Scheffler, M. (2009) Accurate molecular van der Waals interactions from ground-state electron density and free-atom reference data. *Phys. Rev. Lett.*, **102**, 6–9.
60. Becke, A.D. (1988) A multicenter numerical integration scheme for polyatomic molecules. *J. Chem. Phys.*, **88**, 2547–2553.
61. Treutler, O. and Ahlrichs, R. (1995) Efficient molecular numerical integration schemes. *J. Chem. Phys.*, **102**, 346–354.
62. Stratmann, R.E., Scuseria, G.E. and Frisch, M.J. (1996) Achieving linear scaling in exchange-correlation density functional quadratures. *Chem. Phys. Lett.*, **257**, 213–223.
63. te Velde, G., Bickelhaupt, F.M., Baerends, E.J., Fonseca Guerra, C., van Gisbergen, S.J.A., Snijders, J.G. and Ziegler, T. (2001) Chemistry with ADF. *J. Comput. Chem.*, **22**, 931–967.
64. Watson, M.A., Handy, N.C. and Cohen, A.J. (2003) Density functional calculations, using Slater basis sets, with exact exchange. *J. Chem. Phys.*, **119**, 6475–6481.
65. van Lenthe, E. and Baerends, E.J. (2003) Optimized Slater-type basis sets for the elements 1–118. *J. Comput. Chem.*, **24**, 1142–1156.
66. Jensen, F. (2013) Atomic orbital basis sets. *WIREs Comput. Mol. Sci.*, **3**, 273–295.
67. Boys, S.F. (1950) Electronic wave functions I. A general method of calculation for the stationary states of any molecular system. *Proc. R. Soc. London Ser. A*, **200**, 542–554.
68. Ashcroft, N.W. and Mermin, N. (1976) *Solid State Physics*, Harcourt College Publishers, Fort Worth, TX.
69. Kleinman, L. and Bylander, D. (1982) Efficacious form for model pseudopotentials. *Phys. Rev. Lett.*, **48**, 1425–1428.
70. Vanderbilt, D. (1990) Soft self-consistent pseudopotentials in a generalized eigenvalue formalism. *Phys. Rev. B*, **41**, 7892–7895.
71. Blöchl, P.E. (1994) Projector augmented-wave method. *Phys. Rev. B*, **50**, 17953–17979.
72. Davidson, E.R. (1975) The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *J. Comput. Phys.*, **17**, 87–94.
73. Becke, A.D. (1989) Basis-set-free density-functional quantum chemistry. *Int. J. Quantum Chem.*, **36**, 599–609.

74. Chelikowsky, J.R., Troullier, N. and Saad, Y. (1994) Finite-difference-pseudopotential method: electronic structure calculations without a basis. *Phys. Rev. Lett.*, **72**, 1240–1243.
75. Beck, T.L. (2009) Real-space and multigrid methods in computational chemistry, in *Reviews in Computational Chemistry* (eds K.B. Lipkowitz and T.R. Cundari), John Wiley & Sons, Inc., New York, pp. 223–285.
76. Weigend, F., Kattannek, M. and Ahlrichs, R. (2009) Approximated electron repulsion integrals: Cholesky decomposition versus resolution of the identity methods. *J. Chem. Phys.*, **130**, 164106.
77. Sherrill, C.D. (2010) Frontiers in electronic structure theory. *J. Chem. Phys.*, **132**, 110902.
78. Friesner, R.A., Murphy, R.B., Beachy, M.D., Ringnalda, M.N., Pollard, W.T., Dunietz, B.D. and Cao, Y. (1999) Correlated ab initio electronic structure calculations for large molecules. *J. Phys. Chem. A*, **103**, 1913–1928.
79. Murphy, R.B., Cao, Y., Beachy, M.D., Ringnalda, M.N. and Friesner, R.A. (2000) Efficient pseudospectral methods for density functional calculations. *J. Chem. Phys.*, **112**, 10131–10141.
80. Beebe, N.H.F. and Linderberg, J. (1977) Simplifications in the generation and transformation of two-electron integrals in molecular calculations. *Int. J. Quantum Chem.*, **12**, 683–705.
81. Koch, H., de Merás, A.S. and Pedersen, T.B. (2003) Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, **118**, 9481–9484.
82. Feyereisen, M.W., Fitzgerald, G. and Komornicki, A. (1993) Use of approximate integrals in ab initio theory. An application in MP2 energy calculations. *Chem. Phys. Lett.*, **208**, 359–363.
83. Gonzalez-Lafont, A., Truong, T.N. and Truhlar, D.G. (1991) Direct dynamics calculations with neglect of diatomic differential overlap molecular orbital theory with specific reaction parameters. *J. Chem. Chem.*, **95**, 4618–4627.
84. Dewar, M. and Thiel, W. (1977) Ground states of molecules. 38. The MNDO method. Approximations and parameters. *J. Am. Chem. Soc.*, **99**, 4899–4907.
85. Thiel, W. and Voityuk, A.A. (1992) Extension of the MNDO formalism to d orbitals: integral approximations and preliminary numerical results. *Theor. Chim. Acta*, **81**, 391–404.
86. Thiel, W. and Voityuk, A.A. (1996) Extension of the MNDO formalism to d orbitals: integral approximations and preliminary numerical results. *Theor. Chim. Acta*, **93**, 315.
87. Pople, J.A., Santry, D.P. and Segal, G.A. (1965) Approximate self-consistent molecular orbital theory. I. Invariant procedures. *J. Chem. Phys.*, **43**, S129.
88. Dewar, M., Zoebisch, E., Healy, E. and Stewart, J. (1985) AM1: a new general purpose quantum mechanical molecular model. *J. Am. Chem. Soc.*, **107**, 3902–3909.
89. Stewart, J.J.P. (1989) Optimization of parameters for semiempirical methods I. Method. *J. Comput. Chem.*, **10**, 209–220.
90. Kolb, M. and Thiel, W. (1993) Beyond the MNDO model: methodical considerations and numerical results. *J. Comput. Chem.*, **14**, 775–789.
91. Weber, W. and Thiel, W. (2000) Orthogonalization corrections for semiempirical methods. *Theor. Chem. Acc.*, **103**, 495–506.
92. Tuttle, T. and Thiel, W. (2008) OMx-D: semiempirical methods with orthogonalization and dispersion corrections. Implementation and biochemical application. *Phys. Chem. Chem. Phys.*, **10**, 2159–2166.
93. Gaus, M., Cui, Q. and Elstner, M. (2012) DFTB3: extension of the self-consistent-charge density-functional tight-binding method (SCC-DFTB). *J. Chem. Theory Comput.*, **7**, 931–948.
94. Eschrig, H. (1989) *Optimized LCAO Method and Electronic Structure of Extended Systems*, Springer-Verlag, Berlin.
95. Zhechkov, L., Heine, T., Patchkovskii, S., Seifert, G. and Duarte, H.A. (2005) An efficient a posteriori treatment for dispersion interaction in density-functional-based tight binding. *J. Chem. Theory Comput.*, **1**, 841–847.

96. Siegbahn, P.E.M. (1981) The complete active space SCF (CASSCF) method in a Newton-Raphson formulation with application to the HNO molecule. *J. Chem. Phys.*, **74**, 2384.
97. Olsen, J. (2011) The CASSCF method: a perspective and commentary. *Int. J. Quantum Chem.*, **111**, 3267–3272.
98. Andersson, K., Malmqvist, P.-A. and Roos, B.O. (1992) Second-order self-consistent perturbation theory with a complete field reference function. *J. Chem. Phys.*, **96**, 1218–1226.
99. Purvis, G.D. and Bartlett, R.J. (1982) A full coupled-cluster singles and doubles model: the inclusion of disconnected triples. *J. Chem. Phys.*, **76**, 1910.
100. Raghavachari, K., Trucks, G.W., Pople, J.A. and Head-Gordon, M. (1989) A fifth-order perturbation comparison of electron correlation theories. *Chem. Phys. Lett.*, **589**, 37–40.

4

Gaussian Basis Set Hartree–Fock, Density Functional Theory, and Beyond on GPUs

Nathan Luehr[1,2]*, Aaron Sisto*[1,2] *and Todd J. Martínez*[1,2]

[1]*Department of Chemistry and the PULSE Institute, Stanford, CA, USA*
[2]*SLAC National Accelerator Laboratory, Menlo Park, CA, USA*

In this chapter, we discuss the GPU acceleration of Hartree–Fock (HF), density functional theory (DFT), and time-dependent density functional theory (TDDFT) methods within Gaussian basis sets. As mentioned in Chapter 3, self-consistent field (SCF) methods such as HF and DFT contain two principal bottlenecks. The first stems from the calculation of the Hamiltonian matrix elements, which requires evaluation of electron–electron repulsion integrals (ERIs). Formally, for a basis set containing N functions, a total of $\mathcal{O}(N^4)$ ERIs must be evaluated. In the asymptotic limit of large systems, efficient screening of negligibly small ERIs can reduce this number to $\mathcal{O}(N^2)$ or, for certain insulating systems, even $\mathcal{O}(N)$ [1–5]. In the case of DFT, the numerical quadrature of the exchange–correlation (XC) potential is also a time-consuming trask. The second bottleneck results from diagonalization of the $N \times N$ Hamiltonian matrix into its eigenvectors and eigenvalues. Eigensolvers applied to dense matrices run with a complexity of $\mathcal{O}(N^3)$. However, using sparse matrix algebra it is possible, again in asymptotically large systems, to achieve $\mathcal{O}(N)$ scaling for the orbital/density update, which is usually addressed by diagonalization [6–8]. Thus, formal asymptotic analysis is of limited use since the dominant bottleneck results from prefactors rather than scaling exponents. Empirically, for systems up to at least 10,000 basis functions, integral evaluation dominates the SCF runtime, and therefore the present chapter focuses primarily on the GPU acceleration of ERI evaluation.

Numerous ERI evaluation schemes have been developed for use in traditional CPU codes. For very high angular momentum, Rys quadrature methods [9] may provide an advantage on GPUs due to their smaller memory footprint [10–12]. For low angular momentum functions, we find little performance difference between the Rys and the simpler McMurchie–Davidson [13] approach, and prefer the simplicity of the latter for discussions in this chapter.

Electronic Structure Calculations on Graphics Processing Units:
From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

4.1 Quantum Chemistry Review

Chapter 3 contains a general overview of electronic structure methods including HF and DFT methods. Space does not permit full derivations of these methods, nor is an in-depth description of various ERI evaluation algorithms possible in this book. These can be found elsewhere [14–16]. Here, we provide brief overviews in order to put the subsequent discussion of GPU acceleration into context. Unless otherwise noted, we assume a spin-restricted wave function ansatz and atomic units throughout the chapter.

4.1.1 Self-Consistent Field Equations in Gaussian Basis Sets

A primitive Gaussian function is defined as follows:

$$\chi_i(\vec{r}) = N(\vec{r}_x - x_i)^{n_i}(\vec{r}_y - y_i)^{l_i}(\vec{r}_z - z_i)^{m_i} \exp(-\alpha_i|\vec{r} - \vec{R}_i|^2). \tag{4.1}$$

Here, r is the three-dimensional electronic coordinate, N is a normalization constant, $R_i = (x_i, y_i, z_i)$ is the primitive's Cartesian center (usually coinciding with one of the atoms in the molecule), and α_i is an exponent determining the spatial extent of the function. The nonnegative integers n_i, l_i, and m_i fix the function's angular momentum, and their sum $\lambda_i = n_i + l_i + m_i$ gives the primitive's total angular momentum. Functions with $\lambda = 0, 1, 2$ are termed, s-, p-, and d-functions, respectively. The set of $(\lambda + 1)(\lambda + 2)/2$ primitive functions sharing a common center, exponent, and total momentum is referred to as a shell. In order to more closely approximate the solutions to the atomic Schrödinger equation, several primitive functions (centered on the same atom) are combined together into a contracted basis function using fixed contraction weights $c_{\mu i}$:

$$\phi_\mu(\vec{r}) = \sum_i c_{\mu i}\chi_i(\vec{r}). \tag{4.2}$$

These contracted basis functions will be referred to as atomic orbitals (AOs) in the following.

The AOs themselves are combined by linear contraction into molecular orbitals (MOs), each of which is related to the one-particle spatial probability distribution for an electron in the system:

$$\theta_i(\vec{r}) = \sum_\mu^N C_{\mu i}\phi_\mu(\vec{r}). \tag{4.3}$$

The MO coefficients $C_{\mu i}$ are free parameters, and their determination is the primary objective of the SCF procedure. In order to describe an n-electron system, the one-electron MOs are combined with spin functions in a Slater determinant:

$$\Psi(\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n) = \frac{1}{\sqrt{n!}}\begin{vmatrix} \psi_1(\vec{x}_1) & \psi_2(\vec{x}_1) & \cdots & \psi_n(\vec{x}_1) \\ \psi_1(\vec{x}_2) & \psi_2(\vec{x}_2) & \cdots & \psi_n(\vec{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_1(\vec{x}_n) & \psi_2(\vec{x}_n) & \cdots & \psi_n(\vec{x}_n) \end{vmatrix}. \tag{4.4}$$

For the spin-restricted case in which two electrons occupy each spatial orbital, the spin orbitals (depending on both spatial and spin electronic degrees of freedom) can be defined as follows (where σ_k is the spin degree of freedom for the kth electron):

$$\begin{aligned} \psi_{2n-1}(\vec{x}_k) &= \theta_n(\vec{r}_k)\alpha(\sigma_k), \\ \psi_{2n}(\vec{x}_k) &= \theta_n(\vec{r}_k)\beta(\sigma_k). \end{aligned} \tag{4.5}$$

The energy of the wave function Ψ representing n electrons in a system containing A fixed atomic nuclei (each with charge Z_a and located at position R_a) is derived from the expectation value of the electronic Hamiltonian $\widehat{H}$:

$$\widehat{H} = \sum_{i}^{n}\left(\sum_{a}^{A}\frac{Z_a}{\left|\vec{r}_i - \vec{R}_a\right|} - \frac{\nabla_i^2}{2} + \frac{1}{2}\sum_{j\neq i}\frac{1}{|\vec{r}_i - \vec{r}_j|}\right) \tag{4.6}$$

$$E_{\mathrm{RHF}} = \frac{\langle\Psi|\widehat{H}|\Psi\rangle}{\langle\Psi|\Psi\rangle} = 2\sum_{i}^{n/2}\langle\theta_i|\widehat{h}|\theta_i\rangle + 2\sum_{i,j}^{n/2}(\theta_i\theta_i|\theta_j\theta_j) - \sum_{i,j}^{n/2}(\theta_i\theta_j|\theta_i\theta_j). \tag{4.7}$$

Here the MOs are assumed, without loss of generality, to be orthonormal.

$$\langle\theta_i|\theta_j\rangle = \delta_{ij}. \tag{4.8}$$

The one-electron core Hamiltonian operator $\widehat{h}$ accounts for electron–nuclear attraction and electron kinetic energy:

$$\widehat{h}(\vec{r}) = \sum_{a}^{A}\frac{Z_a}{|\vec{r} - \vec{R}_a|} - \frac{\nabla^2}{2} \tag{4.9}$$

and the ERIs account for pairwise interactions between electrons.

$$(\theta_i\theta_j|\theta_k\theta_l) = \int d^3\vec{r}_1 \int d^3\vec{r}_2 \frac{\theta_i^*(\vec{r}_1)\theta_j(\vec{r}_1)\theta_k^*(\vec{r}_2)\theta_l(\vec{r}_2)}{|\vec{r}_1 - \vec{r}_2|}. \tag{4.10}$$

For Kohn–Sham DFT, a similar energy expression is obtained by using the determinant to describe noninteracting pseudo-particles whose total density matches the ground-state electron density:.

$$\rho(\vec{r}) = 2\sum_{i}^{n/2}|\theta_i(\vec{r})|^2. \tag{4.11}$$

In this case, components of the Hartree–Fock energy provide good approximations for the DFT kinetic energy and classical electron repulsion. An additional exchange–correlation functional E_{XC} corrects for the relatively small energetic effects of electron exchange and correlation as well as errors from approximating the kinetic energy as that of the Kohn–Sham determinant:

$$E_{\mathrm{DFT}} = 2\sum_{i}^{n/2}\langle\theta_i|\widehat{h}|\theta_i\rangle + 2\sum_{i,j}^{n/2}(\theta_i\theta_i|\theta_j\theta_j) + E_{\mathrm{XC}}[\rho]. \tag{4.12}$$

Given the exact exchange–correlation functional $E_{\mathrm{XC}}[\rho]$, Eq. (4.12) would provide the exact ground-state energy. Unfortunately, the exact functional is not known in any computationally feasible form. In practice, a variety of approximate functionals are often employed. For simplicity, we focus on the remarkably successful class of generalized gradient approximation (GGA) functionals. These take the form of an integral over a local XC kernel, which depends only on the total density and its gradient:

$$E_{\mathrm{XC}}[\rho] = \int f_{\mathrm{xc}}(\rho(r), \nabla\rho(r))d\vec{r}. \tag{4.13}$$

To calculate the HF or DFT ground-state electronic configuration, we vary the MO coefficients $C_{\mu i}$ to minimize E_{RHF} or E_{DFT} under the constraint of Eq. (4.8) that the MOs remain orthonormal. Functional variation ultimately results in the following conditions on the MO coefficients:

$$\mathbf{F(P)C = ESC}. \tag{4.14}$$

Here, **P** is the density matrix represented in the AO basis:

$$P_{\mu\nu} = \sum_{i}^{n} C_{\mu i} C_{\nu i}^{*}. \tag{4.15}$$

E is a diagonal matrix of MO energies (formally, this matrix is the set of Lagrange multipliers enforcing the constraint that all the molecular orbitals remain orthonormal, i.e., Eq. (4.8)); **S** is the AO overlap matrix

$$S_{\mu\nu} = \langle \phi_{\mu} | \phi_{\nu} \rangle \tag{4.16}$$

and **F(P)** is the nonlinear Fock operator, defined slightly differently for HF and DFT as follows:

$$F_{\mu\nu}^{\mathrm{HF}}(P) = h_{\mu\nu} + \sum_{\lambda\sigma}^{N} P_{\lambda\sigma}[2(\mu\nu|\sigma\lambda) - (\mu\lambda|\nu\sigma)], \tag{4.17}$$

$$F_{\mu\nu}^{\mathrm{DFT}}(P) = h_{\mu\nu} + 2\sum_{\lambda\sigma}^{N} (\mu\nu|\sigma\lambda) P_{\lambda\sigma} + V_{\mu\nu}^{\mathrm{XC}}, \tag{4.18}$$

where **h** is the core Hamiltonian from Eq. (4. 9) now in the AO basis:

$$h_{\mu\nu} = \left\langle \phi_{\mu} \left| \sum_{a}^{A} \frac{Z_a}{\left| \vec{r}_1 - \vec{R}_a \right|} - \frac{\nabla^2}{2} \right| \phi_{\nu} \right\rangle \tag{4.19}$$

and the two-electron ERIs are defined in the AO basis as follows:

$$\begin{aligned}
(\mu\nu|\lambda\sigma) &= \int d^3\vec{r}_1 \int d^3\vec{r}_2 \frac{\phi_{\mu}^{*}(\vec{r}_1)\phi_{\nu}(\vec{r}_1)\phi_{\lambda}^{*}(\vec{r}_2)\phi_{\sigma}(\vec{r}_2)}{|\vec{r}_1 - \vec{r}_2|} \\
&= \sum_{i\in\phi_{\mu}} \sum_{j\in\phi_{\nu}} \sum_{k\in\phi_{\lambda}} \sum_{l\in\phi_{\sigma}} c_{\mu i}^{*} c_{\nu j} c_{\lambda k}^{*} c_{\sigma l} \int d^3\vec{r}_1 \int d^3\vec{r}_2 \frac{\chi_i^{*}(\vec{r}_1)\chi_j(\vec{r}_1)\chi_k^{*}(\vec{r}_2)\chi_l(\vec{r}_2)}{|\vec{r}_1 - \vec{r}_2|} \\
&= \sum_{i\in\phi_{\mu}} \sum_{j\in\phi_{\nu}} \sum_{k\in\phi_{\lambda}} \sum_{l\in\phi_{\sigma}} c_{\mu i}^{*} c_{\nu j} c_{\lambda k}^{*} c_{\sigma l} [ij|kl].
\end{aligned} \tag{4.20}$$

Note that we use round braces to refer to ERIs involving contracted basis functions and square braces to refer to primitive ERIs. Also, to specify that the ith primitive is restricted to members of the μth atomic orbital, we use the notation $i \in \phi_{\mu}$. Finally, for DFT, V^{XC} is determined by functional differentiation of the exchange–correlation energy expression:

$$V_{\mu\nu}^{\mathrm{XC}} = \left\langle \theta_{\mu} \left| \frac{\delta E_{\mathrm{XC}}}{\delta \rho} \right| \theta_{\nu} \right\rangle. \tag{4.21}$$

Because the HF and DFT Fock operators are nonlinear, Eq. (4.14) cannot be solved in a closed form; instead, an iterative approach is used. Starting from some guess for the density matrix **P**, the Fock matrix is constructed and then diagonalized to obtain a matrix of approximate MO orbitals, **C**. The MO coefficients **C** are then used to construct an improved guess for the density matrix using Eq. (4.15), and the process is repeated until **F** and **P** converge to stable values.

4.1.2 Electron–Electron Repulsion Integral Evaluation

The generation and contraction of ERIs into the Fock matrix is the main bottleneck in finite-basis SCF algorithms, and therefore our primary focus in the present chapter. Here, we provide mathematical background which will be useful when considering the implementation of ERIs on GPUs in subsequent sections. Efficient ERI evaluation begins by invoking the Gaussian product theorem, which allows a product of two primitive Gaussian functions located at different centers to be combined into a single Gaussian centered at a point $\overrightarrow{P}$ between the original centers [17]:

$$\begin{aligned}
\Omega_{ij} &= e^{-\alpha_i(\vec{r}-\vec{R}_i)^2} e^{-\alpha_j(\vec{r}-\vec{R}_j)^2} = K_{ij} e^{-\eta_{ij}(\vec{r}-\vec{P}_{ij})^2}, \\
\eta_{ij} &= \alpha_i + \alpha_j, \\
K_{ij} &= e^{-\frac{\alpha_i\alpha_j}{\alpha_i+\alpha_j}(\vec{R}_i-\vec{R}_j)^2}, \\
\overrightarrow{P}_{ij} &= \frac{\alpha_i\overrightarrow{R}_i + \alpha_j\overrightarrow{R}_j}{\alpha_i+\alpha_j}.
\end{aligned} \tag{4.22}$$

This reduces each four-center primitive ERI to a simpler two-center problem:

$$[ij|kl] = [\Omega_{ij}|\Omega_{kl}]. \tag{4.23}$$

Following the McMurchie–Davidson scheme for ERI evaluation, each primitive pair is exactly expanded in a basis of Hermite Gaussians $\{\Lambda_t\}$ [13]:

$$\Omega_{[ij]} = \sum_{t=0} E_t^{[ij]} \Lambda_t. \tag{4.24}$$

The expansion coefficients $E_t^{[ij]}$ differ for each primitive pair $\chi_i\chi_j$ and are calculated from simple recurrence relations [13]. The primitive ERI then becomes

$$[ij|kl] = \sum_p \sum_q E_p^{[ij]} E_q^{[kl]} [\Lambda_p|\Lambda_q]. \tag{4.25}$$

The Hermite ERIs $[\Lambda_p|\Lambda_q]$ are efficiently calculated by recurrence relations starting from the Boys function $F_n(x)$ [13]:

$$F_n(x) = \int_0^1 t^{2n} e^{-xt^2} dt. \tag{4.26}$$

Although the Fock matrices of Eqs. (4.17) and (4.18) involve contributions from N^4 ERIs, many individual terms may be neglected. Because each AO basis function is localized in space, a pair distribution Ω_{ij} will approach zero exponentially as the distance between primitive functions increases. Thus, an AO ERI $(\mu\nu|\sigma\lambda)$ will be nonnegligible only if μ is centered near ν and λ is near σ. For large systems, this reduces the number of integrals to a more manageable N^2. In order to efficiently identify significant ERIs, a Schwarz inequality can be applied to either contracted or primitive integrals [18]:

$$|(\mu\nu|\lambda\sigma)| \leq (\mu\nu|\mu\nu)^{1/2} (\lambda\sigma|\lambda\sigma)^{1/2}. \tag{4.27}$$

4.2 Hardware and CUDA Overview

While Chapter 2 provided an introduction to the concept of GPU programming, we believe it is useful to the reader to briefly repeat some of the essential points as they pertain to GPU acceleration of ERI evaluation. As mentioned previously, each GPU is a massively parallel device, containing thousands of execution cores. However, the performance of these processors results not only from the raw width of execution units but also from a hierarchy of parallelism that forms the foundation of the hardware architecture and, in the case of Nvidia hardware, is ingeniously exposed to the programmer through the CUDA programming model [19]. Developers must understand and respect these hierarchical boundaries if their programs are to run efficiently on GPUs.

At the lowest level, the CUDA programmer writes a small procedure – called a kernel in "CUDA-speak" – that is to be executed by tens of thousands of individual threads in parallel. Although each CUDA thread is logically autonomous, the hardware does not execute each thread independently. Instead, instructions are scheduled for groups of 32 threads, called warps, in a single-instruction multiple-thread (SIMT) manner. Every thread in a warp executes the same instruction stream, with threads masked to null operations (no-ops) for instructions in which they do not participate.

Warps are grouped into larger blocks of up to 1024 threads. Blocks are assigned to local groups of execution units called streaming multiprocessors (SMs). The SM provides hardware-based intra-block synchronization methods, and a small on-chip shared memory is often used for intra-block communication. CUDA blocks can be indexed in one, two, or three dimensions at the convenience of the programmer.

At the highest level, blocks are organized into a CUDA grid. As with blocks, the grid can have up to three dimensions. In general, the grid contains many more blocks and threads than the GPU has physical execution units. When a grid is launched, a hardware scheduler streams CUDA blocks onto the processors. By breaking a task into fine-grained units of work, the GPU can be kept constantly busy, maximizing performance.

As highlighted in Chapter 2 in CUDA, the memory is also structured hierarchically. The host (CPU) memory usually provides the largest space, but can only be accessed through the PCIe interface, which suffers from latencies on the order of several thousand instruction cycles. The GPU's main (global) memory provides several gigabytes of high-bandwidth memory capable of more than 250 GB/s of sustained throughput. In order to enable this bandwidth, global memory accesses incur long latencies, on the order of >500 clock cycles. Global memory operations are parallelized in a SIMT-friendly manner in which the natural width of the memory controller allows simultaneous access by all threads of a warp as long as those threads target contiguous memory locations. Low-latency, on-chip memory is also available. Most usefully, each block can use up to 64 KB of shared memory for intrablock communication, and each thread has up to 255 local registers in which to store intermediate results.

Consideration of the basic hardware design suggests the following basic strategies for maximizing the performance of GPU kernels:

1. Launch *many* threads, ideally one to two orders of magnitude more threads than the GPU has execution cores. For example, a Tesla K20 with 2496 cores may not reach peak performance until at least $\mathcal{O}(10^5)$ threads are launched. Having thousands of processors will be an advantage only if they are all saturated with work. All threads are hardware-scheduled, making them very lightweight to create, unlike host threads. Also, the streaming model ensures that the GPU will not execute more threads than it can efficiently schedule. Thus oversubscription will not throttle performance. Context switches are also instantaneous, and this is beneficial because they allow a processor to stay busy when it might otherwise be stalled, for example, waiting for a memory transaction to complete.

2. Keep each thread as simple as possible. Threads with smaller shared memory and register footprints can be packed more densely onto each SM. This allows the schedulers to hide execution and memory latencies by increasing the chance that a ready-to-execute warp will be available on any given clock cycle.
3. Decouple the algorithm to be as data-parallel as possible. Synchronization between threads always reduces the effective concurrency available to the GPU schedulers, and should be minimized. For example, it is often better to recompute intermediate quantities rather than build shared caches, sometimes even when the intermediates require hundreds of cycles to compute.
4. Maintain regular memory access patterns. On the CPU this is done temporally within a single thread; on the GPU it is more important to do it locally among threads in a warp.
5. Maintain uniform control flow within a warp. Because of the SIMT execution paradigm, all threads in the warp effectively execute every instruction needed by any thread in the warp. Pre-organizing work by expected code-path can eliminate divergent control flow within each warp and improve performance.

These strategies have well-known analogs for CPU programming; however, the performance penalty resulting from their violation is usually much more severe in the case of the GPU. The tiny size of GPU caches relative to the large number of in-flight threads defeats any possibility of cushioning the performance impact of nonideal programming patterns. In such cases, the task of optimization goes far beyond simple FLOP minimization, and the programmer must consider tradeoffs from each of the above considerations on his design.

4.3 GPU ERI Evaluation

Turning now to the task of GPU acceleration, we first consider the simplified problem of evaluating contracted ERIs within a basis of *s*-functions. For such functions, a primitive ERI can be evaluated as follows:

$$[ij|kl] = \frac{\pi^3 K_{ij} K_{kl}}{\eta_{ij}\eta_{kl}\sqrt{\eta_{ij}+\eta_{kl}}} F_0\left(\frac{\eta_{ij}\eta_{kl}}{\eta_{ij}+\eta_{kl}}\left|\vec{P}_{ij}-\vec{P}_{kl}\right|^2\right), \tag{4.28}$$

where F_0 is the Boys function of Eq. (4.26), and K, η, and $\vec{P}$ are primitive pair quantities (PQs) defined in Eq. (4.22).

A convenient way to organize ERI evaluation is to contract unique pairs of atomic orbitals $\{\phi_\mu\phi_\nu|\mu \le \nu\}$ into a vector of dimension $N(N-1)/2$. A generalized outer product of this vector with itself then produces a matrix whose elements are quartets $\{\phi_\mu\phi_\nu, \phi_\lambda\phi_\sigma|\mu \le \nu, \lambda \le \sigma\}$, each representing a (bralket) integral. Because of $(bra|ket) = (ket|bra)$ symmetry, only the upper triangle of the integral matrix needs be computed. Such an ERI matrix is illustrated in the left half of Figure 4.1.

In applying the CUDA model, it is not difficult to break up the task of ERI evaluation into independent units of work. Natural divisions occur between each contracted ERI $(\mu\nu|\lambda\sigma)$ and at a finer grain between primitive integrals $[ij|kl]$. Because of the quantity of available parallel work, there are many possible strategies to map ERIs onto GPU threads. We describe three broadly representative schemes [12, 20]. The first assigns a CUDA block to evaluate each contracted ERI and maps a two-dimensional (2D) CUDA grid onto the 2D ERI grid, shown in Figure 4.1. The threads within each block work together to compute a contracted integral in parallel. We term this the one-block-one-contracted integral (1B1CI) scheme. The second strategy assigns entire contracted integrals to individual CUDA threads. This coarser decomposition we term the one-thread-one-contracted integral (1T1CI) strategy. The final decomposition strategy maps each thread to a single primitive integral (1T1PI) and ignores boundaries between primitives belonging to different AO contractions. A second reduction step is then employed to sum the final contracted integrals from their constituent primitive contributions.

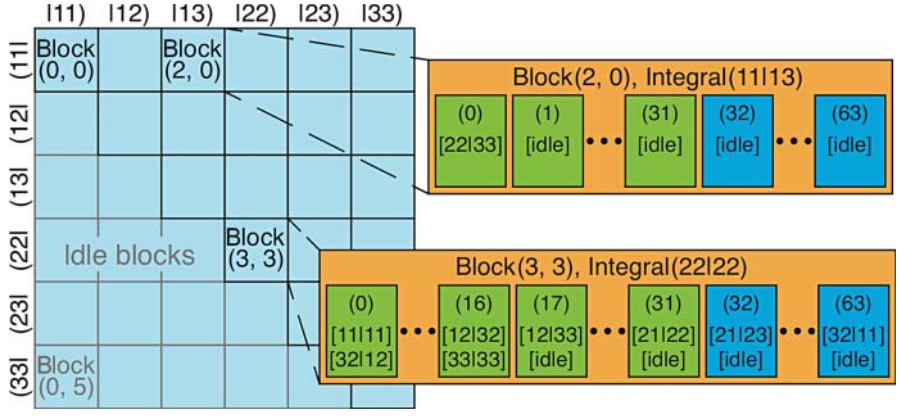


Figure 4.1 *Schematic of one-block one-contracted Integral (1B1CI) mapping. Cyan squares on left represent contracted ERIs each mapped to the labeled CUDA block of 64 threads. Orange squares show mapping of primitive ERIs to CUDA threads (green and blue boxes, colored according to CUDA warp) for two representative integrals, the first a contraction over a single primitive ERI and the second involving $3^4 = 81$ primitive contributions. (See insert for colour representation of this figure)*

4.3.1 One-Block-One-Contracted Integral

Figure 4.1 shows a schematic representation of the 1B1CI mapping. Each cyan square represents a contracted integral that we wish to calculate. The CUDA block responsible for each contracted ERI is labeled within the square. Lower triangular blocks, labeled *idle* in Figure 4.1, would compute redundant integrals due to $(bra|ket) = (ket|bra)$ symmetry. These blocks exit immediately and, because of the GPU's efficient thread scheduling, contribute minimally to the overall execution time. Each CUDA block is made up of 64 worker threads arranged in a single dimension. Blocks are represented by orange rectangles in Figure 4.1. The primitive integrals are mapped cyclically onto the threads, and each thread collects a partial sum in an on-chip register.

The first thread computes and accumulates integrals 1, 65, and so on, while the second thread handles integrals 2, 66, and so on. After all primitive integrals have been evaluated, a block-level reduction produces the final contracted integral.

PQs for bra and ket primitive pairs $\{\chi_i\chi_j|\chi_i \in \phi_\mu, \chi_j \in \phi_\nu\}$ are precomputed on the host and stored in vectors sorted by a combined AO index $\mu\nu$. This allows fetches from neighboring threads to be coalesced when fetching input for each primitive ERI. To fully satisfy CUDA coalescence constraints, the primitive PQs are packed into multiple arrays of CUDA vector types, for example, a float4 array for $\vec{P}_x$, $\vec{P}_y$, $\vec{P}_z$, and η and an additional float array for K values.

Two cases deserving particular consideration are illustrated in Figure 4.1. The upper thread block shows what happens for very short contractions, in the extreme case a single primitive. Since there is only one primitive to compute, all threads other than the first will sit idle. A similar situation arises in the second example. Here, an ERI is calculated over four AOs, each with contraction length 3, which works out to a total of 81 primitive integrals. In this case, none of the 64 threads is completely idle. However, some load imbalance is still present, since the first 17 threads compute a second integral, while the remaining warps, threads 18–31, execute unproductive no-op instructions. It should be noted that threads 32–63 do not perform wasted instructions because the entire warp skips the second integral evaluation. Thus, "idle" CUDA threads do not always map to idle execution units. Finally, as contractions lengthen, load imbalance between threads in a block will become negligible in terms of the runtime, making the 1B1CI strategy increasingly efficient.

4.3.2 One-Thread-One-Contracted Integral

In the 1T1CI strategy, each thread loops over all primitives within a contraction, accumulating the sum in a local register. A schematic representation is shown in Figure 4.2. There the contracted integrals are again represented by cyan squares, but each CUDA block, represented by red outlines, now handles multiple contracted integrals rather than just one. In order to achieve optimal memory performance, the primitive PQs for each AO pair are interleaved so that fetches of primitive *bra* and *ket* pairs from neighboring threads request contiguous addresses. The 2D blocks shown in Figure 4.2 are given dimensions 4×4 for illustrative purposes. In practice, blocks sized at least 16×16 threads should be used. Because threads within the same warp execute in SIMT manner, warp divergence will result whenever neighboring ERIs involve contractions of different lengths. To eliminate these imbalances, the bra and ket PQ arrays must be sorted by contraction length so that blocks handle ERIs of uniform contraction length.

4.3.3 One-Thread-One-Primitive Integral

The 1T1PI strategy is illustrated in Figure 4.3. This approach provides the finest grained parallelism of the mappings we consider. It is similar to the 1B1CI in that contracted ERIs are again broken up between multiple threads. Here, however, the primitives are distributed to CUDA blocks without considering the contraction of which they are members. In Figure 4.3, cyan squares represent 2D CUDA blocks of dimension 16×16, and red lines represent divisions between contracted integrals. Because the block size is not an even multiple of the contraction length, the primitives computed within the same block will, in general, contribute to multiple contracted ERIs. This approach results in perfect load balancing (for primitive evaluation), since each thread does exactly the same amount of work. It is also notable in that 1T1PI imposes few constraints on the ordering of primitive pairs, since they no longer need to be grouped or interleaved by parent AO indices. However, the advantages of the 1T1PI scheme are greatly reduced if we also consider the subsequent reduction step needed to produce the final contracted ERIs. These reductions involve inter-block communication and, for highly contracted basis sets, can prove more expensive than the ERI evaluation itself.

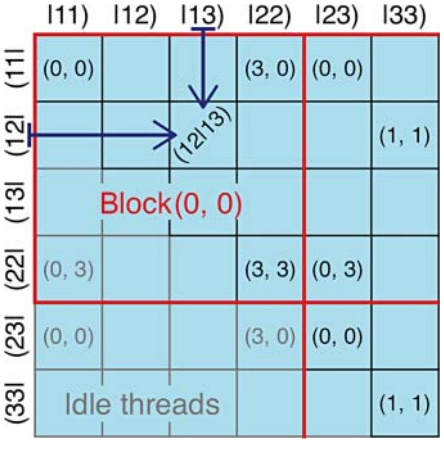


Figure 4.2 *Schematic of one-thread one-contracted Integral (1T1CI) mapping. Cyan squares represent contracted ERIs and CUDA threads. Thread indices are shown in parentheses. Each CUDA block (red outlines) computes 16 ERIs, with each thread accumulating the primitives of an independent contraction, in a local register. (See insert for colour representation of this figure)*

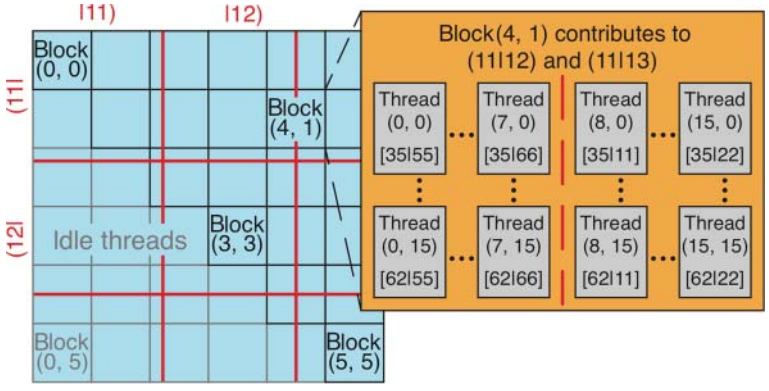


Figure 4.3 *Schematic of one-thread one-primitive integral (1T1PI) mapping. Cyan squares represent two-dimensional tiles of 16 × 16 primitive ERIs, each of which is assigned to a 16 × 16 CUDA block as labeled. Red lines indicate divisions between contracted ERIs. The orange box shows assignment of primitive ERIs to threads (gray squares) within a block that contains contributions to multiple contractions. (See insert for colour representation of this figure)*

4.3.4 Comparison of Contracted ERI Schemes

We now turn to a practical evaluation of the acceleration schemes presented above. We consider the evaluation of ERIs for a system made up of 64 hydrogen atoms arranged in a $4 \times 4 \times 4$ cubic lattice with a separation of 0.74 Å between nearest neighbors. Table 4.1 compares execution times for each method along with timings for GPU-related overheads, such as memory transfers between host and device memories.

Two basis sets are considered. The 6-311G basis represents a low contraction limit in which most (two-thirds) of the AOs include a single primitive component. Here, the 1T1PI mapping provides the best performance. At such a minimal contraction level, very few ERIs must be accumulated between block boundaries, minimizing required inter-block communication. The 1T1CI method is a close second, since for small contractions it represents a parallel decomposition that is only slightly coarser than the ideal 1T1PI scheme. The 1B1CI scheme, on the other hand, is a distant third. This is due to its poor load balancing discussed above. For the 6-311G basis, over 85% of the contracted ERIs involve nine or fewer primitive ERI contributions. Thus, the vast majority of the 64 threads in each block do no work.

It should be noted that simply transferring ERIs between host and device can take longer than the ERI evaluation itself, especially in the case of low basis set contraction. This means that, for efficient

Table 4.1 *Runtime comparison for evaluating ERIs of 64 H atom lattice using 1B1CI, 1T1CI, and 1T1PI methods*

	GPU1B1CI	GPU1T1CI	GPU1T1PI	CPU PQ Pre-calc	GPU–CPU Transfer	GAMESS
6-311G	7.086	0.675	0.428	0.009	0.883	170.8
STO-6G	1.608	1.099	2.863	0.012	0.012	90.6

Times are given in seconds. All GPU calculations were run on an Nvidia 8800 GTX. CPU precalculation records time required to build pair quantities prior to launching GPU kernels. GPU–CPU transfer provides the time required to copy the completed ERIs from device to host memory. Timings for the CPU-based GAMESS program running on an Opteron 175 CPU are included for comparison.

GPU implementations, ERIs can be re-evaluated from scratch faster than they can be fetched from host memory, and much faster than they can be fetched from disk.

The STO-6G basis provides a sharp contrast. Here each contracted ERI includes 6^4 or 1296 primitive ERI contributions. As a result, for the 1T1PI scheme the reduction step becomes much more involved, in fact requiring more time than primitive ERI evaluation itself. This illustrates a key principle that organizing communication is often just as important as minimizing arithmetic instructions when optimizing performance. The 1T1CI scheme performs similar to the 6-311G case. The fact that all ERIs now involve uniform contraction lengths provides a slight boost, since it requires only 60% more time to compute twice as many primitive ERIs compared to the 6-311G basis. The 1B1CI method improves dramatically, as every thread of every block is now saturated with 20 or 21 primitive ERIs.

4.3.5 Extensions to Higher Angular Momentum

A few additional considerations are important for extension to basis functions of higher angular momentum. For nonzero angular momentum functions, shells contain multiple functions. Within the McMurchie–Davidson scheme, all integrals within an ERI shell depend on the same intermediate Hermite integral values $[\Lambda_p|\Lambda_q]$. Thus, it is advantageous to have each thread compute an entire shell of primitive integrals. For example, a thread computing a primitive ERI of class [*sp*|*sp*] is responsible for a total of nine functions.

The performance of GPU kernels is quite sensitive to the memory requirements of each thread. As threads use more memory, the total number of concurrent threads resident on each SM decreases. Fewer active threads, in turn, reduce the GPU's ability to hide execution latencies and lowers throughput performance. Because all threads in a grid reserve the same memory footprint, a single grid handling both low and – more complex – high angular momentum integrals will apply the worst case memory requirements to all threads. To avoid this, separate kernels must be written for each class of integral.

Specialized kernels also provide opportunities to further optimize each routine and reduce memory usage, for example, by unrolling loops or eliminating conditionals. This is particularly important for ERIs involving *d*- and higher angular momentum functions, where loop overheads become nontrivial. For high angular momentum integrals, it is also possible to use symbolic algebra libraries to generate unrolled kernels that are optimized for the GPU [21].

Given a basis set of mixed angular momentum shells, we could naively extend any of the decomposition strategies presented above as follows. First, build the pair quantities as prescribed, without consideration for angular momentum class. Then launch a series of ERI kernels, one for each momentum class, assigning a compute unit (either block or thread depending on strategy being extended) to every ERI in the grid. Work units assigned to ERIs that do not apply to the appropriate momentum class could exit immediately. This strategy is illustrated for a hypothetical system containing four *s*-shells and one *p*-shell in Figure 4.4a. Each square represents a shell quartet of ERIs, that is, all ERIs resulting from the combination of the various angular momentum functions within each of the included AO shells. The elements are colored by total angular momentum class, and a specialized kernel evaluates elements of each color. The problem with this approach is that the number of integral classes increases rapidly with the maximum angular momentum in the system. Thus, the inclusion of *d*-shells would already result in the vast majority of the threads in each kernel exiting without doing any work. A better approach is illustrated in Figure 4.4b. Here, we have sorted the bra and ket pairs by the angular momenta of their constituents: *ss*, then *sp*, and last *pp*. As a result, the ERIs of each class are localized in contiguous subgrids, and kernels can be dimensioned to exactly cover only the relevant integrals.

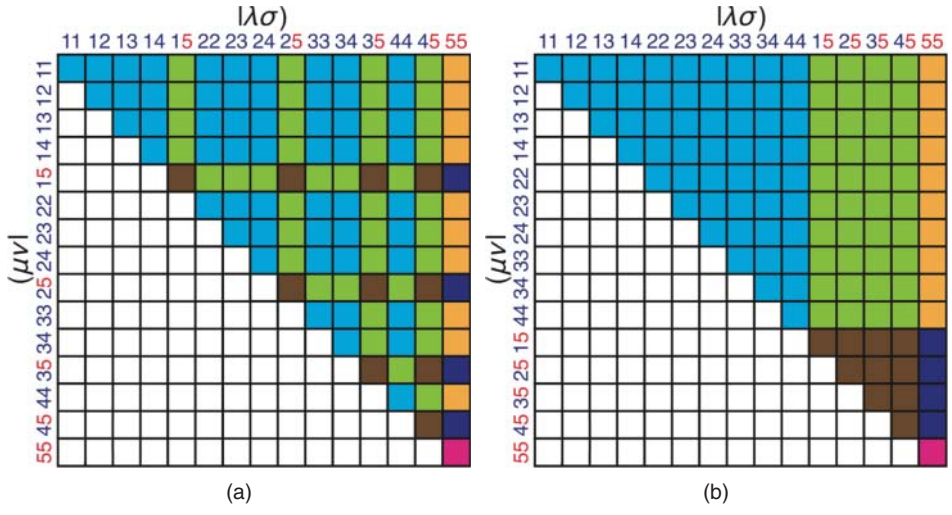


Figure 4.4 *ERI grids colored by angular momentum class for a system containing four* s*-shells and one* p*-shell. Each square represents all ERIs for a shell quartet. (a) Grid when bra and ket pairs are ordered by simple loops over shells. (b) ERI grid for same system with bra and ket pairs sorted by angular momentum,* ss*, then* sp*, then* pp*. Each integral class now handles a contiguous chunk of the total ERI grid. (See insert for colour representation of this figure)*

4.4 Integral-Direct Fock Construction on GPUs

We now consider the contributions of ERIs to the Fock matrix in Eq. (4.17):

$$G_{\mu\nu}(\mathbf{P}) = \sum_{\lambda\sigma}^{N} P_{\lambda\sigma}[2(\mu\nu|\sigma\lambda) - (\mu\lambda|\nu\sigma)]. \tag{4.29}$$

Because the ERIs remain constant from one SCF iteration to the next, it used to be common practice to form **G** from precomputed integrals. However, for systems containing thousands of basis functions, ERI storage quickly becomes impractical. The integral-direct approach, pioneered by Almlof [22], avoids the storage of ERIs by recomputing them on the fly during each formation of the Fock matrix.

Beyond capacity constraints, the direct approach offers performance advantages over conventional algorithms based on integral storage. As observed previously, ERIs can sometimes be recalculated faster than they can be recalled from storage (even when this storage is high-speed random access memory). As advances in instruction throughput continue to outpace those for communication bandwidths, the balance will shift further in favor of integral-direct algorithms. Another advantage results from the knowledge of the density matrix during Fock construction. This allows the direct approach to augment the Schwarz bound with density matrix information

$$|(\mu\nu|\lambda\sigma)P_{\lambda\sigma}| \leq (\mu\nu|\mu\nu)^{1/2}(\lambda\sigma|\lambda\sigma)^{1/2}|P_{\lambda\sigma}|, \tag{4.30}$$

in order to identify and eliminate more insignificant integrals than is possible for precomputed ERIs. In the case of the GPU, additional advantages can be achieved by abandoning the construction of contracted ERIs $(\mu\nu|\lambda\sigma)$ in favor of direct construction of Fock elements $G_{\mu\nu}$ from Gaussian primitive functions. These longer contractions simplify the parallel structure of the algorithm and improve GPU performance.

To minimize the number of integrals that must be evaluated, the symmetry of real-valued ERIs

$$(\mu\nu|\lambda\sigma) = (\lambda\sigma|\mu\nu) = (\nu\mu|\lambda\sigma) = (\nu\mu|\sigma\lambda) \tag{4.31}$$

is often exploited in order to compute only unique integrals where $\mu \leq \nu$, $\lambda \leq \sigma$, and $\mu\nu \leq \lambda\sigma$, where $\mu\nu$ and $\lambda\sigma$ are compound indices corresponding to the element numbers in an upper triangular matrix. Each ERI is then combined with various density elements and scattered into multiple locations in the Fock matrix. This reduces the number of ERIs that must be evaluated by a factor of 8 compared to a naïve implementation. However, gathering inputs from the density matrix introduces irregular memory access patterns, and scattering outputs to the Fock matrix creates dependencies between ERIs. GPU performance is extremely sensitive to these considerations, so that even an eightfold reduction in work is easily swamped by an even larger performance slowdown. It is helpful to start from a naïve, but completely parallel algorithm, and then exploit ERI symmetry only where it provides a practical benefit. To this end, $G_{\mu\nu}$ in Eq. (4.29) is evaluated in separate Coulomb

$$J_{\mu\nu} = \sum_{\lambda\sigma}^{N} (\mu\nu|\sigma\lambda) P_{\lambda\sigma} \tag{4.32}$$

and exchange

$$K_{\mu\nu} = \sum_{\lambda\sigma}^{N} (\mu\lambda|\nu\sigma) P_{\lambda\sigma} \tag{4.33}$$

contributions.

4.4.1 GPU J-Engine

The strategies for handling ERIs developed above provide a good starting point for the evaluation of the Coulomb operator in Eq. (4.32) [23]. The shift from individual ERIs to larger contractions of primitive ERIs

$$J_{\mu\nu} = \sum_{\substack{\chi_i \in \phi_\mu \\ \chi_j \in \phi_\nu}} \sum_{\substack{\lambda \\ \sigma}} \sum_{\substack{\chi_k \in \phi_\lambda \\ \chi_l \in \phi_\sigma}} c^*_{\mu i} c_{\nu j} c^*_{\lambda k} c_{\sigma l} [ij|kl] P_{\lambda\sigma} \tag{4.34}$$

can be visualized by noting that all integrals across any row of the ERI grids discussed above contribute to the same element $J_{\mu\nu}$.

As with simple ERI evaluation, the first step is to enumerate basis shell pairs $\phi_\mu\phi_\nu$ for bra and ket. Because $J_{\mu\nu}$ is symmetric, it is sufficient to compute its upper triangle, and only bra pairs where $\mu \leq \nu$ need to be considered. Also for ket pairs, the terms $(\mu\nu|\lambda\sigma)P_{\lambda\sigma}$ and $(\mu\nu|\sigma\lambda)P_{\sigma\lambda}$ are equal, and both contribute to the same Coulomb element. Thus, symmetry again allows a reduction to ket pairs where $\lambda \leq \sigma$. For a basis including s, p, and d shells, sorting the pairs by angular momentum class results in a total of six pair classes, ss, sp, sd, pp, pd, and dd, and 36 specialized kernels. It is not possible to exploit the final class of ERI symmetry $(\mu\nu|\lambda\sigma) = (\lambda\sigma|\mu\nu)$ without introducing dependencies between rows of the ERI grid and, as a result, performance sapping inter-block communication. Thus, ignoring integrals neglected by screening, the J-Engine nominally computes $N^4/4$ integrals.

Each included shell pair is expanded by iterating over primitive pairs $\{ij|\chi_i \in \phi_\mu, \chi_j \in \phi_\nu\}$ and appending PQ data to bra and ket arrays. Along with the usual $\vec{P}$, η, and K quantities, we also precompute, for the bra, a Schwarz contribution $B^{\text{bra}}_{ij} = (ij|ij)^{1/2}$ and Hermite expansion coefficients $\{E^{[ij]}_p\}$, where

$$[ij| = \sum_p E^{[ij]}_p [\Lambda_p|. \tag{4.35}$$

For ket pairs, a density-weighted Schwarz contribution $B_{ij}^{\text{ket}} = (ij|ij)^{1/2}|P_{ij}|^{\max}$ is stored such that the full density-weighted Schwarz bound can be efficiently evaluated as follows:

$$|(ij|kl)P_{kl}| \leq B_{ij}^{\text{bra}} B_{kl}^{\text{ket}} = (ij|ij)^{1/2}(kl|kl)^{1/2}|P_{kl}|^{\max}. \tag{4.36}$$

Here, the max is taken over all functions in the shell pair. For ket PQs, the corresponding block of the density matrix is also transformed to the Hermite basis and incorporated into the Hermite coefficients to produce the combined $\{D_p^{[ij]}\}$ [24]:

$$|ij]P_{\lambda\sigma} = \sum_p E_p^{[ij]} P_p^{\lambda\sigma} |\Lambda_p] = \sum_p D_p^{[ij]} |\Lambda_p]. \tag{4.37}$$

The generalized outer product of these bra and ket PQ vectors represents all primitive integrals needed to evaluate the Coulomb matrix. This structure is illustrated in the left half of Figure 4.5, where each pixel represents a primitive ERI and is colored according to the magnitude of the Schwarz bound for that integral. Before mapping the grid of primitive ERIs to CUDA compute units, it is important to consider the pattern of ERI magnitudes within the ERI matrix of Figure 4.5. Before evaluating an ERI, its Schwarz bound is formed by the product $B_{ij}^{\text{bra}} B_{kl}^{\text{ket}}$. If the bound falls below the ERI threshold, $\varepsilon \approx 10^{-11}$, the ERI is deemed insignificant and no further evaluation is performed. If the primitives are ordered arbitrarily, as in Figure 4.5a, skipped and evaluated integrals will be interspersed among the threads of each warp. As a result, neglecting integrals offers almost no performance advantage, since all cores in the warp are occupied to compute the non-negligible elements. A solution is to sort the primitive bra and ket arrays for each angular momentum class by decreasing Schwarz contributions $B_{ij}^{\text{bra|ket}}$. This results in the grid shown in Figure 4.5b, where significant integrals are localized in the upper left corner of each grid. Because the bounds $B_{ij}^{\text{bra|ket}}$ rapidly decrease with increasing ij distance, the number of bra and ket pairs grows linearly with the size of the system, and the total number of significant ERIs will be only N^2, a tremendous reduction from the full set of N^4 integrals.

In addition to removing warp divergence when computing significant ERIs, the sorted PQs can also be exploited to completely avoid examining bounds for most negligible integrals. After sorting

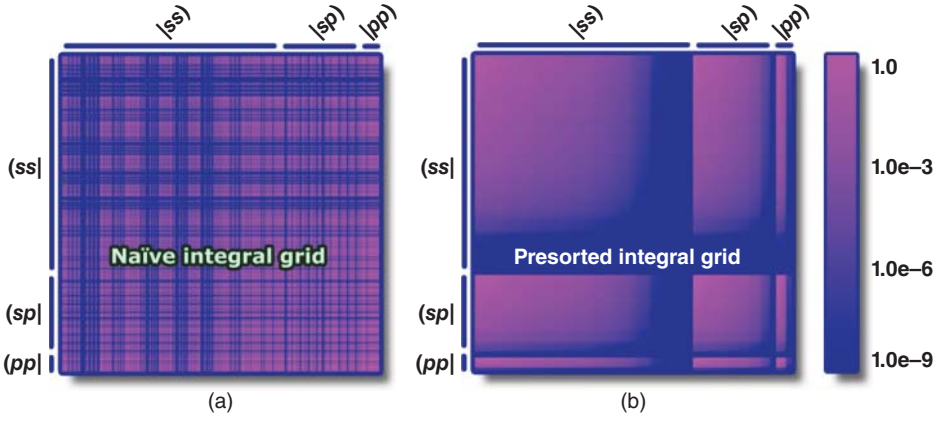


Figure 4.5 *Organization of ERIs for Coulomb formation. Rows and columns correspond to primitive bra and ket pairs, respectively. Each ERI is colored according to the magnitude of its Schwarz bound. Data are derived from calculation on ethane molecule. Figure (a) obtained by arbitrary ordering of pairs within each angular momentum class and suffers from load imbalance because large and small integrals are computed in neighboring cells, and (b) that sorts bra and ket primitives by Schwarz contribution within each momentum class, providing an efficient structure for parallel evaluation (See insert for colour representation of this figure)*

the primitives, the bound is guaranteed to decrease scanning across any row or down any column. As a result, once one negligible integral has been located, all others in that row or column can be skipped as well. To take advantage of this, the algorithm must assign work units to scan rows or columns of the ERI grid. Rows are more convenient, since contributions from all ERIs in a row contribute to a single Coulomb element. This allows individual ERIs to be efficiently calculated from PQs and accumulated in local registers to produce primitive Coulomb contributions as follows:

$$J^{[ij]} = [ij|\sum_{kl}|kl]P_{\lambda\sigma} = \sum_{p} E_p^{[ij]} \sum_{q} D_q^{[kl]}[p|q]. \tag{4.38}$$

Here it is implicitly assumed that $\chi_k \in \phi_\lambda$ and $\chi_l \in \phi_\sigma$. In practice, it is convenient to postpone the final summation over the p index until after primitives are contracted into AOs. Thus, a handful of Coulomb values in the Hermite basis, $\{J_p^{[ij]}\}$, are collected for each ERI row:

$$J_p^{[ij]} = \sum_{q} D_q^{[kl]}[p|q]. \tag{4.39}$$

After the kernel executes, the intermediate values $J_p^{[ij]}$ are copied to the host where a final reduction into Coulomb elements is performed:

$$J_{\mu\nu} = \sum_{p} E_p^{[ij]} \sum_{\substack{\chi_i \in \phi_\mu \\ \chi_j \in \phi_\nu}} J_p^{[ij]}. \tag{4.40}$$

The dimension of the CUDA work units assigned to each row is a matter of optimization. Perhaps the simplest approach would execute 1D blocks, assigning each thread to an entire row in Figure 4.5. This approach limits the total number of threads to the number of bra primitives, which for small systems is insufficient to saturate the thousands of cores on the GPU. Also, because a vertical 1D block spans a wide range of Schwarz ERI bounds, the top thread will encounter its first negligible ERI many columns after the bottom thread. For the intervening span of columns, many threads sit idle. A square block, on the other hand, should minimize divergence. Based on empirical tests, a block dimension of 8×8 was chosen. The eight threads in each row of a block cooperate to compute interleaved ERI elements across a row of the ERI grid. Once all rows in the block find negligible ERIs, block-level reductions are performed to sum final outputs for each row. Figure 4.6 illustrates the algorithm.

4.4.2 GPU K-Engine

The GPU-based construction of the exchange operator in Eq. (4.33) follows a similar approach to that employed in the GPU J-Engine algorithm [23]. Here we highlight the adjustments that are required to accommodate the increased complexity of exchange which results from the split of the output index $\mu\nu$ between the bra and ket. As a result, rows of the bra-by-ket ERI grid do not contribute to a single matrix element but scatter across an entire row of the K matrix. Additionally, symmetry among the ERIs is more difficult to exploit in K since symmetric pairs, for example, $(\mu\lambda|\nu\sigma) \leftrightarrow (\mu\lambda|\sigma\nu)$, now contribute to multiple matrix elements. The split of the density index $\lambda\sigma$ between bra and ket also precludes the pre-contraction of density elements into the ket PQs.

The complications above can be naïvely removed by changing the definitions of bra and ket to the so-called physicist notation, where pairs include a primitive from each of two electrons:

$$(\mu\lambda|\nu\sigma) = \langle\mu\nu|\lambda\sigma\rangle. \tag{4.41}$$

With such $\mu\nu$ and $\lambda\sigma$ pairs, a GPU algorithm analogous to the J-Engine could easily be developed. Unfortunately, the new definitions of bra and ket also affect the pairwise Schwarz bound, which now

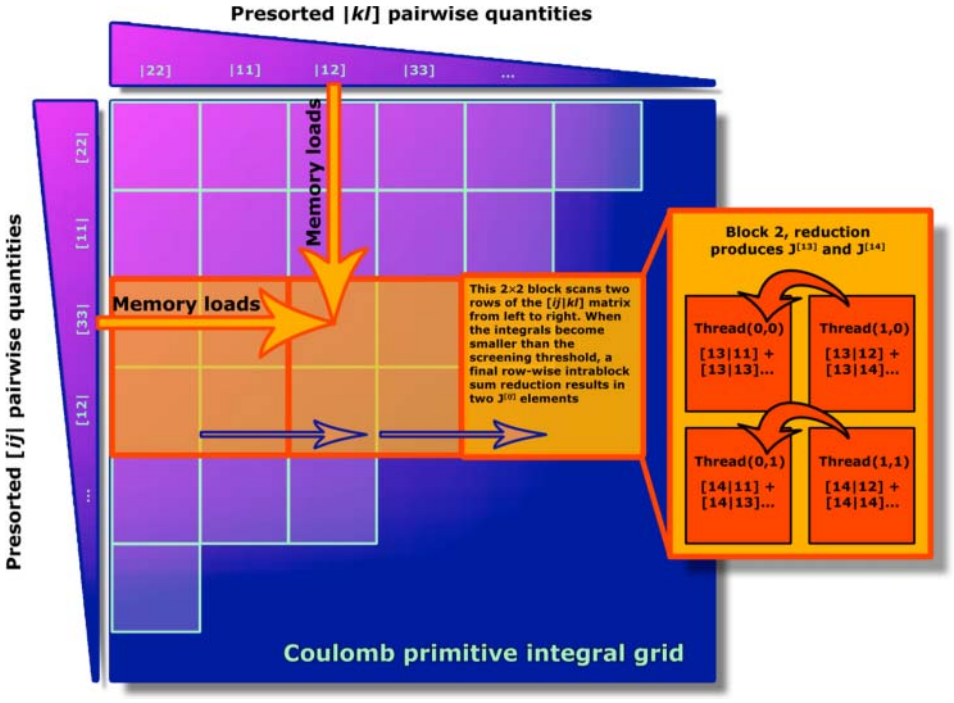


Figure 4.6 *Schematic representation of a J-Engine kernel for one angular momentum class, for example, (*ss|ss*). Cyan squares represent significant ERI contributions. Sorted bra and ket vectors are represented by triangles to the left and above the grid. The path of a 2×2 block as it sweeps across the grid is shown in orange. The final reduction across rows of the block is illustrated within the inset to the right. (See insert for colour representation of this figure)*

becomes the following:

$$\langle \mu\nu|\mu\nu\rangle^{1/2}\langle \lambda\sigma|\lambda\sigma\rangle^{1/2} = (\mu\mu|\nu\nu)^{1/2}(\lambda\lambda|\sigma\sigma)^{1/2}. \tag{4.42}$$

As the distance R between ϕ_μ and ϕ_ν increases, the quantity $(\mu\mu|\nu\nu)$ decays slowly as $1/R$ compared to e^{-R^2} for $(\mu\nu|\mu\nu)$. This weaker bound means that all N^4 ERIs would need to be examined, leading to a severe performance reduction compared to the N^2 Coulomb algorithm. Thus, the scaling advantages of the $\mu\lambda/\nu\sigma$ pairing are very much worth maintaining, even at the cost of reduced hardware efficiency.

The K-Engine begins with the now-usual step of enumerating AO shell pairs $\phi_\mu\phi_\lambda$. Because the $\phi_\mu\phi_\lambda$ and $\phi_\lambda\phi_\mu$ pairs contribute to different matrix elements, symmetry cannot be exploited without introducing dependencies between exchange outputs. Neglecting symmetry, AO pairs are constructed for both $\mu \leq \lambda$ and $\mu > \lambda$ pairs. As with Coulomb evaluation, the pairs are separated by angular momentum class, and different kernels are tuned for each type of integral. Inclusion of $\mu > \lambda$ pairs requires additional pair and kernel classes compared to the J-Engine, since, for example, kernels handling *ps* pairs are distinct from those handling *sp* pairs.

The J-Engine ordering of bra and ket primitive pairs would leave contributions to individual K elements scattered throughout the ERI grid. In order to avoid inter-block communication, it is necessary to localize these exchange contributions. This can be accomplished by sorting the primitive pairs by the μ index, so that the ERIs contributing to each element, $K_{\mu\nu}$, form a contiguous tile in the ERI grid. This is illustrated in Figure 4.7. Within each segment of μ pairs, primitive PQs are additionally

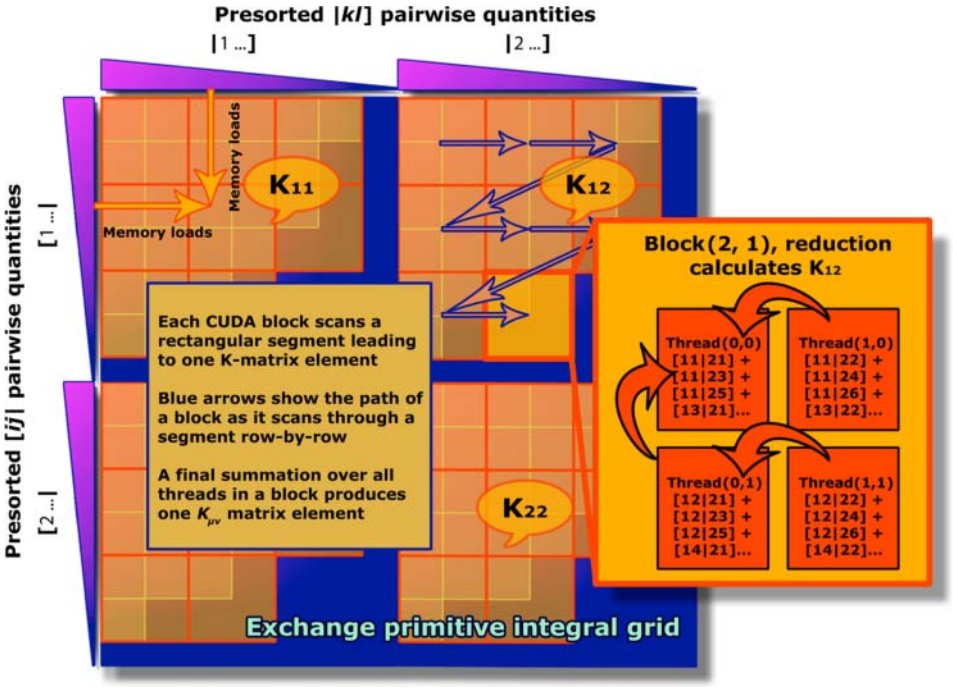


Figure 4.7 *Schematic of a K-Engine kernel. Bra and ket PQ arrays are represented by triangles to the left and above the grid. The pairs are grouped by ν and λ index and then sorted by bound. The paths of four blocks are shown in orange, with the zigzag pattern illustrated by arrows in the top right. The final reduction of an exchange element within a 2 × 2 block is shown to the right. (See insert for colour representation of this figure)*

sorted by Schwarz contributions $(\mu\lambda|\mu\lambda)^{1/2}$, so that significant integrals are concentrated in the top left of each $\mu\nu$ tile.

Since the density index is split between bra and ket, density information cannot be included in the primitive expansion coefficients or Schwarz contributions. Instead, additional vectors are packed for each angular momentum class: P^{ss}, P^{sp}, and so on. The packed density is ordered by shell pair and organized into CUDA vector types to allow for efficient fetching. For example, each entry in the P^{sp} block contains three elements: sp_x, sp_y, and sp_z, packed into a float4 (assuming single precision) vector type. The maximum overall density element in a shell pair, $|P_{\lambda\sigma}|^{\max}$, is also precomputed for each shell pair in order to minimize memory access when computing the exchange Schwarz bound.

$$|(\chi_\mu\chi_\lambda|\chi_\nu\chi_\sigma)P_{\lambda\sigma}| \leq [\chi_\mu\chi_\lambda|\chi_\mu\chi_\lambda]^{1/2}[\chi_\nu\chi_\sigma|\chi_\nu\chi_\sigma]^{1/2}|P_{\lambda\sigma}|^{\max}. \tag{4.43}$$

To map the exchange to the GPU, a 2D CUDA grid is employed in which each block computes a single $K_{\mu\nu}$ element, and is thus assigned a tile of the primitive ERI grid. Each CUDA block passes through its tile of $[\mu\cdots|\nu\cdots]$ primitive integrals in a zig-zag pattern, computing one primitive shell per thread. Ordering pairs by bound allows a block to skip to the next row as soon as the boundary of insignificant integrals is located. As with the J-Engine, the uniformity of ERI bounds within each block is maximized by dimensioning square 2D CUDA blocks. Figure 4.7 shows a 2×2 block for illustrative purposes, but a block dimension of at least 8×8 would be used in practice. When all significant ERIs have been evaluated, a block-level reduction is used to compute the final exchange matrix element. This reduction represents the only inter-thread communication required by the K-Engine

algorithm and detracts negligibly from the overall performance. The K-Engine approach is similar to the 1B1CI ERI scheme above. However, because each block is responsible for thousands of primitive ERIs, exchange evaluation does not suffer from the load imbalance observed for the 1B1CI algorithm.

The structure of the ERI grid allows neighboring threads to fetch contiguous elements of bra and ket PQ data. Access to the density matrix is more erratic, because it is not possible to order pairs within each $\mu\nu$ tile simultaneously by the Schwarz bound and the λ or σ index. As a result, neighboring threads must issue independent density fetches, which are in principle scattered across the density matrix. In practice, once screening is applied, most significant contributions to an element $K_{\mu\nu}$ arise from localized $\lambda\sigma$ blocks, where λ is near μ and σ is near ν. Thus, CUDA textures can be used to ameliorate the performance penalty imposed by the GPU on nonsequential density access.

Another difficulty related to the irregular access of density elements is that the density-weighted Schwarz bounds for primitive ERIs in Eq. (4.43) are not strictly decreasing across each row or down each column of a $\mu\nu$ tile. As a result, the boundary between significant and negligible ERIs is not as sharply defined as in the Coulomb case. Yet, using the density to preempt integral evaluation in exchange is critical. This can be appreciated by comparison with the Coulomb operator, in which the density couples only to the ket pair. Since, in general, both the density element $P_{\lambda\sigma}$ and the Schwarz bound $(\lambda\sigma|\lambda\sigma)^{1/2}$ decrease as the distance $r_{\lambda\sigma}$ increases, density-weighting the Coulomb Schwarz bound has the effect of making the already small $(\lambda\sigma|\lambda\sigma)^{1/2}$ terms even smaller. In exchange, on the other hand, the density couples the bra and ket so that small density matrix elements can reduce otherwise large bounds and greatly reduce the number of ERIs that need to be evaluated. In fact, for large insulator systems, we expect the total number of non-negligible ERIs to reduce from N^2 Coulomb integrals to N exchange ERIs.

In order to incorporate the density into the ERI termination condition, the usual exit threshold ε is augmented by an additional guard multiplier $G = \sim 10^{-5}$. Each warp of 32 threads then terminates ERI evaluation across each row of the ERI tile when it reaches a contiguous set of primitive ERIs, where the following holds for every thread:

$$[\chi_\mu\chi_\lambda|\chi_\mu\chi_\lambda]^{1/2}[\chi_\nu\chi_\sigma|\chi_\nu\chi_\sigma]^{1/2}|P_{\lambda\sigma}|^{\max} < G\varepsilon. \tag{4.44}$$

In principle, this nonrigorous exit condition could neglect significant integrals in worst case scenarios. However, in practice, empirical tests demonstrate that the exit procedure produces the same result obtained when density information is not exploited and results in a considerable performance improvement.

For SCF calculations, the density and exchange matrices are symmetric. In this case, we need only compute the upper triangle of the matrix elements. This amounts to exploiting $(\mu\nu|\lambda\sigma) \leftrightarrow (\lambda\sigma|\mu\nu)$ ERI symmetry, and means that we nominally calculate $N^2/2$ ERIs. This is 4 times more ERIs than are generated by traditional CPU codes that take full advantage of the eightfold ERI symmetry. However, comparisons at the level of FLOPs are too simplistic when analyzing the performance of massively parallel architectures. In the present case, freeing the code of inter-block dependencies boosts GPU performance more than enough to compensate for a fourfold work increase.

For insulating systems, the AO density matrix elements $P_{\mu\nu}$ will rapidly decay to zero with increasing distance $r_{\mu\nu}$. This can be exploited to prescreen the exchange elements $K_{\mu\nu}$, for which the sum in Eq. (4.33) will be small. Perhaps the simplest approach is to impose a simple distance cutoff so that $K_{\mu\nu}$ is approximated as follows:

$$K_{\mu\nu} = \begin{cases} \sum_{\lambda\sigma}^{N} (\mu\lambda|\nu\sigma) P_{\lambda\sigma} & \text{if } r_{\mu\nu} < R^{\text{MASK}} \\ 0 & \text{otherwise} \end{cases}. \tag{4.45}$$

This is effective because, if μ is far from ν, then either λ is far from σ and $P_{\lambda\sigma}$ is zero, or one of the $\mu\lambda$ or $\nu\sigma$ bounds will be zero. The mask condition is trivially precomputed for each element

$K_{\mu\nu}$ and packed in a bit mask, 1 indicating that the exchange element should be evaluated and 0 indicating that it should be skipped. Each block of CUDA threads checks its mask at the beginning of the exchange kernel, and blocks that are assigned a zero bit exit immediately. As will be shown for practical calculations in the following, this distance mask greatly reduces the number of ERIs evaluated by the K-Engine.

A potential problem with the simple distance mask is that R^{MASK} is basis set and system-dependent. For example, the range at which the density matrix decays depends on the energy gap between the highest occupied and the lowest unoccupied molecular orbitals (HOMO and LUMO). Also, diffuse functions in the basis sets will increase the distance at which the $\mu\lambda$-Schwarz bound becomes negligible.

These limitations can be overcome by employing the Schwarz bound to produce a rigorous bound on the full exchange matrix elements as follows:

$$K_{\mu\nu} = \sum_{\lambda\sigma}^{N} (\mu\lambda|\nu\sigma) P_{\lambda\sigma} \leq \sum_{\lambda\sigma}^{N} (\mu\lambda|\mu\lambda)^{1/2} P_{\lambda\sigma} (\nu\sigma|\nu\sigma)^{1/2}. \tag{4.46}$$

Casting the AO Schwarz bounds as a matrix **Q**

$$Q_{\mu\nu} = (\mu\nu|\mu\nu)^{1/2} \tag{4.47}$$

the following rigorous bound on the density matrix can be derived [25]:

$$K_{\mu\nu} \leq (\mathbf{QPQ})_{\mu\nu}. \tag{4.48}$$

For large systems, the matrix products can easily be computed in sparse algebra. Then, a bit mask is constructed to neglect all matrix elements $K_{\mu\nu}$ for which $(\mathbf{QPQ})_{\mu\nu} < \tau^{\mathrm{MASK}}$ and the kernels are called just as in the distance mask case.

4.4.3 Exchange–Correlation Integration

After ERIs, the evaluation of the exchange–correlation potential represents a second major bottleneck for DFT calculations. For clarity we limit our consideration here to the acceleration of GGA-type functionals using GPUs. Our implementation is very similar to that reported by Yasuda [26]. In the general spin-polarized case, where $\rho_\alpha \neq \rho_\beta$, Eqs. (4.13) and (4.21) for the exchange–correlation energy and potential become the following:

$$E_{\mathrm{XC}} = \int f_{xc}(\rho_\alpha(\vec{r}), \rho_\beta(\vec{r}), \gamma_{\alpha\alpha}(\vec{r}), \gamma_{\alpha\beta}(\vec{r}), \gamma_{\beta\beta}(\vec{r})) d^3\vec{r}, \tag{4.49}$$

$$V_{\mu\nu}^{\mathrm{XC}\alpha} = \int \left[\frac{\partial f_{\mathrm{xc}}}{\partial \rho_\alpha} \phi_\mu \phi_\nu + \left(2 \frac{\partial f_{\mathrm{xc}}}{\partial \gamma_{\alpha\alpha}} \nabla \rho_\alpha + \frac{\partial f_{\mathrm{xc}}}{\partial \gamma_{\alpha\beta}} \nabla \rho_\beta \right) \cdot \nabla(\phi_\mu \phi_\nu) \right] d^3\vec{r}, \tag{4.50}$$

where an equation analogous to Eq. (4.50) is used for $V_{\mu\nu}^{\mathrm{XC}\beta}$, and γ represents the gradient invariants as follows:

$$\gamma_{\sigma\sigma'}(\vec{r}) = \nabla \rho_\sigma(\vec{r}) \cdot \nabla \rho_{\sigma'}(\vec{r}). \tag{4.51}$$

Typical XC kernel functions f_{xc} are not amenable to analytical integration; thus, instead, a quadrature grid is employed to evaluate Eqs (4.49) and (4.50). Because molecular potentials exhibit discontinuous cusps near each nucleus, molecular quadrature grids are typically constructed by superposing spherical grids centered at each atom. One common and efficient approach combines the Euler–Maclaurin radial and Lebedev angular quadratures for each atomic grid. Thus, integration

around an atom centered at $\vec{a}$ might be performed as follows, where E_i and L_j represent radial and angular weights respectively, p is a combined ij index, and $\lambda_p = E_i L_j$:

$$\int f(\vec{r}) d^3\vec{r} \approx \sum_i \sum_j E_i L_j f(\vec{a} + \vec{r}_{ij}) = \sum_p \lambda_p f(\vec{a} + \vec{r}_p). \tag{4.52}$$

Each atomic grid independently integrates over all $\mathbb{R}^3$ but is most accurate in the region of its own nucleus. In forming molecular grids, quadrature points from different atomic grids must be weighted to normalize the total sum over all atoms and to ensure that each atomic quadrature dominates in the region around its nucleus. An elegant scheme introduced by Becke [27] accomplishes this by defining a spatial function $w_a(\vec{r})$ for each atomic quadrature a, which gives the weight assigned to that quadrature in the region of $\vec{r}$. All double counting is avoided by constraining the weight function so that

$$\sum_a w_a(\vec{r}) = 1 \tag{4.53}$$

holds for all $\vec{r}$. Also, constructing $w_a(\vec{r})$ so that it is near unity in the region of the ath nucleus causes each atomic quadrature to dominate in its most accurate region. The final quadrature is then evaluated as follows:

$$\int f(\vec{r}) d^3\vec{r} \approx \sum_a \sum_p w_a(\vec{R}_a + \vec{r}_p) \lambda_p f(\vec{R}_a + \vec{r}_p) = \sum_a \sum_p w_{ap} \lambda_p f(\vec{r}_{ap}). \tag{4.54}$$

The calculation of the Becke quadrature weights w_{ap}, itself a computationally involved task, is performed through the following equations:

$$\begin{aligned}
w_{np} &= \frac{P_n(\vec{r}_{np})}{\sum_m P_m(\vec{r}_{np})}, \\
P_a(\vec{r}) &= \prod_{b \neq a} s(\mu_{ab}), \\
\mu_{ab}(\vec{r}) &= \frac{|\vec{R}_a - \vec{r}| - |\vec{R}_b - \vec{r}|}{|\vec{R}_a - \vec{R}_b|} = \frac{\vec{r}_a - \vec{r}_b}{\vec{R}_{ab}}, \\
s(\mu) &= \frac{1}{2}(1 - g(\mu)), \\
g(\mu) &= p(p(p(\mu))), \\
p(\mu) &= \frac{3}{2}\mu - \frac{1}{2}\mu^3.
\end{aligned} \tag{4.55}$$

As with ERIs, the problem is easily parallelized, since the weight for each one of the $\sim 10^6$ grid points is independent of all the others. Thus, the grid weights are easily accelerated on the GPU, with a single CUDA thread assigned to each grid point. Because $0 \leq s(\mu) \leq 1$, the evaluation of each $P_m(\vec{r}_{np})$ can be terminated as soon as the running product becomes negligibly close to zero. Furthermore, if the numerator $P_n(\vec{r}_{np})$ is zero, then the denominator need not even be evaluated. These early exit conditions greatly accelerate the algorithm, but also introduce warp divergence on the GPU because neighboring threads break out of loops at different iterations. Organizing the grid points into spatial bins and sorting by atomic quadrature index ensures that neighboring threads follow similar execution paths. The resulting weights for many quadrature points are negligible, and these are removed from

the grid. Finally, the Becke and spherical weights are multiplied and the atomic labels are dropped to form the final set of quadrature points and weights:

$$\begin{aligned} w_{ap}\lambda_p &\rightarrow \omega_i, \\ \vec{r}_{ap} &\rightarrow \vec{r}_i. \end{aligned} \tag{4.56}$$

Once the grid has been established, the density and its gradient at each grid point are needed in order to evaluate the XC kernel. For an AO alpha-spin density matrix, these are evaluated as follows:

$$\rho_\alpha(\vec{r}) = \sum_{\mu,\nu} P^\alpha_{\mu\nu}\phi_\mu(\vec{r})\phi_\nu(\vec{r}), \tag{4.57}$$

$$\nabla\rho_\alpha(\vec{r}) = \sum_{\mu,\nu} P^\alpha_{\mu\nu}\phi_\mu(\vec{r})\nabla\phi_\nu(\vec{r}). \tag{4.58}$$

As a result of the exponential decay of the basis functions ϕ_μ, the sum for each grid point $\vec{r}$ need only include a few significant AOs. These are efficiently identified by spatially binning the significant primitive pair distributions, which are determined as

$$\{\chi_i\chi_j | \varepsilon_{ao} < \max_{\vec{r}} \chi_i(\vec{r})\chi_j(\vec{r})P^{\max}_{\mu\nu}, \chi_i \in \phi_\mu, \chi_i \in \phi_\nu\}, \tag{4.59}$$

where ε_{ao} is some small threshold. The primitive lists and quadrature points are then copied to the GPU, where each thread computes the sums of Eqs. (4.57) and (4.58) for a single grid point. The primitive pairs are sorted by increasing exponent within each spatial bin. As a result, the entire bin may be skipped after the first negligible point-primitive pair combination has been reached. As in the evaluation of quadrature weights, the spatial binning of grid points minimizes warp divergence by ensuring that neighboring threads share identical sets of significant basis functions. The same procedure is repeated for the beta density in spin-polarized calculations.

Next, the density at each quadrature point is used to evaluate the XC-kernel function and its various derivatives on the grid. Thus the following values are computed for each grid point:

$$\begin{aligned} a_i &= \omega_i f_{\text{xc}}(\rho_\alpha(\vec{r}_i), \rho_\beta(\vec{r}_i), \gamma_{\alpha\alpha}(\vec{r}_i), \gamma_{\alpha\beta}(\vec{r}_i), \gamma_{\beta\beta}(\vec{r}_i)), \\ b_i &= \omega_i \frac{\partial f_{\text{xc}}(\vec{r}_i)}{\partial \rho_\alpha}, \\ c_i &= \omega_i \left(2\frac{\partial f_{\text{xc}}(\vec{r}_i)}{\partial \gamma_{\alpha\alpha}}\nabla\rho_\alpha + \frac{\partial f_{xc}(\vec{r}_i)}{\partial \gamma_{\alpha\beta}}\nabla\rho_\beta \right). \end{aligned} \tag{4.60}$$

This step has a small computational cost and can be performed on the host without degrading performance. This is desirable as a programming convenience because implementing various density functionals can be performed without editing CUDA kernels and because the host provides robust and efficient support for various transcendental functions needed by some density functionals. During this step, the kernel values a_i are also summed to evaluate the total exchange–correlation energy per Eq. (4.49).

The final step is to construct the AO matrix elements for the exchange–correlation potential. This task is again performed on the GPU. For all non-negligible primitive pair distributions $\{\chi_i\chi_j | \varepsilon_{ao} < \max_{\vec{r}} \chi_i(\vec{r})\chi_j(\vec{r}), \chi_i \in \phi_\mu, \chi_j \in \phi_\nu\}$, the pair quantities K_{ij}, $\vec{P}_{ij}$, and η_{ij} from Eq. (4.22) are packed into arrays and transferred to device memory. The grid point positions $\vec{r}_i$ are spatially binned and

transferred to the GPU with corresponding values b_i and c_i. A single CUDA kernel then performs the summation

$$\begin{aligned}\widetilde{V}_{\mu\nu} &= \sum_i \phi_\mu(\vec{r}_i)\left(\frac{b_i}{2}\phi_\nu\left(\vec{r}_i\right) + c_i\nabla\phi_\nu(\vec{r}_i)\right) \\ &= \sum_i \sum_{k\in\mu} \sum_{l\in\nu} c_{\mu k}c_{\nu l}\chi_k(\vec{r}_i)\left(\frac{b_i}{2}\chi_l\left(\vec{r}_i\right) + c_i\nabla\chi_l(\vec{r}_i)\right)\end{aligned} \tag{4.61}$$

and the final matrix elements are computed as follows:

$$V_{\mu\nu}^{\mathrm{XC}} = \widetilde{V}_{\mu\nu} + \widetilde{V}_{\nu\mu}. \tag{4.62}$$

The calculation is configured on a 2D matrix in which each row corresponds to a primitive pair $\chi_k\chi_l$, and each column is associated with a single grid point. The CUDA grid is configured as a 1D grid of 2D blocks, similar to the GPU J-Engine described earlier. A stack of blocks spanning the primitive pairs sweeps across the matrix. Because the points are spatially binned, entire bins can be skipped whenever a negligible point-primitive pair combination is encountered. As in the Coulomb algorithm, each row of threads across a CUDA block cooperates to produce a matrix element in the primitive basis $\widetilde{V}_{kl}$. These are copied to the host where the final contraction into AO matrix elements takes place.

4.5 Precision Considerations

ERI values span a wide dynamic range from $\sim 10^3$ Ha all the way [28] down to $\sim 10^{-10}$, below which ERIs can be neglected while maintaining chemical accuracy. In order to represent values across the resulting range of 13 decimal orders, 64-bit double-precision floating-point arithmetic is required. With the advent of hardware-based double-precision CPU instructions in the 1980s, it became standard practice to use double precision uniformly for all floating-point operations in quantum chemistry calculations. In fact, for various architectural and algorithmic reasons, single-precision arithmetic offered only a slight performance advantage and was, thus, largely abandoned in quantum chemistry programs. However, as a result of their pedigree in graphics and multimedia applications, for which reduced precision is adequate, GPU designs have emphasized single-precision performance. In fact, the earliest GPGPU codes for quantum chemistry had to make do with minimal (or even no) double-precision support on the device [10, 12, 20, 28]. Modern architectures, such as Nvidia's recent Kepler GPUs, continue to provide a significant performance advantage of at least 3.5× to single-precision operations. With the development of vectorized instruction sets, even CPUs now offer single precision a 2:1 performance advantage over double precision. Beyond operation throughput, single-precision operands halve the memory footprint and bandwidth requirements of a kernel. This is extremely important for massively parallel architectures where the large number of in-flight threads makes on-chip memories premium resources. As a result, it is advantageous to design algorithms that perform as many operations as possible in single precision [29, 30]. In the bandwidth- and instruction-limited cases, this would provide up to a twofold speedup. Of course, care must be exercised to ensure that adequate precision is provided to guarantee robust and correct results.

In practice, all finite precision operations introduce some error. Certain operations, such as taking differences between nearly equal values or accumulating tiny elements into much larger sums, greatly amplify this error. Consider the following sum between two decimal numbers, each represented with seven decimal digits of precision.

	1	.	3	6	7	8	0	2	$\times 10^3$
+	4	.	7	2	2	1	5	8	$\times 10^{-4}$

In order to carry out this addition, the processor must first shift the numbers to a common exponent (for simplicity, assume intermediate operations are carried out without loss of precision).

	1	.	3	6	7	8	0	2	0	0	0	0	0	0	0	$\times 10^3$
+	0	.	0	0	0	0	0	0	4	7	2	2	1	5	8	$\times 10^3$
	1	.	3	6	7	8	0	2	4	7	2	2	1	5	8	$\times 10^3$

The intermediate sum is not seven-digit-representable. Truncation to the nearest representable number results in the removal of the gray digits and produces a final result (in black) that is identical to the first operand. An algorithm requiring many such additions would accumulate significant errors.

Although there are algorithms to increase the precision of summation without using higher precision at all [31], the most robust solution to this accumulation problem is to perform the addition in higher precision. The same operation at 14-digit precision might look like the following after shifting to a common exponent:

	1	.	3	6	7	8	0	2	1	6	7	0	2	5	1	0	0	0	0	0	0	$\times 10^3$
+	0	.	0	0	0	0	0	0	4	7	2	2	1	5	8	3	9	2	4	8	2	3	$\times 10^3$
	1	.	3	6	7	8	0	2	6	3	9	2	4	0	9	3	9	2	4	8	2	3	$\times 10^3$

The addition no longer results in disastrous error. However, because the final result is still truncated to the 14 digits in black, the digits in gray have no impact on the operation. Notably, the same result would have been obtained if the smaller number had used its 7-digit rather than 14-digit representation. This can be important if the operands considered here were themselves produced through many prior operations.

This situation is exactly obtained when ERIs are accumulated into the Coulomb and exchange matrices. The vast majority of these ERIs are orders of magnitude smaller than the few largest values. These small ERIs, readily identified by Schwarz bound, can be computed in single precision and accumulated into the Fock matrix with a single double-precision ADD operation. A separate set of double-precision ERI kernels are then used to compute large ERI contributions [30]. Using bound-sorted pair lists for Coulomb and exchange kernels automatically segregates the double- and single-precision ERIs. This is shown for the Coulomb case in Figure 4.8 and results in minimized warp divergence within both single- and double-precision kernels:

$$J_p^{[ij]} = \sum_q \begin{cases} \left(E_p^{[ij]} D_q^{[kl]} [\Lambda_p|\Lambda_q]\right)^{\{32\}} & \text{if } \sqrt{[ij|ij][kl|kl]}|P_{kl}|^{\max} < \tau^{\text{prec}} \\ \left(E_p^{[ij]} D_q^{[kl]} [\Lambda_p|\Lambda_q]\right)^{\{64\}} & \text{if } \sqrt{[ij|ij][kl|kl]}|P_{kl}|^{\max} \geq \tau^{\text{prec}} \end{cases}. \tag{4.63}$$

In splitting the work, it is important that bounds be deterministic and identical when evaluated in double- and single-precision kernels to avoid skipping or double counting ERIs near the boundary. This is not entirely trivial for inexact finite-precision operations. For example, neither single-nor

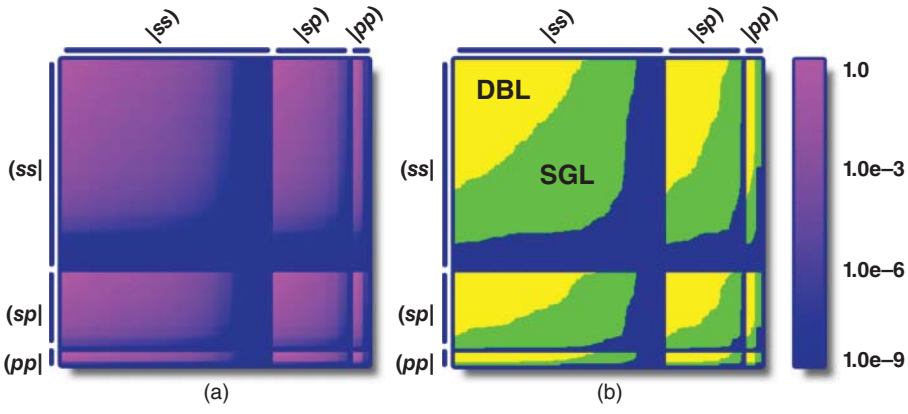


Figure 4.8 *Organization of double- and single-precision workloads within Coulomb ERI grids. As in Figure 4.5, rows and columns correspond to primitive bra and ket pairs. (a) Each ERI is colored according to the magnitude of its Schwarz bound. (b) ERIs are colored by required precision. Yellow ERIs require double precision, while those in green may be evaluated in single precision. Blue ERIs are neglected entirely. (See insert for colour representation of this figure)*

double-precision products are exactly commutative, so $a \cdot b \neq b \cdot a$. Thus, when evaluating the Schwarz bound, it is important to maintain the same level of precision and order of operations.

Next, it must be determined how large τ^{prec} can be set while maintaining sufficient precision. This is best answered by empirical study [30]. Figure 4.9 shows results for some representative test systems treated with a range of precision thresholds. The relative errors in final RHF energies averaged over the test systems are shown for various basis sets and values of τ^{prec}. Mixed precision calculations provide an effective precision that is intermediate between double and single precision. The error is very well behaved on the log–linear plot so that a safe empirical bound is readily constructed. This is shown as the black line in Figure 4.9, whose equation is as follows:

$$\text{Error}(\tau) < 2.0 \times 10^{-6} \tau^{0.7}. \tag{4.64}$$

By inverting Eq. (4.64), an appropriate precision can be selected for any desired relative error in the final energy. This allows a safe level of precision to be selected at the start of the SCF without pessimistically requiring that every operation be performed in slower double-precision arithmetic. It is further possible to vary the precision level during the course of an SCF calculation. This dynamic precision approach is suggested by viewing the SCF as an iterative correction procedure that improves an initially approximate density at each step. At each iteration, τ^{prec} must be chosen so that errors resulting from finite precision do not overwhelm whatever other errors exist in the density. In this way, the precision error can be reduced at the same rate as the density matrix converges ultimately to the point where full double-precision quality is obtained. The error present in the density matrix at each iteration can be approximated from the convergence criteria. A good choice is the maximum element of the commutator, **SPF–FPS**, where **F** and **P** are the Fock and density matrices and **S** is the AO overlap matrix.

Table 4.2 summarizes the efficiency of the dynamic precision approach. In all cases the error is well controlled such that final energies match full double-precision results to within the convergence threshold of 10^{-5} Ha. Also, the use of reduced precision in early SCF iterations has no discernable effect on the number of SCF iterations required to reach convergence. In terms of performance, the dynamic precision approach consistently exceeds a 2× speedup over double precision. This is noteworthy since, in terms of theoretical peak performance, the Tesla C2050 GPU used for these tests

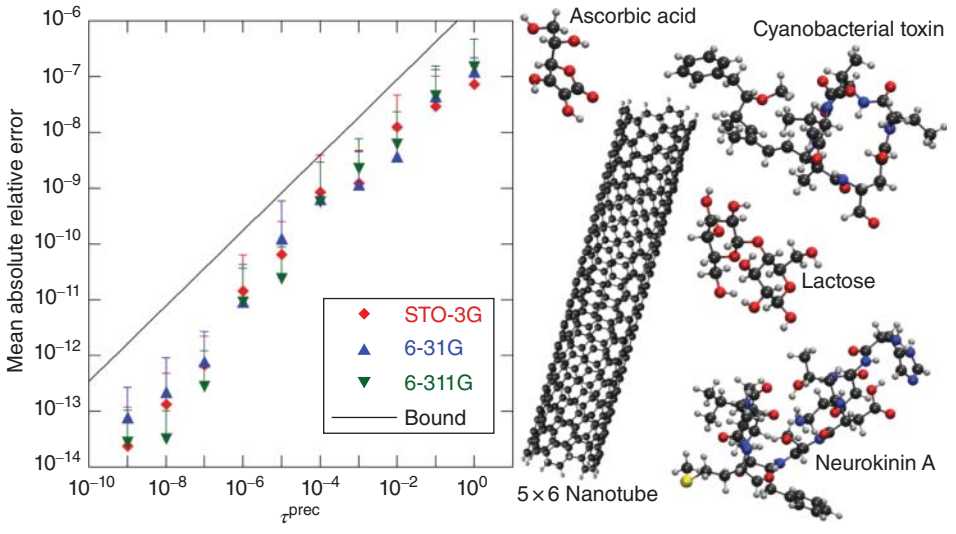


Figure 4.9 *Relative error in final energies versus precision threshold τ^{prec} for various basis sets. Test molecules are shown on the right. Each point is averaged over the five test systems. Error bars represent 2 standard deviations above the mean. The black line shows the empirical error bound of Eq. (4.64)*

Table 4.2 *Dynamic precision performance for RHF calculations on various molecules using the 6-31G basis set*

System	Atoms	BFS	GPUs	Time (s)	Iters	Final energy	$\lvert\Delta E\rvert$	Speedup
Ascorbic acid	20	124	1	1.47	12	−680.6986414386	1.2E−07	2.0
Lactose	45	251	1	6.83	10	−1290.0883461502	1.4E−07	2.2
Cyano toxin	110	598	1	66.23	13	−2492.3971987780	4.9E−07	2.3
Neurokinin A	157	864	1	146.75	14	−4091.3672646454	9.7E−08	2.3
5×6 Nanotube	386	3320	8	1138.49	15	−13793.7293926019	1.1E−07	2.7
Crambin	642	3597	8	741.23	12	−17996.6562927786	2.3E−07	1.8
Ubiquitin	1231	6680	8	7418.08	18	−29616.4426380779	4.5E−07	2.0
T-cadherin EC1	1542	8404	8	10592.68	16	−36975.6726052678	3.4E−07	2.0

Columns list number of atoms in each system, number of AO orbitals in basis, number of Tesla C2050 GPUs employed in each calculation, time in seconds for entire SCF energy evaluation using dynamic precision, number of iterations required to converge the maximum matrix element of **SPF–FPS** to 10^{-5} a.u., final dynamic precision energies in Hartree, absolute difference between dynamic and double precision final energies in Hartree, and the total speedup for the SCF calculation running dynamic precision over double precision.

offers only a 2:1 advantage for single precision. Theoretical peaks are only one factor in determining overall performance. As noted above, single-precision intermediates also require smaller register footprints than double-precision values. This allows the compiler to produce more efficient code for single precision kernels and accounts for much of the advantage enjoyed by single-precision ERI kernels.

4.6 Post-SCF Methods

Although the GPU J- and K-Engines were developed above in the context of SCF acceleration, these algorithms can also be applied more generally to calculate properties and implement post-SCF methods. Extension to Coulomb and exchange gradient integrals, for example, is straightforward and

follows the same principles developed in detail above [32]. As a more interesting example, we consider a GPU-accelerated implementation of the configuration interaction singles (CIS) and TDDFT algorithms [33]. Full reviews of these methods can be found elsewhere [34]. The basic working equations for TDDFT can be written in the following matrix notation:

$$\begin{pmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{B} & \mathbf{A} \end{pmatrix} \begin{pmatrix} \mathbf{X} \\ \mathbf{Y} \end{pmatrix} = -\omega \begin{pmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{-1} \end{pmatrix} \begin{pmatrix} \mathbf{X} \\ \mathbf{Y} \end{pmatrix}. \tag{4.65}$$

Here, $\mathbf{X}$ and $\mathbf{Y}$ are vectors representing transition amplitudes, and ω is the excitation energy. Application of Eq. (4.65) for Hartree–Fock theory results in the time-dependent Hartree–Fock (TDHF) method, where the operators A and B are defined as follows:

$$A_{ai,bj} = \delta_{ij}\delta_{ab}(\varepsilon_a - \varepsilon_i) + (\theta_i\theta_a|\theta_j\theta_b) - (\theta_i\theta_j|\theta_a\theta_b), \tag{4.66}$$

$$B_{ai,bj} = (\theta_i\theta_a|\theta_b\theta_j) - (\theta_i\theta_b|\theta_a\theta_j). \tag{4.67}$$

For TDDFT, the operators are defined similarly, but the Coulomb-like ERI is replaced by a functional derivative of the exchange correlation potential:

$$A_{ai,bj} = \delta_{ij}\delta_{ab}(\varepsilon_a - \varepsilon_i) + (\theta_i\theta_a|\theta_j\theta_b) + (\theta_i\theta_j|f_{\mathrm{xc}}|\theta_a\theta_b), \tag{4.68}$$

$$B_{ai,bj} = (\theta_i\theta_a|\theta_b\theta_j) + (\theta_i\theta_b|f_{\mathrm{xc}}|\theta_a\theta_j), \tag{4.69}$$

$$(\theta_i\theta_j|f_{\mathrm{xc}}|\theta_a\theta_b) = \iint \theta_i(\vec{r}_1)\theta_j(\vec{r}_1)\frac{\delta^2 E_{\mathrm{xc}}}{\delta\rho(\vec{r}_1)\delta\rho(\vec{r}_2)}\theta_a(\vec{r}_2)\theta_b(\vec{r}_2). \tag{4.70}$$

In the above equations, ERIs are written in terms of molecular orbitals that result from a prior SCF calculation. Subscripts i and j refer to occupied orbitals, while a and b index virtual orbitals. Spin is neglected for clarity. Using the Tamm–Dancoff approximation (TDA), the B matrix is set to zero. This results either in CIS for TDHF or in TDA–TDDFT for TDDFT:

$$\mathbf{AX} = \omega\mathbf{X}. \tag{4.71}$$

This equation is efficiently solved using iterative diagonalization algorithms such as Davidson's method [35]. Such algorithms solve the eigenvalue problem through a series of matrix vector products between A and trial eigenvectors:

$$(\mathbf{AX})_{bj} = \sum_{ia}[\delta_{ij}\delta_{ab}(\varepsilon_a - \varepsilon_i) + (\theta_i\theta_a|\theta_j\theta_b) - (\theta_i\theta_j|\theta_a\theta_b)]X_{ia}. \tag{4.72}$$

In order to apply the J- and K-Engine algorithms, these products can be carried out in the AO basis using the following transformations:

$$\sum_{ia}[(\theta_i\theta_a|\theta_j\theta_b) - (\theta_i\theta_j|\theta_a\theta_b)]X_{ia} = \sum_{\mu\nu} C_{\mu j}C_{\nu b}F_{\mu\nu}, \tag{4.73}$$

$$F_{\mu\nu} = \sum_{\lambda\sigma} T_{\lambda\sigma}[(\mu\nu|\lambda\sigma) - (\mu\lambda|\nu\sigma)], \tag{4.74}$$

$$T_{\lambda\sigma} = \sum_{ia} X_{ia}C_{\lambda i}C_{\sigma a}. \tag{4.75}$$

Here, $\mathbf{T}$ is a nonsymmetric transition density expressed in the AO basis. The dominant computational step is the construction of $\mathbf{F}$, which is analogous to computing two-electron contributions to the

SCF Fock matrix. Because the matrix **T** is nonsymmetric, two small adjustments to the algorithms developed above are required. For the J-Engine, it is sufficient to symmetrize the transition density:

$$\widetilde{T}_{\mu\nu} = \frac{1}{2}(T_{\mu\nu} + T_{\nu\mu}). \tag{4.76}$$

However, for a nonsymmetric density, the exchange matrix itself becomes nonsymmetric. As a result, the K-Engine must be called twice to handle both the upper and lower triangles of **F**. Thus, ignoring screening, the excited-state exchange contribution nominally calculates all N^4 ERIs. However, the transition density $T_{\lambda\sigma}$ contains only a single electron and is much more sparse than the total ground-state density. This allows the K-Engine to neglect many more ERIs for the excited state and ultimately allows the excited state K-build to outperform its ground-state analog.

4.7 Example Calculations

Here we illustrate the practical application of the GPU algorithms discussed previously in real-world calculations. All GPU calculations described in the following were carried out using the TeraChem quantum chemistry package. To elucidate the computational scaling of our GPU implementation with increasing system size, we consider two types of systems: First, linear alkene chains ranging in length between 25 and 707 carbon atoms provide a benchmark at the low-dimensional limit. Second, cubic water clusters containing between 10 and 988 molecules provide a dense three-dimensional test system, which is more representative of most condensed phase systems of interest. Examples of each system type are shown in Figure 4.10.

For each system, a restricted B3LYP calculation was performed on a single Tesla M2090 GPU using the 6-31G basis set. Figure 4.11 shows the timing breakdown during the first SCF iteration for the J-Engine, K-Engine, linear algebra (LA), and DFT exchange–correlation potential. For the water clusters, the K-Engine and distance-masked K-Engines were tested in separate runs, and are both provided for comparison. A conservative screening distance of 8 Å was chosen for the distance-masked K-Engine. Exponents resulting from a power fit of each series are provided in the legend.

The linear algebra time is dominated by the diagonalization of the Fock matrix, which scales visibly worse than the Fock formation routines. For large systems, diagonalization will clearly

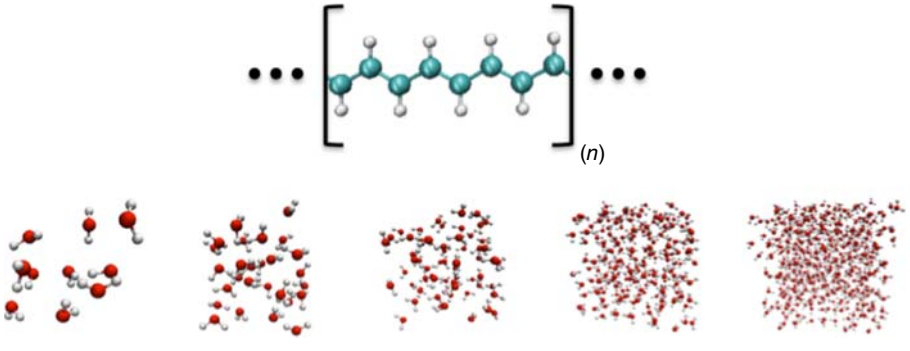


Figure 4.10 *One-dimensional alkene and three-dimensional water-cube test systems. Alkene lengths vary from 24 to 706 carbon atoms and water cubes range from 10 to nearly 850 water molecules. A uniform density is used for all water boxes*

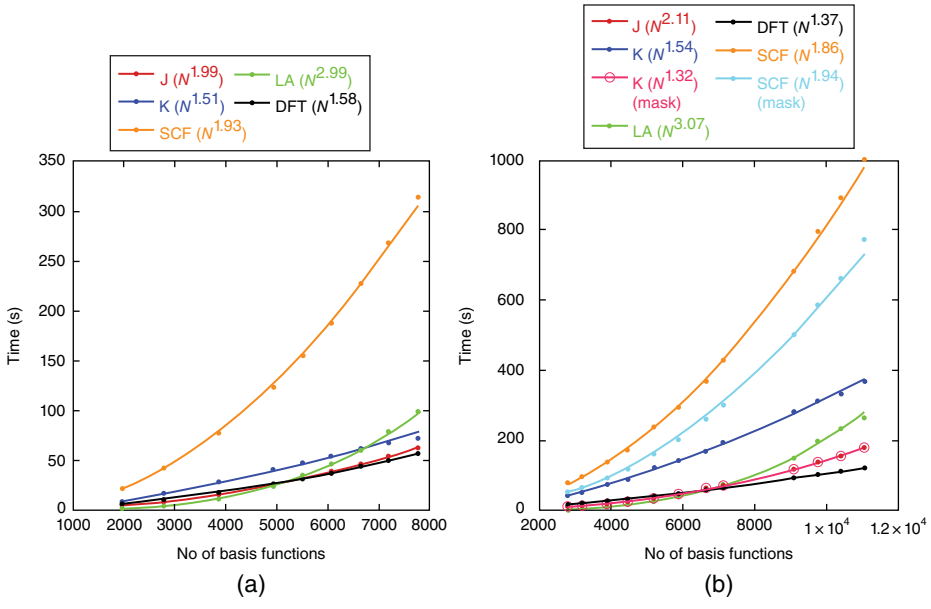


Figure 4.11 *First SCF iteration timings in seconds for (a) linear alkenes and (b) cubic water clusters. Total times are further broken down into J-Engine, K-Engine, distance-masked K-Engine, linear algebra (LA), and DFT exchange–correlation contributions. For water clusters, total SCF times are shown for both the naïve and distance-masked (mask) K-Engine. All calculations were performed using a single Tesla M2090 GPU and the 6-31G basis set. Power fits show scaling with increasing system size, and the exponent for each fit is provided in the legend. (See insert for colour representation of this figure)*

become the principal bottleneck. This is especially true for low-dimensional systems, where the Fock formation is particularly efficient due to the sparsity of the density matrix. However, for the more general 3D clusters, linear algebra remains a small contribution to the total time past 10,000 basis functions. If the more efficient masked K-Engine can be employed, the crossover point, at which linear algebra becomes dominant, shifts to about 8000 basis functions.

Although the time per basis function is lower for the alkene test series, the overall scaling is consistent with the water box calculations. This behavior is good evidence that we have penetrated the asymptotic regime of large systems, where the dimensionality of the physical system should impact the prefactor rather than the exponent. That we easily reach this crossover point on GPUs is itself noteworthy. Even more noteworthy is the fact that the K-Engine exhibits sub-quadratic scaling without imposing any of the assumptions or bookkeeping mechanisms such as neighbor lists common among linear scaling exchange algorithms [2, 5, 36, 37]. The further imposition of a simple distance mask on exchange elements provides an effective linear scaling approach for insulating systems. The scaling of the SCF with the masked K-Engine further exemplifies the impact of the linear algebra computation on the overall efficiency. The N^3 scaling of LA becomes significantly more dominant in large systems as J- and K-Engines become more efficient. The validity of the distance-based exchange screening approximation was confirmed by comparing the final converged absolute SCF energies of each water cluster calculated with and without screening. The absolute total electronic energy is accurate to better than 0.1 mHartree in all systems considered, which is well within the accuracy required for chemical purposes.

Our GPU J-Engine scales quadratically with system size, which compares well with our scaling analysis above. Further acceleration would require modification of the long-range Coulomb potential,

for example, by factorization into a multipole expansion [38]. Extrapolating beyond Figure 4.11, this may become necessary for water clusters beyond 12,000 basis functions, when the J-Engine finally becomes the dominant component in Fock formation. As for DFT, the GPU exchange–correlation methods exhibit linear scaling, matching the scaling efficiency that has been achieved in CPU codes but, as demonstrated below, with much greater performance.

Parallelizing Fock construction over multiple GPUs is trivially accomplished by splitting the CUDA grids into equal chunks in the *y*-dimension. For small systems, this strategy is inefficient because it doubles the latency involved in transferring data between the host and device and in launching GPU kernels. For large systems, such latencies become negligible compared to the kernel execution times, and this splitting strategy becomes increasingly efficient, as shown in Figure 4.12.

The scaling efficiency of a code provides a useful check on the quality of the algorithm and its implementation. However, the absolute performance is of much greater importance for practical applications. To assess the GPU's overall usefulness for quantum chemistry, we again use our water box test cases treated at the B3LYP and 6-31G level of theory. Rather than a single GPU, we now employ four cards, across two GPU architectures: the Tesla M2090 and newer GTX Titan. Notably, a 1533-atom single-point energy calculation on a cubic water cluster required only 2.47 h for the entire SCF process, and *ab initio* dynamics, requiring thousands of single-point evaluations of multiple excited states, are feasible up to at least 2876 basis functions comprising the nanostar dendrimer discussed below.

As a point of comparison, the same calculations were carried out using the CPU implementation available in the GAMESS program. Parameters such as integral neglect thresholds and SCF convergence criteria were matched as closely as possible to the previous GPU calculations. The CPU calculation was also parallelized over all eight CPU cores available in a dual Xeon X5680 3.33 GHz server. Figure 4.13 shows the speedup of the GPU implementation relative to GAMESS for the first SCF iteration including both Fock construction and diagonalization, the latter of which is performed in both codes on the CPU. Performance is similar between the two GPU models on small structures

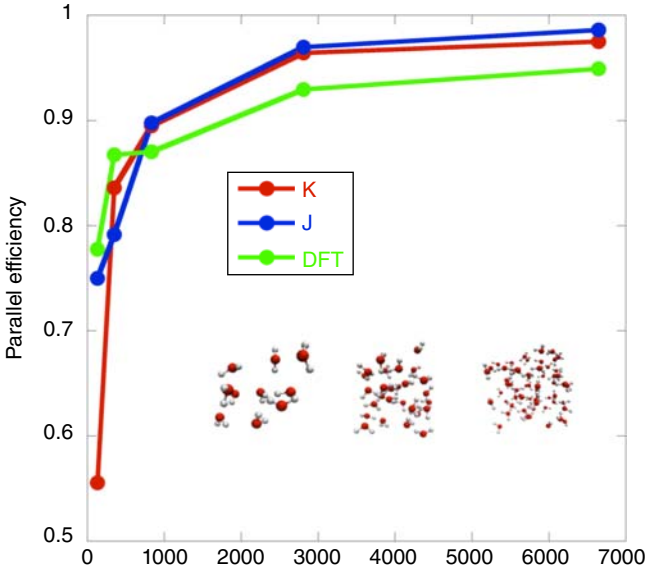


Figure 4.12 *Multi-GPU parallel efficiency for J-Engine, K-Engine, and exchange–correlation Fock formation based on first iteration time for water clusters, run on 2 M2090 GPUs*

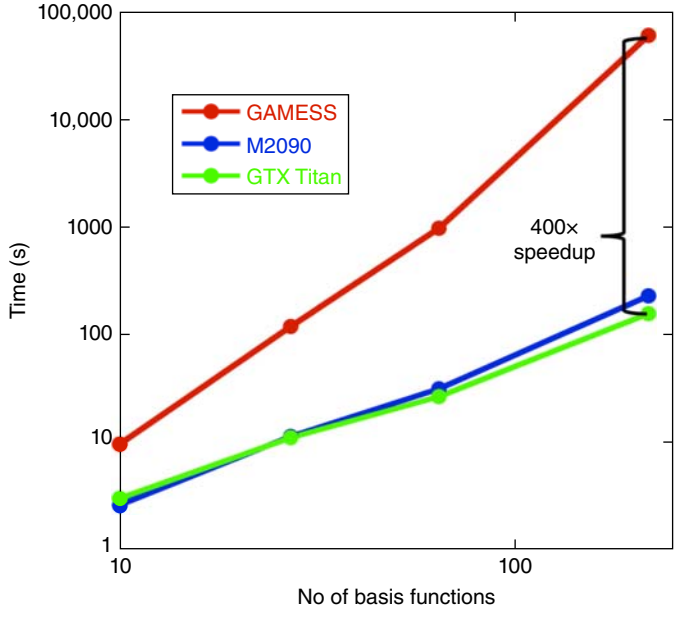


Figure 4.13 *Total SCF time of TeraChem on eight CPUs and four GPUs, relative to GAMESS on eight CPUs for water clusters*

for both alkenes and water clusters. However, GTX Titan exhibits a markedly larger speedup above 100 orbitals and reaches 400× at the largest systems examined. During Fock construction, the CPU cores are left idle in our GPU implementation. It is also possible to reserve some work for the CPU [39], but given the performance advantage of the GPU implementation, this would offer only a very small performance improvement.

Dynamical and excited-state properties of large molecules both in gas and condensed phases are of interest in a wide range of scientific fields. Furthermore, theoretical studies of large chromophores in protein and solvent environments are necessary to understand complex reaction mechanisms and macroscopic behavior of many biological systems. GPU quantum chemistry methods are poised to make quantitative studies of such systems accessible for the first time. Here, the absorption spectrum of a large light-harvesting phenylacetylene dendrimer is calculated using 300 K, ground-state *ab initio* molecular dynamics (AIMD), using ground-state DFT, for conformational sampling and linear response TDDFT for calculation of single-point excitation energies. Calculations were conducted using a range-corrected hybrid exchange–correlation functional (ωPBEh) with a 6-31G basis set. This dendrimer, named the nanostar, was first synthesized by Moore and coworkers [40], and the optical properties were characterized therein. However, this is the first time the optical absorption spectrum has been calculated for the full macromolecule over the relevant excitation energy range. To converge the detailed structure of the optical absorption spectrum shown in Figure 4.14, single-point TDDFT excitations including the 20 lowest excited states were sampled every 20 fs across a 50-ps MD trajectory. This resulted in approximately 2500 individual excited-state calculations on top of 250,000 ground-state calculations constituting the time steps of the MD trajectory.

Comparison to experimental spectra shows excellent agreement in the relative positions of the two high-energy absorption peaks and absolute excitation peak maxima within 1.0 eV of experiments. The lowest energy peak, at ~2.70 eV experimentally (calculated at 3.28 eV), is attributed to excitation centered on the perylene core. The next higher energy peaks are attributed to the conjugated

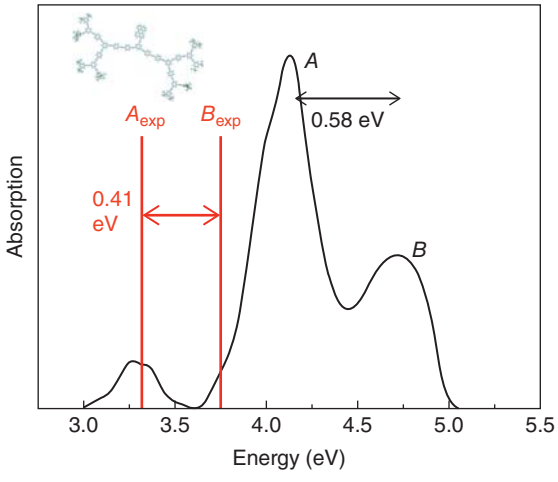


Figure 4.14 *Absorption spectrum of the nanostar dendrimer with peripheral chromophore absorption peaks calculated (A, B) in vacuum and measured experimentally (A_{exp}, B_{exp}) at 300 K in hexane*

branch segments, differing in energy due to different degrees of excitonic delocalization associated with branch lengths. The two peaks labeled A, B in the spectrum are compared to the experimental measurement at 300 K. The absolute energies between the corresponding calculated and experimental absorption peaks differ by 0.7 eV (a consequence of imperfections in the exchange–correlation function used), and the A–B energy differences stand in excellent agreement, indicating that energy gaps between singly excited states are relatively well described within TDDFT. These results show the potential of GPUs to enable new pathways to the prediction and discovery of optical properties in macromolecules.

4.8 Conclusions and Outlook

Massively parallel architectures provide tremendous performance for electronic structure and AIMD calculations using atom-centered Gaussian basis sets. As shown above, speedups of several orders of magnitude are possible for large systems. This is surprising in light of the fact that, in terms of peak theoretical performance, GPUs enjoy a mere 4–8-fold performance advantage over recent CPUs. Clearly, many CPU ERI codes achieve only a fraction of peak performance. Correlated electronic structure methods, such as coupled cluster or RI-MP2, provide a good point of contrast. These methods are usually formulated in terms of linear algebra routines, which have received extensive assembly-level optimization and achieve a high percentage of peak performance on modern CPUs. Thus when ported to GPUs, the speedups are quite modest, on the order of 4–6× [41–46].

What is most important about GPU speedups may not be the raw performance but the fact that this performance is achieved from codes written in high-level languages. Given the complexity of ERI evaluation, it is unlikely that integral codes will ever gain the benefit of nonportable, hand-optimized assembly. This limits CPU ERI codes to a fraction of the total theoretical performance. On the other hand, the GPU is able to achieve a much higher portion of peak performance despite the fact that our ERI routines are expressed essentially in C. Perhaps the future will see CPU hardware adjusting to this new paradigm for streaming processors and offering much improved performance for many scientific applications. Indeed, we believe the availability of the CUDA framework for CPUs would be a significant step toward this improved performance.

GPUs enable *ab initio* calculations to be performed on systems with hundreds of atoms as a matter of routine, enabling first-principles MD on entire proteins [47, 48] and other materials of interest. As a result, DFT calculations are beginning to invade applications that were feasible only for semiempirical or even empirical force fields only a few years ago. The GPU acceleration can be combined with many other acceleration techniques, such as multiple time scale integration in AIMD [49], to further increase the molecular sizes and time scales which are computationally feasible.

Moving beyond SCF calculations, electronic structure codes have traditionally abandoned the atomic orbital basis with its inefficient ERI evaluation, depending instead on highly efficient linear algebra routines. Highly efficient integral codes, however, open the door to improved AO-direct methods, which will greatly increase the size of systems that can be treated with correlated methods.

References

1. Challacombe, M. and Schwegler, E. (1997) Linear scaling computation of the Fock matrix. *Journal of Chemical Physics*, **106** (13), 5526–5536.
2. Burant, J.C., Scuseria, G.E., and Frisch, M.J. (1996) A linear scaling method for Hartree–Fock exchange calculations of large molecules. *Journal of Chemical Physics*, **105** (19), 8969–8972.
3. Schwegler, E. and Challacombe, M. (1996) Linear scaling computation of the Hartree–Fock exchange matrix. *Journal of Chemical Physics*, **105** (7), 2726–2734.
4. Schwegler, E., Challacombe, M., and HeadGordon, M. (1997) Linear scaling computation of the Fock matrix. 2. Rigorous bounds on exchange integrals and incremental Fock build. *Journal of Chemical Physics*, **106** (23), 9708–9717.
5. Ochsenfeld, C., White, C.A., and Head-Gordon, M. (1998) Linear and sublinear scaling formation of Hartree–Fock-type exchange matrices. *Journal of Chemical Physics*, **109** (5), 1663–1669.
6. Daw, M.S. (1993) Model for energetics of solids based on the density-matrix. *Physical Review B*, **47** (16), 10895–10898.
7. Li, X.P., Nunes, R.W., and Vanderbilt, D. (1993) Density-matrix electronic-structure method with linear system-size scaling. *Physical Review B*, **47** (16), 10891–10894.
8. Rubensson, E.H., Rudberg, E., and Salek, P. (2008) Density matrix purification with rigorous error control. *Journal of Chemical Physics*, **128** (7), 074106
9. Rys, J., Dupuis, M., and King, H.F. (1983) Computation of electron repulsion integrals using the Rys quadrature method. *Journal of Computational Chemistry*, **4** (2), 154–157.
10. Yasuda, K. (2008) Two-electron integral evaluation on the graphics processor unit. *Journal of Computational Chemistry*, **29** (3), 334–342.
11. Asadchev, A., Allada, V., Felder, J. *et al.* (2010) Uncontracted Rys quadrature implementation of up to G functions on graphical processing units. *Journal of Chemical Theory and Computation*, **6** (3), 696–704.
12. Ufimtsev, I.S. and Martinez, T.J. (2008) Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation. *Journal of Chemical Theory and Computation*, **4** (2), 222–231.
13. McMurchie, L.E. and Davidson, E.R. (1978) One-electron and 2-electron integrals over Cartesian Gaussian functions. *Journal of Computational Chemistry*, **26** (2), 218–231.
14. Parr, R.G. and Yang, W. (1989) *Density-Functional Theory of Atoms and Molecules*, Oxford University Press, Oxford.
15. Szabo, A. and Ostlund, N.S. (1982) *Modern Quantum Chemistry*, McGraw Hill, New York.
16. Helgaker, T., Jørgensen, P., and Olsen, J. (2000) *Molecular Electronic-Structure Theory*, Wiley, New York.
17. Boys, S.F. (1950) Electronic wave functions. 1. A general method of calculation for the stationary states of any molecular system. *Proceedings of the Royal Society of London A*, **200** (1063), 542–554.

18. Whitten, J.L. (1973) Coulombic potential-energy integrals and approximations. *Journal of Chemical Physics*, **58** (10), 4496–4501.
19. NVIDIA (2013) *CUDA C Programming Guide*. In *Design Guide* [Online] NVIDIA Corporation, `docs.nvidia.com` (accessed 7 March 2014).
20. Ufimtsev, I.S. and Martinez, T.J. (2008) Graphical processing units for quantum chemistry. *Computing in Science and Engineering*, **10** (6), 26–34.
21. Titov, A.V., Ufimtsev, I., Martinez, T., and Dunning, T.H. (2010) Calculating molecular integrals of d and higher angular momentum functions on GPUs. *Abstracts of Papers of the American Chemical Society*, **240**.
22. Almlof, J., Faegri, K., and Korsell, K. (1982) Principles for a direct SCF approach to LCAO-MO ab initio calculations. *Journal of Computational Chemistry*, **3** (3), 385–399.
23. Ufimtsev, I.S. and Martinez, T.J. (2009) Quantum chemistry on graphical processing units. 2. Direct self-consistent-field implementation. *Journal of Chemical Theory and Computation*, **5** (4), 1004–1015.
24. Ahmadi, G.R. and Almlof, J. (1995) The Coulomb operator in a Gaussian product basis. *Chemical Physics Letters*, **246** (4–5), 364–370.
25. Kussmann, J. and Ochsenfeld, C. (2013) Pre-selective screening for matrix elements in linear-scaling exact exchange calculations. *Journal of Chemical Physics*, **138** (13), 134114
26. Yasuda, K. (2008) Accelerating density functional calculations with graphics processing unit. *Journal of Chemical Theory and Computation*, **4** (8), 1230–1236.
27. Becke, A.D. (1988) A multicenter numerical-integration scheme for polyatomic-molecules. *Journal of Chemical Physics*, **88** (4), 2547–2553.
28. Levine, B. and Martinez, T.J. (2003) Hijacking the playstation2 for computational chemistry. *Abstracts of Papers of the American Chemical Society*, **226** (U426).
29. Asadchev, A. and Gordon, M.S. (2012) Mixed-precision evaluation of two-electron integrals by Rys quadrature. *Computer Physics Communications*, **183** (8), 1563–1567.
30. Luehr, N., Ufimtsev, I.S., and Martinez, T.J. (2011) Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs). *Journal of Chemical Theory and Computation*, **7** (4), 949–954.
31. Kahan, W. (1965) Further remarks on reducing truncation errors. *Communications of the ACM*, **8** (1), 40.
32. Ufimtsev, I.S. and Martinez, T.J. (2009) Quantum chemistry on graphical processing units. 3. Analytical energy gradients, geometry optimization, and first principles molecular dynamics. *Journal of Chemical Theory and Computation*, **5** (10), 2619–2628.
33. Isborn, C.M., Luehr, N., Ufimtsev, I.S., and Martinez, T.J. (2011) Excited-state electronic structure with configuration interaction singles and Tamm–Dancoff time-dependent density functional theory on graphical processing units. *Journal of Chemical Theory and Computation*, **7** (6), 1814–1823.
34. Dreuw, A. and Head-Gordon, M. (2005) Single-reference ab initio methods for the calculation of excited states of large molecules. *Chemical Reviews*, **105** (11), 4009–4037.
35. Davidson, E.R. (1975) Iterative calculation of a few of lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *Journal of Computational Physics*, **17** (1), 87–94.
36. Schwegler, E. and Challacombe, M. (1999) Linear scaling computation of the Fock matrix. IV. Multipole accelerated formation of the exchange matrix. *Journal of Chemical Physics*, **111** (14), 6223–6229.
37. Neese, F., Wennmohs, F., Hansen, A., and Becker, U. (2009) Efficient, approximate and parallel Hartree–Fock and hybrid DFT calculations. A 'chain-of-spheres' algorithm for the Hartree–Fock exchange. *Chemical Physics*, **356** (1–3), 98–109.
38. White, C.A. and Head-Gordon, M. (1994) Derivation and efficient implementation of the fast multipole method. *Journal of Chemical Physics*, **101** (8), 6593–6605.

39. Asadchev, A. and Gordon, M.S. (2012) New multithreaded hybrid CPU/GPU approach to Hartree–Fock. *Journal of Chemical Theory and Computation*, **8** (11), 4166–4176.
40. Devadoss, C., Bharathi, P., and Moore, J.S. (1996) Energy transfer in dendritic macromolecules: Molecular size effects and the role of an energy gradient. *Journal of the American Chemical Society*, **118** (40), 9635–9644.
41. Olivares-Amaya, R., Watson, M.A., Edgar, R.G. *et al.* (2010) Accelerating correlated quantum chemistry calculations using graphical processing units and a mixed precision matrix multiplication library. *Journal of Chemical Theory and Computation*, **6** (1), 135–144.
42. Vogt, L., Olivares-Amaya, R., Kermes, S. *et al.* (2008) Accelerating resolution-of-the-identity second-order Moller–Plesset quantum chemistry calculations with graphical processing units. *Journal of Physical Chemistry A*, **112** (10), 2049–2057.
43. DePrince, A.E. and Hammond, J.R. (2011) Coupled cluster theory on graphics processing units I. The coupled cluster doubles method. *Journal of Chemical Theory and Computation*, **7** (5), 1287–1295.
44. Bhaskaran-Nair, K., Ma, W.J., Krishnamoorthy, S. *et al.* (2013) Noniterative multireference coupled cluster methods on heterogeneous CPU–GPU systems. *Journal of Chemical Theory and Computation*, **9** (4), 1949–1957.
45. DePrince, A.E., Kennedy, M.R., Sumpter, B.G., and Sherrill, D.C. (2014) Density-fitted singles and doubles coupled cluster on graphics processing units. *Molecular Physics*, **112**, 844–852.
46. Ma, W.J., Krishnamoorthy, S., Villa, O., and Kowalski, K. (2011) GPU-based implementations of the noniterative regularized-CCSD(T) corrections: Applications to strongly correlated systems. *Journal of Chemical Theory and Computation*, **7** (5), 1316–1327.
47. Kulik, H.J., Luehr, N., Ufimtsev, I.S., and Martinez, T.J. (2012) Ab Initio quantum chemistry for protein structures. *Journal of Physical Chemistry B*, **116** (41), 12501–12509.
48. Ufimtsev, I.S., Luehr, N., and Martinez, T.J. (2011) Charge transfer and polarization in solvated proteins from ab initio molecular dynamics. *Journal of Physical Chemistry Letters*, **2** (14), 1789–1793.
49. Luehr, N., Markland, T.E., and Martinez, T.J. (2014) Multiple time step integrators in *ab initio* molecular dynamics. *Journal of Chemical Physics*, **140**, 084116.

5

GPU Acceleration for Density Functional Theory with Slater-Type Orbitals

Hans van Schoot[1] *and Lucas Visscher*[2]

[1]*Scientific Computing & Modeling NV, Theoretical Chemistry, Vrije Universiteit, Amsterdam, The Netherlands*
[2]*Amsterdam Center for Multiscale Modeling (ACMM), Theoretical Chemistry, VU University Amsterdam, Amsterdam, The Netherlands*

In this chapter, we describe the GPU acceleration of density functional theory (DFT) calculations with Slater-type orbital (STOs) as developed for the Amsterdam Density Functional (ADF) program. This implementation is focused on accelerating the numerical integration step in ADF, which consumes the majority of CPU time in the existing implementation.

5.1 Background

DFT has become the workhorse of quantum chemistry and is widely used to optimize molecular structures, investigate reaction energies and barriers, and calculate molecular response properties [1–3]. Most implementations of DFT developed for molecular systems use atom-centered basis functions, typically with Gaussian-type orbitals (GTOs), as these facilitate the evaluation of Coulomb interactions (see Chapter 4). The ability to analytically calculate two-electron integrals thereby outweighs disadvantages of the Gaussian orbital approach such as their wrong decay at long distance and slow convergence with respect to representation of the nuclear cusp [4]. An interesting alternative is to use STOs, which decay exponentially with the distance from the nucleus and mirror more closely the exact electronic wave function. This leads to a faster convergence with basis set size

Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

and, therefore, the possibility to use smaller basis sets [5]. The well-known disadvantage of using STO basis sets lies in the evaluation of multicenter two-electron integrals for which no analytical expressions are available. These integrals are therefore to be avoided by using density fitting and numerical integration techniques, placing more emphasis on the accuracy of the fitting approach and the employed integration grid. The latter is, however, also employed to evaluate matrix elements of the exchange-correlation (XC) potential, a function that is typically more difficult to integrate than the relatively smooth Coulomb potential. Compared to a GTO implementation, the STO approach is therefore computationally competitive for nonhybrid functionals because the bulk of the work is related to numerical integration of the XC functional and (for large systems) diagonalization of the Fock matrix, which are steps that are common to both approaches.

In this chapter, we will focus on the ADF [6–8] GPU implementation for numerical integration, although the discussed algorithm should also be applicable for the evaluation of the XC energy expression and derivatives thereof in programs with Gaussian basis sets. In this step, we find an almost linear scaling of the computational cost with the number of grid points, which in turn depends on the desired numerical accuracy in the calculation. All calculations are always done in double precision, as single-precision integration generates too much numerical noise to be of use in the employed integration scheme.

The ADF program is written in Fortran and has been under continuous development since the 1970s. This has resulted in a very large number of features, adaptation to parallel architectures using MPI, and well-optimized CPU code for most of the computationally intensive parts of the calculations. This makes ADF a difficult target for GPU acceleration because the code is clearly too large to write a GPU implementation from scratch and, moreover, does not allow porting just a few key routines to the GPU. With this in mind, we decided to focus on accelerating the evaluation of the Coulomb and exchange-correlation Fock matrix elements by moving the numerical integration to the GPU in a hybrid computing scheme that allows GPUs and CPUs to work together. We thereby need to target multiple numerical integration routines that specialize in, for example, evaluating the Fock matrix in the self-consistent field (SCF) cycles or in the iterative solution of the time-dependent density functional theory (TDDFT) equations. These routines are all slightly different but can utilize a similar hybrid computing design.

5.2 Theory and CPU Implementation

In this section, we will first summarize the DFT and SCF approach that is used in ADF, before discussing in more detail the CPU implementation, which was the starting point for the GPU adaptation. An introduction to DFT and SCF procedures in the quantum chemical context can be found in Chapter 3 or standard textbooks (see, e.g., [9] or [10]) The original implementation of the linear scaling SCF method in ADF is described in detail elsewhere [11] and has since then remained largely intact while allowing for accommodating various new developments such as analytical frequencies [12] and various molecular properties [13].

5.2.1 Numerical Quadrature of the Fock Matrix

The key algorithm that is used in all calculations is the determination of molecular orbitals and the electron density matrix using an SCF procedure. For nonhybrid DFT calculations, superposing the electron densities of the fragments from which a molecule is built forms a good starting density (by default atoms, but molecular fragments may also be used). For hybrid DFT, in which also a starting guess for the orbitals is required, one uses a superposition of fragment density matrices for orthonormalized orbitals. In both cases, a sufficiently converged result is typically obtained in a few tens of SCF iterations.

The most time-consuming part in the SCF iterations is the construction of the Fock matrix. The Fock matrix elements are calculated via three-dimensional (3D) numerical integration [14] using (5.1), with $\{\phi\}$ representing the basis set and $V(r_1)$ representing the potential energy operator containing both the Coulomb and exchange-correlation potential:

$$F_{\mu\nu} \leftarrow \int \phi_\mu(r_1) V(r_1) \phi_\nu(r_1) dr_1 \approx \sum_k^{N_G} W_k \phi_\mu(r_k) V(r_k) \phi_\nu(r_k). \tag{5.1}$$

Because the number of matrix elements $F_{\mu\nu}$ scales quadratically with the number of atomic orbitals N_{AO}, and the number of integration points N_G scales linearly, the cost of this step formally increases cubically with system size. Neglecting integration grid points in which one or more of the contributors to the integrand become negligibly small reduces this unfavorably high scaling. While this approximation can strongly reduce the number of floating-point operations that should be carried out, and ideally reduce the scaling with system size to linear in the asymptotic limit, it also introduces checking and branching in the algorithm, thereby possibly affecting the peak performance that can be attained. Such undesired side effects are minimized by operating as much as possible on entire batches of grid points, keeping an easy-to-optimize simple algorithm in the innermost parts of the code.

To reduce the number of operations as much as possible, we can rewrite the second part of Eq. (5.1) by combining the weighting factor W_k and the value of the potential V_k into a weighted potential value $O_k = W_k V(r_k)$. These weighted potential values can be reused for every combination of μ and ν, simplifying the integration to

$$F_{\mu\nu} \leftarrow \sum_k \phi_{\mu k} O_k \phi_{\nu k}. \tag{5.2}$$

Equation (5.2) can be rewritten as a matrix–matrix multiplication by introducing an auxiliary matrix $A_{\nu k} = O_k \phi_{\nu k}$. Use of this auxiliary matrix then also reduces the number of operations needed to evaluate the full Fock matrix to $N_{\mathrm{AO}} N_G + 2N_{\mathrm{AO}}^2 N_G$; this is demonstrated in Example 5.1.

Example 5.1 Using a temporary array to reduce the operations needed to create the Fock matrix.

```
do iBasis = 1, naos ! naos is the number of basis functions
 ! prepare the temporary array by multiplying the operator with basis function i
 do kGrid = 1, nGrid ! nGrid is the number of grid points
      temp(kGrid) = operat(kGrid)*basval(kGrid,iBasis)
 end do
 do jBasis = 1, naos
      ! loop over all gridpoints, and multiply temp with basis function j
      do kGrid = 1, nGrid
           FockMat(jBasis,iBasis) += temp(kGrid) * basval(kGrid,jBasis)
      end do
 end do
end do
```

●

5.2.2 CPU Code SCF Performance

A common element shared by the many different types of calculations possible with the ADF program is the determination of the Kohn–Sham orbitals for a given fixed molecular structure. For these so-called single-point calculations, the computational cost can be split into three major parts: preparation for the SCF (∼5–15%), the SCF cycles (70–80%), and the total bonding energy calculation (∼15%).

Table 5.1 *Timing details for a single-point DFT calculation on the budesonide molecule ($C_{25}H_{34}O_6$), using a Becke-type grid, the TZ2P basis set, the GGA BLYP functional, and the frozen cores "small" setting in ADF*

			Creating Fock matrix (**focky**)			
Integration grid accuracy	Total time	SCF cycle	Total **focky**	Numerical integration	Generate potentials	Other
Computational cost in seconds						
Normal	136.8	100.7	82.0	47.1	23.5	11.4
Good	274.7	197.2	179.1	115.6	47.4	16.1
Very good	710.8	544.0	518.7	338.2	145.5	35.0
Percentage of total time						
Normal	100%	74%	60%	34%	17%	8%
Good	100%	72%	65%	42%	17%	6%
Very good	100%	77%	73%	48%	20%	5%

The calculation was performed on a 12-core Intel Xeon E5-2620 running at 2.0 GHz.

Most of the time in the SCF cycle, as shown in Table 5.1, is spent on calculating the Fock matrix, and the largest part of the Fock matrix calculation is spent in the numerical integration routine. For high grid accuracies, the numerical integration of the Fock matrix elements covers almost 50% of the computational cost.

Because the Fock matrix is Hermitian (real-symmetric for calculations without inclusion of spin–orbit coupling), only half of the matrix elements are calculated using the symmetry $F_{\mu\nu} = F_{\nu\mu}$ to obtain the complete matrix. This reduces the computational cost and memory footprint by a factor of 2, but makes the loops over the atomic orbitals slightly more involved. For the loops over the grid points, a partitioning in batches of 128 points is used to optimize cache memory usage and allows for easy parallelization with MPI. Every MPI process works on its own set of batches, minimizing the need for communication between CPU cores and cluster nodes. Because batches of grid points always contain spatially localized points, it is possible to employ distance cutoffs to speed up the calculation. This is done by evaluating only the matrix elements over the subset of basis functions that have sufficiently large values in this batch and ignoring all functions that have values close to zero in the batch [8]. The number of these active basis functions, $N_{\mathrm{Act}.O}$, depends on the geometry of the system, the basis set, and the atom types in the system, and it will approach a constant value for large systems. For typical systems treated with ADF, $N_{\mathrm{Act}.O}$ is in the range of 300–700 basis functions, and using these distance effects will give a speed-up factor of 10 or more, as the number of operations needed to evaluate the full Fock matrix is reduced to $N_{\mathrm{Act}.O}N_G + 2N^2_{\mathrm{Act}.O}N_G$.

While it is most efficient to discard functions already before entering the inner loops of the algorithm, it is possible to obtain additional speedups by considering the combination of two functions that enter the integral in Eq. (5.1). If the product of the maximum values of ϕ_μ and ϕ_ν in the current batch of grid points is smaller than a threshold T, $|\max(\phi_\mu)\max(\phi_\nu)| < T$, their contribution to the Fock matrix element $F_{\mu\nu}$ can be neglected. While this is a minor speedup compared to the locality effects used to select the basis functions per batch of grid points, the check requires negligible time because these maximum values are already available from the basis functions selection routine.

The contraction of Eq. (5.2) can, in principle, be done as a matrix multiplication by using vendor-optimized libraries, but this creates an extra overhead for a relatively small number of operations in each matrix multiplication (due to all the optimizations related to linear scaling cutoffs). The current Fortran code attains roughly 4.5 billion floating-point operations per second (GFLOPS) on a single core of a Xeon E5-2620 CPU for which the MKL DGEMM shows a peak performance of about 14 GFLOPS. This higher speed of the library routine can for our purposes,

however, be obtained only by discarding some of the optimizations mentioned above, which would result in more flops to be carried out and a worse overall performance.

For the specific example shown in Table 5.1, a single-point calculation on budesonide, more than half of the computational time is used to create the Fock matrix by the routine **focky**. Of the steps done in **focky**, 18% goes into calculating the GGA potential in the integration grid, 9% is spent on calculating the Coulomb potential, and 65% is spent on the numerical integration (Eq. (5.2)) itself. The remaining time is spent on combining the results from different CPU cores into a single matrix.

The numerical integrations performed in the analytical frequencies code of ADF are similar to the one described above. The program then needs to calculate geometrical derivatives of the Fock matrix. This requires generation of a full perturbed Fock matrix, because this matrix needs no longer be Hermitian or real-symmetric. This shifts the emphasis more toward the integration step, a trend that is strengthened by the partial reuse of operator and basis values in the grid for each geometrical derivative. Two Fortran modules **M_bas** and **M_bas_mat** contain a number of routines for calculating different combinations of basis functions and their derivatives. **M_bas** is the lower level module that offers routines for generating basis values and numerical integration, while **M_bas_mat** is the higher level module for calculating Fock matrices by calling routines from **M_bas** and some other modules.

When calculating analytical frequencies, ∼65% of the computational cost is inside a routine called **f1u1_ai_gga**, which calculates the Fock matrix derivatives shown in Eq. (5.3). It spends about 80–85% of its time on the numerical integration, and the remaining 15–20% goes into generating the values of the operator and basis functions (and their derivatives) on the grid:

$$F_{\nu\mu}^{(1)} \leftarrow \sum_{k} \sum_{d}^{x,y,z} W_k \left[\nabla_d \phi_\mu \left(r_k\right)\right] V_d \left(r_k\right) \phi_\nu \left(r_k\right). \tag{5.3}$$

This integration is very similar to the one used in **focky** (Eq. (5.1)), and is optimized using the same linear scaling and operation-efficiency arguments.

5.3 GPU Implementation

In this section, we will first identify the hardware and software requirements for accelerating the ADF package with GPUs. We then describe the CUDA code we developed for the numerical integration and analyze its performance, followed by a description of the hybrid computational scheme used to accelerate ADF. Finally, we compare the performance of the GPU-accelerated ADF code with the reference CPU code for real simulations.

5.3.1 Hardware and Software Requirements

As mentioned in the previous section, we need GPU devices with high double-precision (DP) performance, as single precision would generate too much numerical noise. The Nvidia Fermi Tesla series, consisting of the C2050/2075 and M2090 models, were one of the first GPUs with a high DP peak performance (∼500 GFLOPs). The next generation is called Kepler and, at the time of writing, has three Tesla models, the K10, K20, and the K40. The K10 has over 4.5 TFLOPs single-precision performance but only ∼190 GFLOPs of DP performance. The K20 and K40 are aimed at double precision and achieve over 1000 GFLOPs of DP performance, while a six-core Intel Xeon E5-2620 manages only ∼90 GFLOPs. The current CUDA code for GPU acceleration in ADF was developed on the C2050 and C2075 Tesla devices, but it also runs on the newer K20 and K40.

The LAPACK routine DGEMM for matrix multiplications as implemented in the Nvidia cuBLAS library achieves ∼300 GFLOPs on the Fermi Tesla generation, which is ∼60% of the theoretical

peak performance. This DGEMM code is highly optimized, so for a home-made implementation with CUDA we cannot expect to achieve similar speeds. Half of the DGEMM performance on Fermi is therefore a reasonable target for our CUDA kernels, and further optimization would be useful only after porting other parts of the **focky** routine to the GPU.

The reasoning for not using standard BLAS libraries to solve Eq. (5.1) on the CPU also applies to the GPU, and determines the features needed for the CUDA kernels. The kernel should calculate only half of the Fock matrix if it is symmetric, and it should use distance cutoffs to reduce the number of active basis functions in a batch of integration grid points. The threshold check to skip specific combinations of basis functions inside a batch of grid points will introduce branching at warp level on the GPU, something that should be avoided. Because the contributions of these combinations are negligible by definition, we simply omit the check in the GPU kernel.

The kernels should be tuned for high performance with a relatively small problem size, because the number of active basis functions (functions that have a value higher than the threshold in at least one of the points in an integration batch) is usually not very large. The number of active basis functions is system dependent, but typically ∼25–50% of the total basis functions. The total number of basis functions for the budesonide molecule ($C_{25}H_{34}O_6$) with a TZ2P basis set is 1338, giving us an estimate of 300–700 active basis functions per integration batch for this example. The small problem size means that only a relatively small grid of CUDA threads can be created, making the usual practice of hiding memory latency behind the large number of other threads inefficient. CUDA kernels that do not use a large number of threads are called *low-occupancy kernels*, and they are sometimes much faster than high-occupancy kernels [15]. The low-occupancy kernels are tuned to take maximum advantage of the available shared and register memory, and reaching the best performance comes down to finding the best balance between the number of values calculated per thread, the amount of pre-caching, and the size of the CUDA blocks. Low-occupancy kernels therefore generally need modifications to obtain high performance on a new generation of CUDA devices, because the proportions between the hardware resources have changed.

5.3.2 GPU Kernel Code

Figure 5.1 shows a schematic representation of the CUDA kernel and wrapper routine used to offload the numerical integration in **focky**. The GPU kernel splits the calculation into batches of eight grid points to fit in shared memory, and pre-fetches the values used in the next iteration during the calculation loop to hide the global memory latency. Every thread calculates four values of the Fock matrix in the calculation loop to increase the available time for pre-fetching. The kernel transposes one set of basis function values to prevent shared memory bank conflicts in the calculation loop. Because the GPU calculates the Fock matrix without the use of an auxiliary matrix, the number of operations needed to evaluate the full Fock matrix is $3N^2_{\mathrm{Act.}O}N_G$.

The kernel is called from a C wrapper routine that takes care of the host-to-device memory transfers and synchronization between the host and device. The wrapper routine pads the basis functions array because the kernel accepts only multiples of 16 basis functions as input. This reduces the complexity of the CUDA kernel and also the amount of registers used on the GPU. The wrapper routine uses asynchronous copy operations to send data to the device to allow overlapping of computation and memory transfers for different MPI threads sharing the GPU.

It is important to optimize the kernel for a realistic problem size (300–700 active basis functions for the budesonide test system) instead of focusing on peak performance. Table 5.2 shows the performance of the current kernel for various problem sizes and different GPU devices. The timings are collected outside of the wrapper routine and thus include the time required for memory transfers. The transfer of the results from the device to the host is not taken into account, because it needs to be performed only after all batches of grid points are finished.

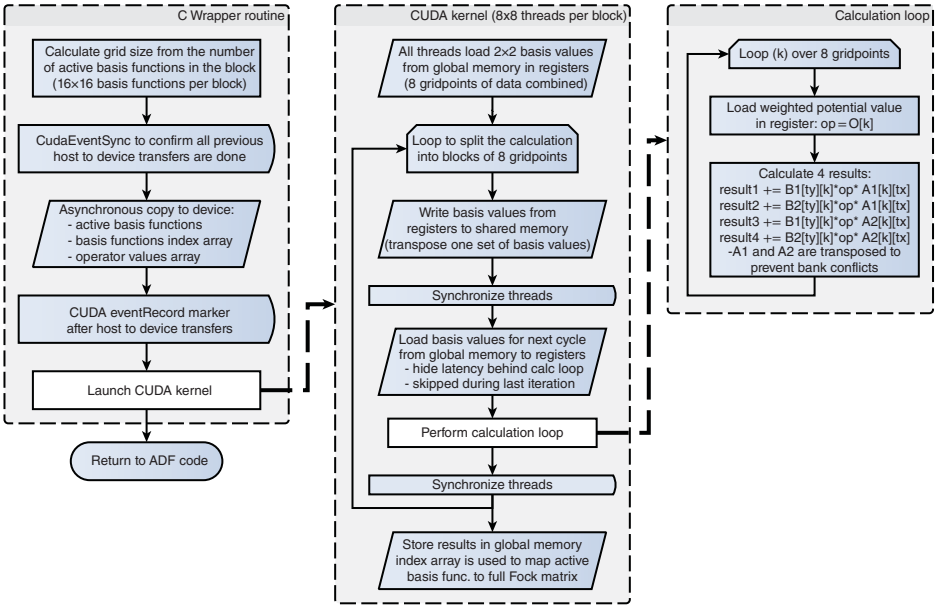


Figure 5.1 *Schematic representation of the CUDA kernel and wrapper routine for the numerical integration of Fock matrix elements*

Table 5.2 *Speed of the GPU kernel in GFLOPS on Nvidia Fermi and Kepler Tesla hardware*

N_{AO}	300	500	700	1400	4000
C2050/C2075	67	95	109	123	134
K20	71	121	173	222	262
K40 (boost)	78	133	191	298 (352)	348 (408)

The kernel is optimized for the 300–700 active basis functions (N_{AO}) range, and not for peak performance.

If the kernel is compiled with 16×16 threads per CUDA block instead of the current 8×8, it achieves a slightly larger top speed of around 155 GFLOPS on Fermi Tesla hardware. However, doing this reduces the performance in the region of 300–700 basis functions by ~10%, making the 8×8 threads per block a better choice. The kernel has not yet been retuned for the Kepler architecture, but performs reasonably well on both the K20 and K40 devices. The performance difference between K20 and K40 comes from the larger number of SMX units and the higher clock frequency on the K40. The boost state on K40 allows the device to automatically increase the clock frequency when the power usage is below the maximum, providing 17% speedup for sufficiently large calculations.

The GPU performance shown in Table 5.2 should not be directly compared with the performance of the CPU code (which reaches ~5.6 GFLOPs on a single CPU core), because the CPU routine needs less floating-point operations to compute the Fock matrix elements due to the use of an auxiliary matrix as explained in Section 5.2.2. Figure 5.2 shows the performance of the GPU code relative to the CPU version, based on the time needed to calculate the Fock matrix for different numbers of active basis functions in batches of integration grid points.

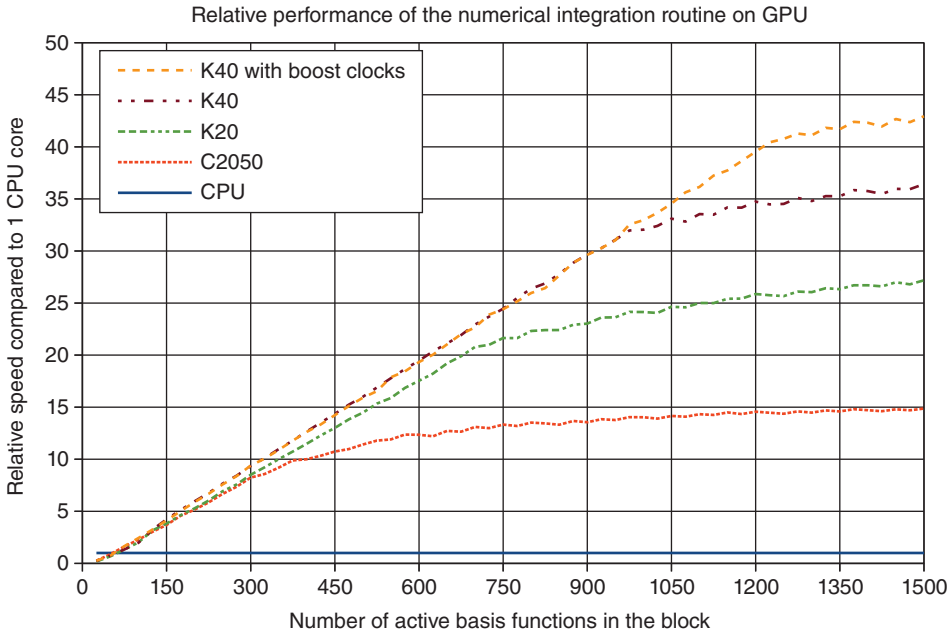


Figure 5.2 *Comparison of the time required by the GPU and CPU code for computing contributions to the Fock matrix through numerical integration*

5.3.3 Hybrid CPU/GPU Computing Scheme

As explained in Section 5.2.2, the calculation of the Fock matrix is partitioned into batches of grid points to allow for easy parallelization with MPI. There are no dependencies in the calculations of different batches, except for the summation of the results into the final Fock matrix. Figure 5.3 shows a hybrid computing scheme in which both the CPU and GPU work simultaneously on the calculation of the Fock matrix. The CPU calculates the values of basis functions, the Coulomb potential, and the XC potential at the grid points, which is followed by the GPU performing the numerical integration. Because the results of the GPU calculation are not needed to start the next batch, the CPU can continue working on the next batch while the GPU processes the numerical integration. The GPU stores the numerical integration results on the device, and they are transferred back to the host memory after all batches have been computed.

The ratio between numerical integration work and calculating the basis function and operator values on the grid depends on the calculation settings such as grid accuracy, basis set size, and the molecular system, but is ~2:1 for most simulations. It can be lower in cases such as graphene sheets, or higher for dense metal clusters with large basis sets, but test calculations on two organic molecules and two metal clusters with various settings showed this 2:1 ratio. So if the GPU code is twice as fast as the CPU, the time needed to perform the numerical integration will be completely hidden behind the remaining CPU code, removing the integration bottleneck. When ADF runs in parallel on N CPU cores and uses M GPUs in the machine, the GPU code needs to be roughly $2N/M$ times faster than a single CPU core to completely hide the GPU time behind other CPU work. Figure 5.2 can now be used to identify how many CPU cores can share the available GPUs to perform the numerical integration in the background.

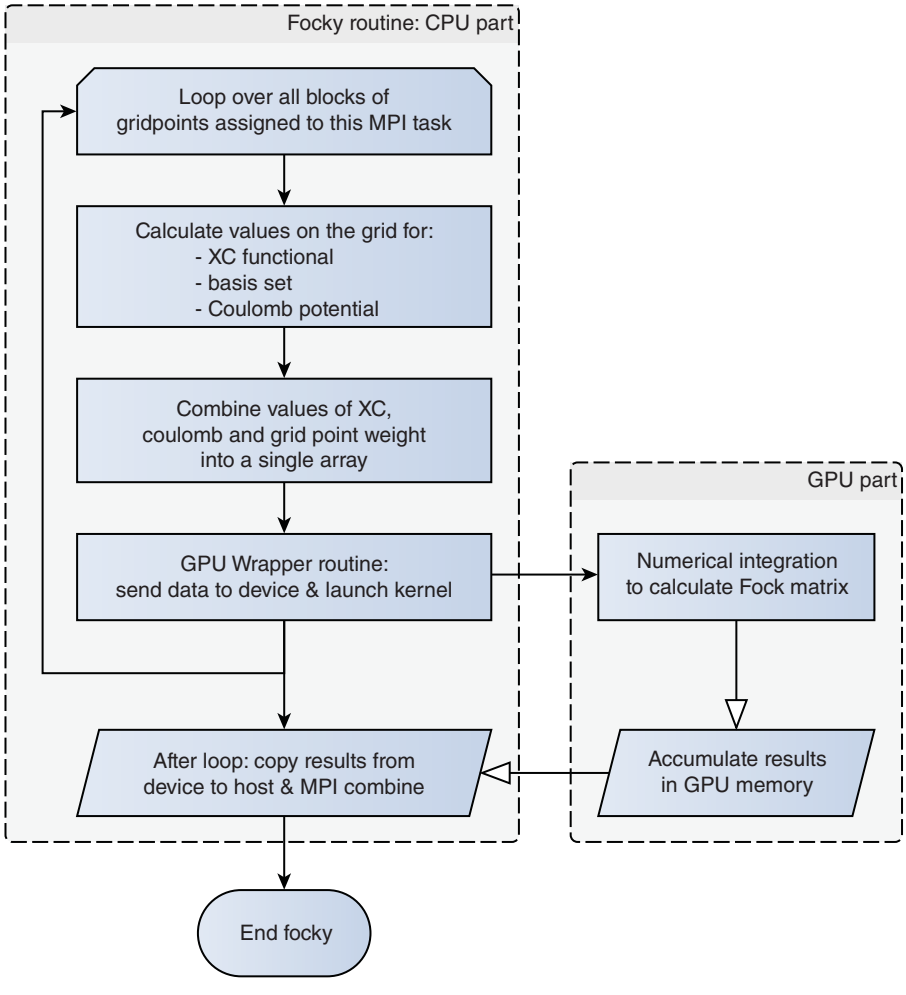


Figure 5.3 *Schematic representation of the hybrid CPU/GPU algorithm for the MPI parallel Fock matrix calculation*

The real application performance is slightly better than the speedup shown in Figure 5.2, because there are no dependencies between GPU kernels from different MPI threads. This allows the hardware to overlap the communication to the device from one MPI thread with the computation time of another. This effect is not very large on the Fermi Tesla hardware due to serialization on the single queue, leading to artificial dependencies. This was solved by the introduction of HyperQ with the Kepler K20, which upgraded the scheduler to a 32-channel queue system. This allows all MPI threads to use a separate hardware queue on the device, making it possible to run multiple integration kernels and memory transfers simultaneously. The kernel concurrency virtually increases the problem size the device is working on, meaning that the device will obtain speeds similar to a problem of double the size when working on four kernels simultaneously. Figure 5.2 shows that a single K20 GPU (~27× faster) is indeed sufficient to perform the numerical integration work for 12 CPU cores, and a K40 with boost clocks should be sufficient for up to 20 CPU cores.

Table 5.3 *CPU timings for a GPU-accelerated single-point DFT calculation on the budesonide molecule ($C_{25}H_{34}O_6$) using a Becke-type grid, a TZ2P basis set, the GGA BLYP functional, and the frozen cores "small" setting in ADF*

			Creating Fock matrix (**focky**)			
Integration grid accuracy	Total time	SCF cycle	Total **focky**	Numerical integration	Generate potentials	Other
Computational cost in seconds						
Normal	94.1	56.3	36.7	3.0	24.3	9.4
Good	170.7	86.3	67.7	6.7	49.0	12.0
Very good	375.7	194.5	170.6	14.7	132.5	23.4
Speed-up factor compared to CPU-only calculation						
Normal	1.45	1.79	2.24			
Good	1.61	2.29	2.64			
Very good	1.89	2.80	3.04			

The calculation was performed on 12 cores of the Intel Xeon E5-2620 CPU and a single Nvidia K40.

5.3.4 Speed-Up Results for a Single-Point Calculation

Table 5.3 shows the results of the test calculations with GPU acceleration enabled, as well as the speedup when compared to the CPU timings from Table 5.1. The hybrid computing scheme clearly performs as planned, as almost no time is spent on numerical integration. The small part that remains is mostly the overhead of the wrapper routine handling the memory transfers.

We can now also estimate the minimum average speedup of the numerical integration: the 12 CPU cores needed 338.2 s (Table 5.1) to perform the numerical integration on the "very good" grid setting, where the K40 used at most 170.6 s to perform the calculation. This makes the K40 1.98 times faster when compared to 12 CPU cores, or 23.8 times faster when compared to a single core. If we compare this number to the K40 curve in Figure 4.2, knowing that the average block in the calculation has between 300 and 700 active basis functions, we can conclude that the HyperQ feature works and multiple blocks are computed concurrently on the device.

5.3.5 Speed-Up Results for an Analytical Frequency Calculation

A large part of the computational cost during an analytical frequency calculation is spent on numerical integration (see Section 5.2.2). These calculations can also be GPU-accelerated using a hybrid scheme similar to the one used for the Fock matrix formation during the SCF. The most expensive part is the numerical integration of derivative Fock matrix elements, Eq. (5.3), which can be performed with a slightly modified kernel that includes a loop over the geometric derivatives of the basis functions. This method was chosen for ease of implementation, but requires the GPU to perform $3 \times (3N^2_{\text{Act.}O}N_G)$ operations to calculate the full derivative Fock matrix. The CPU code uses an auxiliary matrix (Eq. (5.4)), and thus only needs $5N_{\text{Act.}O}N_G + 2N^2_{\text{Act.}O}N_G$ operations to calculate the derivative Fock matrix:

$$A_{\mu k} = \sum_{d}^{x,y,z} O_{kd}\phi_{\mu kd}. \tag{5.4}$$

Because the GPU code does not use the auxiliary matrix, it is less efficient when compared to the CPU code, meaning that we will only see good acceleration if the ratio of CPU cores to GPU devices is small. A calculation with two CPU cores and two GPUs (a K20 and a K40) will give an estimation of the speedup that could be obtained with a realistic production setup (6–8 cores per GPU), if the GPU kernel is rewritten to use an auxiliary matrix.

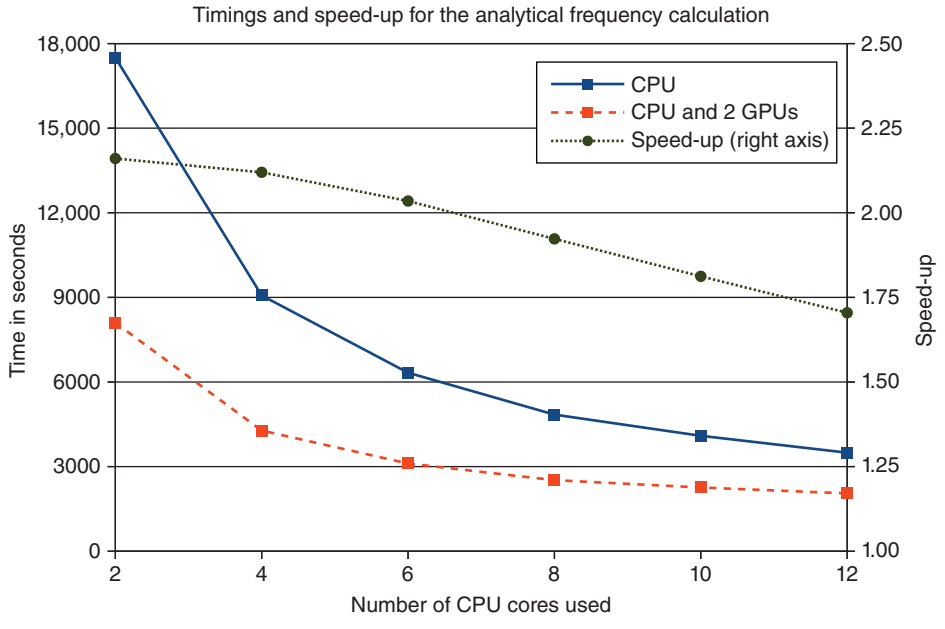


Figure 5.4 *Timings and speedup for the calculation of Hessian matrix elements for an analytical frequency calculation on the budesonide molecule ($C_{25}H_{34}O_6$) with BLYP/TZ2P and frozen core setting "small," using a Becke grid quality of "good" in ADF*

Figure 5.4 shows the timings and speedup for an analytical frequency calculation on the budesonide molecule ($C_{25}H_{34}O_6$) using a TZ2P basis set, the GGA BLYP functional, the frozen cores "small" setting, and a Becke grid quality "good" in ADF. The calculation is performed for only 3 of the 65 atoms, creating a partial Hessian, and Figure 5.4 shows only the timings for the loop over the atomic nuclei, because this loop will take ~98% of the total calculation time for a full Hessian. The timings are thus representative for a full frequency calculation. Figure 5.4 shows a 2.16× speedup when using only 2 CPU cores (6 CPU cores per GPU), and a 1.70× speedup when using 12 CPU cores (6 cores per GPU).

The calculations in the frequencies code are accelerated using the same hybrid scheme as explained in Section 5.3.3. The only difference is the ratio between numerical integration and calculation of the operator and basis function (derivative) values on the grid, which is ~4:1 for the **f1u1_ai_gga** routine. This gives a possible speedup factor of 5, which is slightly higher when compared to the **focky** routine. However, the number of CPU cores that share a GPU needs to be lower; otherwise the numerical integration cannot be hidden behind CPU work, reducing the possible speedup.

The timings and speedup of the **f1u1_ai_gga** routine are shown in Figure 5.5. When only two CPU cores are used, all numerical integration work performed on the two GPUs is hidden behind other CPU work, giving a 4.43× speedup. The estimated 5× possible speedup is not achieved because of the increased overhead of the wrapper routine, which reorders the input arrays on the CPU before sending them to the GPU. This is necessary to enable coalesced data fetches from global memory in the numerical integration kernel, something that is otherwise difficult to achieve with the current data structure used in the ADF gradients code. The speedup reduces to 2.66× when using 12 CPU cores, showing that there is still room for improvement with the current GPU acceleration scheme.

Bringing the auxiliary matrix scheme to GPU would be a good way to reduce the number of operations required in each integration grid batch and at the same time solve the data structure problem.

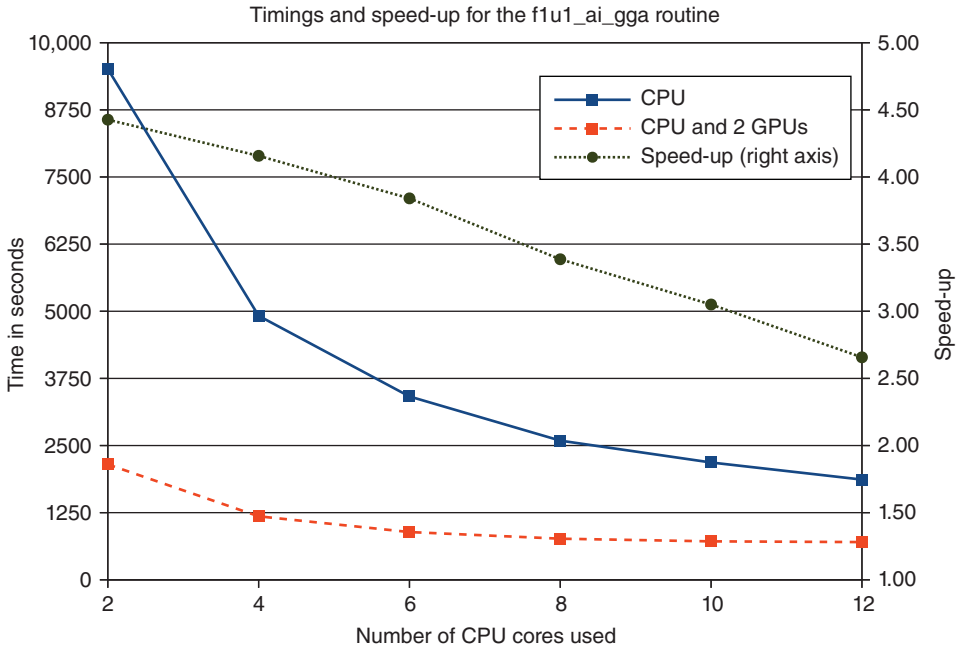


Figure 5.5 *Timings and speedup for the calculation of matrix elements of the Fock matrix derivatives (routine* ***f1u1_ai****_gga) required for analytical frequency calculation. Test system is the budesonide molecule ($C_{25}H_{34}O_6$) with BLYP/TZ2P and frozen core setting "small," using a Becke grid quality of "good" in ADF*

5.4 Conclusion

DFT calculations with STOs as implemented in the ADF software package were accelerated using a low-occupancy CUDA kernel for the numerical integration of Fock matrix elements, aimed at obtaining good performance for relatively small problem sizes. The kernel achieved almost 50% of the DGEMM performance on the Tesla Fermi architecture (134 GFLOPS), and was portable enough to run twice as fast on K20 (262 GFLOPS) and thrice faster on K40 (408 GFLOPS).

A hybrid computational scheme was developed to offload the heavy numerical work to the GPU, keeping the CPU working at the same time. The balance in workload between CPU and GPU causes the numerical integration bottleneck to disappear if the number of CPU cores per GPU is not too large. The accelerated ADF code was shown to be 1.45–1.89 times faster for single-point calculations, depending on the grid accuracy settings, and 1.70–2.16 times faster for frequency calculations, depending on the ratio of CPU cores to GPU devices.

Software packages with a large amount of legacy code, such as ADF, can be difficult targets for GPU acceleration, as it is impossible to port them entirely to GPUs within a reasonable amount of time. The hybrid scheme described here can be a good solution for such cases, offering decent speedup without sacrificing any of the features that make the code unique.

GPU acceleration in ADF is work in progress with plenty of room for improvement. In the near future, we hope to optimize the frequencies code and extend the acceleration to functionals beyond GGAs and other property calculations.

References

1. Kohn, W., Becke, A.D. and Parr, R.G. (1996) Density functional theory of electronic structure. *Journal of Physical Chemistry A*, **100** (31), 12974–12980.
2. Neese, F. (2009) Prediction of molecular properties and molecular spectroscopy with density functional theory: From fundamental theory to exchange-coupling. *Coordination Chemistry Reviews*, **253** (5–6), 526–563.
3. Cohen, A.J., Mori-Sánchez, P. and Yang, W. (2012) Challenges for density functional theory. *Chemical Reviews*, **112** (1), 289–320.
4. Kutzelnigg, W. (2013) Expansion of a wave function in a Gaussian basis. I. Local versus global approximation. *International Journal of Quantum Chemistry*, **113** (3), 203–217.
5. Güell, M., Luis, J.M., Solà, M. and Swart, M. (2008) Importance of the basis set for the spin-state energetics of iron complexes. *Journal of Physical Chemistry A*, **112** (28), 6384–6391.
6. ADF2013, *SCM, Theoretical Chemistry, Vrije Universiteit, Amsterdam, The Netherlands*, `http://www.scm.com` (accessed 07 September 2015).
7. Te Velde, G., Bickelhaupt, F.M., Baerends, E.J. *et al.* (2001) Chemistry with ADF. *Journal of Computational Chemistry*, **22** (9), 931–967.
8. Fonseca Guerra, C., Snijders, J., Te Velde, G. and Baerends, E. (1998) Towards an order-N DFT method. *Theoretical Chemistry Accounts*, **99** (6), 391–403.
9. Cramer, C.J. (2004) *Essentials of Computational Chemistry: Theories and Models*, John Wiley & Sons, p. 628.
10. Jensen, F. (2007) *Introduction to Computational Chemistry*, 2nd edn, John Wiley & Sons, Chichester.
11. Fonseca Guerra, C., Visser, O., Snijders, J.G. te Velde, G. and Baerends, E.J. (1995) *Parallelisation of the Amsterdam Density Functional Program*. In: Methods and Techniques for Computational Chemistry, E. Clementi and C. Corongiu, eds, pp. 303–395, STEF, Cagliari. `https://www.scm.com/Doc/metecc.pdf`.
12. Wolff, S.K. (2005) Analytical second derivatives in the Amsterdam density functional package. *International Journal of Quantum Chemistry*, **104** (5), 645–659.
13. Nicu, V.P., Neugebauer, J., Wolff, S.K. and Baerends, E.J. (2008) A vibrational circular dichroism implementation within a Slater-type-orbital based density functional framework and its application to hexa- and hepta-helicenes. *Theoretical Chemistry Accounts*, **119** (1–3), 245–263.
14. Franchini, M., Philipsen, P.H.T. and Visscher, L. (2013) The Becke fuzzy cells integration scheme in the Amsterdam density functional program suite. *Journal of Computational Chemistry*, **34** (21), 1819–1827.
15. Volkov V. 2010 *Better performance at lower occupancy*. Proceedings of the GPU Technology Conference, GTC; http://people.sc.fsu.edu/~gerlebacher/gpus/better_performance_at_lower_occupancy_gtc2010_volkov.pdf

6

Wavelet-Based Density Functional Theory on Massively Parallel Hybrid Architectures

Luigi Genovese[1], *Brice Videau*[2], *Damien Caliste*[1], *Jean-François Méhaut*[2], *Stefan Goedecker*[3] *and Thierry Deutsch*[1]

[1] *Université Grenoble Alpes, INAC, F-38000 Grenoble, France, and CEA, INAC, F-38000 Grenoble, France*

[2] *Université Joseph Fourier – Laboratoire d'Informatique de Grenoble – INRIA, Grenoble, France*

[3] *Institut für Physik, Universität Basel, Basel, Switzerland*

In this chapter, we describe the GPU acceleration of density functional theory (DFT) calculations based on wavelet basis sets as realized in the BigDFT code. Daubechies wavelets have not been traditionally used for DFT calculations, but they exhibit properties that make them attractive for both accurate and efficient DFT simulations. Here we explain how an existing MPI parallel wavelet-based DFT implementation can benefit from the computational power of GPUs by offloading numerically intensive operations to GPUs.

6.1 Introductory Remarks on Wavelet Basis Sets for Density Functional Theory Implementations

Quantum mechanics and electromagnetism are widely perceived as leading to a "first-principles" approach to materials and nanosystems: given appropriate software implementations and sufficiently powerful hardware, it is possible to calculate properties without resorting to any adjustable parameters. In the 1980s, it became clear that, indeed, numerous material properties such as total energies, electronic structure, and the related dynamic, dielectric, mechanical, magnetic, and vibrational properties, can be obtained with an accuracy that can be considered as truly predictive.

Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

As described in Chapter 3, DFT in its Kohn–Sham (KS) single-particle approach [1] is at present the most widely used methodology for electronic structure calculations. In the recent years, the KS formalism has been proven to be one of the most efficient and reliable first-principles methods for predicting material properties and processes that exhibit a quantum mechanical behavior. The combination of high accuracy and relatively simple form of most common exchange-correlation (XC) functionals make KS-DFT probably the most powerful tool for *ab initio* simulations of the properties of matter. As a consequence, the computational machinery of DFT calculations has been widely developed during the last decade, giving rise to a plethora of DFT codes. DFT calculations have thus become increasingly common, with application domains including solid-state physics, chemistry, materials science, biology, and geology.

From a computational point of view, one of the most important characteristics of a DFT code is the set of basis functions used to express the KS orbitals. The domain of applicability of a code is tightly connected to this choice. For example, a nonlocalized basis set such as plane waves (see Chapter 7) is highly suitable for electronic structure calculations of periodic and/or homogeneous systems such as crystals or solids. It is much less efficient in expanding localized information, which has a wider range of components in the reciprocal space, leading to high memory requirements. For this reason, DFT codes based on plane waves are not convenient for simulating inhomogeneous or isolated systems such as molecules.

A distinction should also be made between codes that use systematic and nonsystematic basis sets. A systematic basis set allows the calculation of the exact solution of the KS equations with arbitrarily high precision as the number of basis functions is increased. In other words, the numerical precision of the results is related to the number of basis functions used to expand the KS orbitals. With such a basis set it is thus possible to obtain results that are free of errors related to the choice of the basis, eliminating a source of uncertainty. A systematic basis set allows thus the calculation of the exact solution of a particular XC functional. On the other hand, an example of a nonsystematic set is provided by Gaussian-type basis functions (see Chapter 4), for which over-completeness may be achieved before convergence. Such basis sets are more difficult to use, since the basis set must be carefully tuned by hand, which sometimes requires preliminary knowledge of the system under investigation. This is the most important weakness of this popular basis set.

In 2005, the EU FP6-STREP-NEST BigDFT project funded a consortium of four European laboratories (L_Sim, CEA Grenoble; Basel University, Switzerland; Louvain-la-Neuve University, Belgium; and Kiel University, Germany) with the aim of developing a novel approach for DFT calculations based on Daubechies wavelets. Rather than simply building a DFT code from scratch, the objective of this 3-year project was to test the potential benefit of a new formalism in the context of electronic structure calculations.

As a matter of fact, Daubechies wavelets exhibit a set of properties that make them ideal for a precise and optimized DFT approach. In particular, they are systematic and thus provide a reliable basis set for high-precision results, whereas their locality (both in real and reciprocal space) improves the efficiency and the flexibility of the treatment. Indeed, a localized basis set allows the optimization of the number of degrees of freedom for a required accuracy [2], which is highly desirable given the complexity and inhomogeneity of the systems under investigation nowadays. Moreover, an approach based on localized functions makes it possible to explicitly control the nature of the boundaries of the simulation domain, which allows considering complex environments such as mixed boundary conditions and/or systems with a net charge.

As outlined in Chapter 1, the possibility of using graphics processing units (GPUs) for scientific calculations has raised much interest in the past few years. A technology that was initially developed for home PC hardware has rapidly evolved in the direction of programmable parallel streaming processors. The features of these devices, in particular the very low price to performance ratio, together with the relatively low energy consumption, make them attractive platforms for intensive scientific computations. Many scientific applications have been recently ported to GPUs, including, for example, molecular dynamics [3], quantum Monte Carlo [4], and finite element methods [5].

The numerical operations of the BigDFT code are well suited for GPU acceleration. On one hand, the computational nature of 3D separable convolutions allows writing efficient routines that benefit the computational power of GPUs. On the other hand, the parallelization scheme of the BigDFT code is optimal in this sense: GPUs can be used without affecting the nature of the communications between the different MPI processes.

In what follows, we give an overview of typical numerical operations in BigDFT that were ported to GPUs and discuss the effect of GPU acceleration of some code sections on the overall code performance.

6.2 Operators in Wavelet Basis Sets

Wavelet basis sets have not been widely used for electronic structure calculations. Most of the efforts so far have been devoted to their use for all-electron calculations. Since this basis set is not very common, we here explain its use in the context of KS-DFT calculation. For an exhaustive presentation of how wavelet basis sets can be used for numerical simulations, we refer the reader to the work by Goedecker [6]. In what follows, we summarize the main properties of Daubechies wavelets, with a special focus on the representation of the objects (wave functions and operators) involved in the KS-DFT formalism. We will first start by illustrating the principles of a one-dimensional (1D) Daubechies wavelets basis.

6.2.1 Daubechies Wavelets Basis and Convolutions

Every wavelet family comprises a *scaling function* ϕ, and a second function ψ, properly called a *wavelet*. Figure 6.1 illustrates the least asymmetric Daubechies wavelet family of order $2m = 16$, the basis set that is used at present in the BigDFT code. These functions feature a compact support $[1 - m, m]$ and are smooth; and therefore are localized in Fourier space as well. A basis set is simply generated by the integer translates of the scaling and wavelet functions, with arguments measured

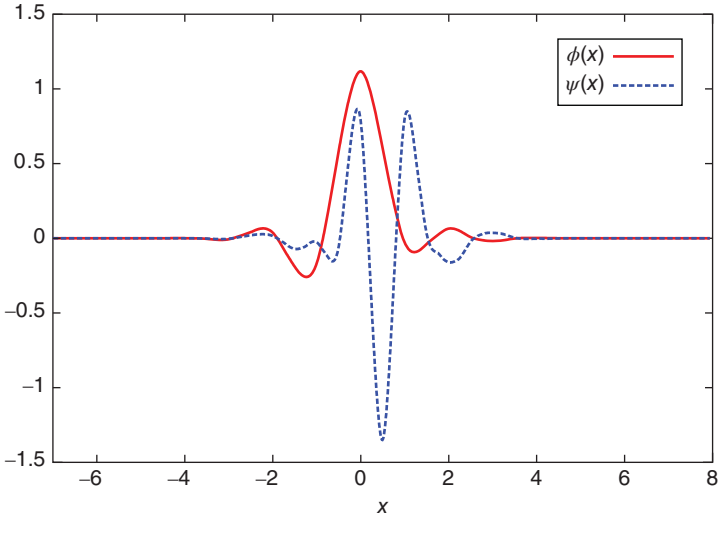


Figure 6.1 *Least asymmetric Daubechies wavelet family of order $2m = 16$. Note that both the scaling function $\phi(x)$ and the wavelet $\psi(x)$ are different from zero only within the interval $[1 - m, m]$*

in units of the grid spacing h. For instance, a 1D domain of extension L, centered at $x = 0$, can be spanned by the following set of N scaling functions:

$$\langle x|\phi_i\rangle \equiv \phi_i(x) = \frac{1}{\sqrt{h}}\,\phi\left(\frac{x}{h} - i\right), \quad i = -N/2, \ldots, N/2, \tag{6.1}$$

where $h = L/(N-1)$ is the (uniform) grid spacing. The basis set can be completed by the addition of the translates of the wavelet functions ψ_i. These functions form an orthogonal basis set:

$$\langle \phi_i|\phi_j\rangle = \delta_{ij} = \langle \psi_i|\psi_j\rangle, \quad \langle \phi_i|\psi_j\rangle = 0. \tag{6.2}$$

The most important feature of any wavelet basis set is related to the concept of *multiresolution*. This feature builds upon the following *scaling equations* (or "refinement relations"):

$$\phi(x) = \sqrt{2}\sum_j h_j\,\phi(2x - j); \quad \psi(x) = \sqrt{2}\sum_j g_j\,\phi(2x - j), \tag{6.3}$$

which relate the wavelet representation at some resolution to that at twice the given resolution, and so on. According to the standard nomenclature, the sets of the h_j and $g_j = (-1)^j h_{-j}$ coefficients are called *low-* and *high-pass* filters, respectively. A wavelet family is therefore completely defined by its low-pass filter. In the case of Daubechies-2m wavelets, $j \in [1-m, m]$.

The representation $f(x)$ of a function in the above-defined basis set is given by

$$f(x) = \sum_{i=-N/2}^{N/2} c_i\,\phi_i(x) + \sum_{i=-N/2}^{N/2} d_i\,\psi_i(x), \tag{6.4}$$

where the expansion coefficients are formally given by $c_i \equiv \langle \phi_i|f\rangle$, $d_i \equiv \langle \psi_i|f\rangle$. Using the refinement Eq. (6.3), one can map the basis appearing in Eq. (6.4) to an equivalent one including only scaling functions on a finer grid of spacing $h/2$.

The multiresolution property plays a fundamental role also for the wavelet representation of differential operators. For example, it can be shown that the *exact* matrix elements of the kinetic operator

$$T_{i-j} \equiv -\frac{1}{2}\int \mathrm{d}x\;\phi_i(x)\partial^2\phi_j(x) \tag{6.5}$$

are equal to the entries of an eigenvector of a matrix, which solely depends on the low-pass filter [6].

Daubechies-2m wavelets exhibit m vanishing moments, thus any polynomial of degree less than m can be represented exactly by an expansion over the sole scaling functions of order m. For higher order polynomials, the error is $\mathcal{O}(h^m)$, that is, vanishingly small as soon as the grid is sufficiently fine. Hence, the difference between the representation of Eq. (6.4) and the exact function f is decreasing as h^m. The discretization error due to Daubechies-2m wavelets is therefore controlled by the grid spacing. Among all the orthogonal wavelet families, Daubechies wavelets feature the minimum support length for a given number of vanishing moments.

Given a potential V known numerically on the points $\{x_k\}$ of a uniform grid, it is possible to identify an effective approximation for the potential matrix elements $V_{ij} \equiv \langle \phi_j|V|\phi_i\rangle$. It has been shown [2, 7] that a quadrature filter $\{\omega_k\}$ can be defined such that the matrix elements given by

$$V_{ij} \equiv \langle \phi_j|V|\phi_i\rangle = \sum_k \omega_{k-i}\;V(x_k)\;\omega_{k-j} \tag{6.6}$$

yield excellent accuracy with the optimal convergence rate $\mathcal{O}(h^{2m})$ for the potential energy. The same quadrature filter can be used to express the grid point values of a (wave) function given its expansion

coefficients in terms of scaling functions:

$$f(x_k) = \sum_i c_i\ \omega_{k-i} + \mathcal{O}(h^m); \tag{6.7}$$

$$c_i = \sum_k f(x_k)\ \omega_{k-i} + \mathcal{O}(h^m). \tag{6.8}$$

As a result, the potential energy can be equivalently computed either in real space or in the wavelet space; that is $\langle f|V|f\rangle = \sum_k f(x_k)V(x_k)f(x_k) \equiv \sum_{ij} c_i V_{ij} c_j$. The quadrature filter elements can therefore be considered as the most reliable transformation between grid point values $f(x_k)$ and scaling function coefficients c_i, as they provide exact results for polynomials of order up to $m-1$ and do not alter the convergence properties of the basis set discretization. The filter $\{\omega_k\}$ is of length $2m$ and is defined unambiguously by the moments of the scaling functions (which in turn depend only on the low-pass filter) [8].

Using the above formulae, the (so far 1D) Hamiltonian matrix $H_{ij} = T_{ij} + V_{ij}$ can be constructed. Note that, in contrast to other discretization schemes (finite differences, DVR, plane waves, etc.), in the wavelet basis set *neither* the potential *nor* the kinetic terms have diagonal representations. Instead, $\hat{H}$ is represented by a *band matrix* of width $2m$.

6.2.2 The Kohn–Sham Formalism

In the KS formulation of DFT, the KS wave functions $|\Psi_i\rangle$ are eigenfunctions of the KS Hamiltonian (see also Chapter 3), with pseudopotential V_{psp}:

$$\left(-\frac{1}{2}\nabla^2 + V_{\text{KS}}[\rho] + V_{\text{psp}}\right)|\Psi_i\rangle = \epsilon_i|\Psi_i\rangle. \tag{6.9}$$

The KS potential $V_{\text{KS}}[\rho]$ is a functional of the electronic density of the system:

$$\rho(\mathbf{r}) = \sum_{i=1}^{\mathcal{N}_{\text{orbitals}}} n_{\text{occ}}^{(i)}|\Psi_i(\mathbf{r})|^2, \tag{6.10}$$

where $n_{\text{occ}}^{(i)}$ is the occupation of orbital i.

The KS potential $V_{\text{KS}}[\rho] = V_H[\rho] + V_{\text{xc}}[\rho] + V_{\text{ext}}$ contains the Hartree potential V_H, solution of the Poisson's equation $\nabla^2 V_H = -4\pi\rho$, the XC potential V_{xc}, and the external ionic potential V_{ext} acting on the electrons. In the BigDFT code, the pseudopotential term V_{psp} is of the form of norm-conserving GTH–HGH pseudopotentials [9–11], which have a local and a nonlocal term, $V_{\text{psp}} = V_{\text{local}} + V_{\text{nonlocal}}$. The KS Hamiltonian can then be written as the action of three operators on the wave function:

$$\left(-\frac{1}{2}\nabla^2 + V_L + V_{\text{nonlocal}}\right)|\Psi_i\rangle = \epsilon_i|\Psi_i\rangle, \tag{6.11}$$

where $V_L = V_H + V_{\text{xc}} + V_{\text{ext}} + V_{\text{local}}$ is a real-space-based (local) potential, and V_{nonlocal} comes from the pseudopotentials.

As usual in a KS-DFT calculation, the application of the Hamiltonian is a part of a self-consistent cycle needed for minimizing the total energy. In addition to the usual orthogonalization routine, in which scalar products $\langle\Psi_i|\Psi_j\rangle$ should be calculated, another operation that is performed on wave functions in the BigDFT code is the preconditioning. This is calculated by solving the Helmholtz equation

$$\left(-\frac{1}{2}\nabla^2 - \epsilon_i\right)|\tilde{g}_i\rangle = |g_i\rangle, \tag{6.12}$$

where $|g_i\rangle$ is the gradient of the total energy with respect to the wave function $|\Psi_i\rangle$, of energy ϵ_i. The preconditioned gradient $|\tilde{g}_i\rangle$ is found by solving Eq. (6.12) by a preconditioned conjugate gradient method.

The complete Hamiltonian contains also the nonlocal part of the pseudopotential which, thanks to the orthogonality of Daubechies wavelets, can directly be applied in the compressed form. A schematic of all these operations is depicted in Figure 6.3.

6.2.3 Three-Dimensional Basis

For a 3D description, the simplest basis set is obtained by a set of products of equally spaced scaling functions on a grid of spacing h':

$$\phi_{i,j,k}(\mathbf{r}) = \phi(x/h' - i)\ \phi(y/h' - j)\ \phi(z/h' - k). \tag{6.13}$$

In other words, the 3D basis functions are a tensor product of 1D basis functions. Note that we are using a cubic grid, where the grid spacing is the same in all directions, but the following description can be straightforwardly applied to general orthorombic grids.

The basis set of Eq. (6.13) is equivalent to a mixed basis set of scaling functions on a twice coarser grid of grid spacing $h = 2h'$:

$$\phi^0_{i,j,k}(\mathbf{r}) = \phi(x/h - i)\ \phi(y/h - j)\ \phi(z/h - k) \tag{6.14}$$

augmented by a set of seven wavelets

$$\begin{aligned}
\phi^1_{i,j,k}(\mathbf{r}) &= \psi(x/h - i)\ \phi(y/h - j)\ \phi(z/h - k),\\
\phi^2_{i,j,k}(\mathbf{r}) &= \phi(x/h - i)\ \psi(y/h - j)\ \phi(z/h - k),\\
\phi^3_{i,j,k}(\mathbf{r}) &= \psi(x/h - i)\ \psi(y/h - j)\ \phi(z/h - k),\\
\phi^4_{i,j,k}(\mathbf{r}) &= \phi(x/h - i)\ \phi(y/h - j)\ \psi(z/h - k),\\
\phi^5_{i,j,k}(\mathbf{r}) &= \psi(x/h - i)\ \phi(y/h - j)\ \psi(z/h - k),\\
\phi^6_{i,j,k}(\mathbf{r}) &= \phi(x/h - i)\ \psi(y/h - j)\ \psi(z/h - k),\\
\phi^7_{i,j,k}(\mathbf{r}) &= \psi(x/h - i)\ \psi(y/h - j)\ \psi(z/h - k).
\end{aligned} \tag{6.15}$$

This equivalence follows from the fact that, from Eq. (6.3), every scaling function and wavelet on a coarse grid of spacing h can be expressed as a linear combination of scaling functions at the fine grid level h', and vice versa.

A KS wave function $\Psi(\mathbf{r})$ can thus be expanded in this basis:

$$\Psi(\mathbf{r}) = \sum_{i_1,i_2,i_3} c^0_{i_1,i_2,i_3}\phi^0_{i_1,i_2,i_3}(\mathbf{r}) + \sum_{j_1,j_2,j_3}\sum_{\nu=1}^{7} c^\nu_{j_1,j_2,j_3}\phi^\nu_{j_1,j_2,j_3}(\mathbf{r}). \tag{6.16}$$

The sums over i_1, i_2, i_3 (j_1, j_2, j_3) run over all grid points where scaling functions (wavelets) are centered. These points are associated with regions of low and high resolution levels, respectively.

In a simulation domain, there are three categories of grid points: those that are closest to the atoms ("fine region") carry 1D (3D) scaling function and seven (3D) wavelets; those that are further from the atoms ("coarse region") carry only one scaling function, corresponding to a resolution which is half that of the fine region; and those that are even further away ("empty region") carry neither scaling functions nor wavelets. To determine these regions of different resolution, we construct two spheres around each atom a; a small one with radius $R^f_a = \lambda^f \cdot r^f_a$, and a large one with radius $R^c_a = \lambda^c \cdot r^c_a$ ($R^c_a > R^f_a$). The values of r^f_a and r^c_a are fixed for each atom type, whereas λ^f and λ^c can be specified

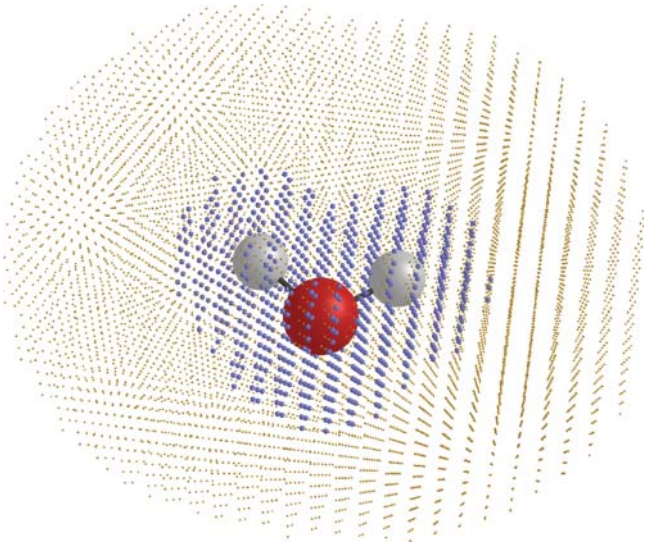


Figure 6.2 *Simulation grid points*

by the user in order to control the accuracy of the calculation. The fine (coarse) region is then given by the union of all the small (large) spheres, as shown in Figure 6.2. Hence in BigDFT the basis set is controlled by three user-specified parameters: systematic convergence of the total energy is achieved by increasing the values of λ^c and λ^f while reducing the value of h.

6.2.4 The Kinetic Operator and the Local Potential

For the pure fine scaling function representation described in Eq. (6.13), the result of the application of the kinetic energy operator on this wave function has the expansion coefficients $\hat{s}_{i'_1,i'_2,i'_3}$, which are related to the original coefficients $s_{i'_1,i'_2,i'_3}$ by a convolution

$$\hat{s}_{i'_1,i'_2,i'_3} = \frac{1}{2} \sum_{j'_1 j'_2 j'_3} K_{i'_1-j'_1,i'_2-j'_2,i'_3-j'_3} s_{j'_1,j'_2,j'_3}, \tag{6.17}$$

where

$$K_{i_1,i_2,i_3} = T_{i_1} \delta_{i_2} \delta_{i_3} + \delta_{i_1} T_{i_2} \delta_{i_3} + \delta_{i_1} \delta_{i_2} T_{i_3}, \tag{6.18}$$

and T_i are the filters of the 1D second derivative in Daubechies scaling functions basis (Eq. (6.5)), which can be computed analytically.

The potential V_L is defined in real space, in particular on the points of the finer grid of spacing h'. The application of the local potential in Daubechies basis consists of the basis decomposition of the function product $V_L(\mathbf{r})\Psi(\mathbf{r})$. As explained in the literature [2, 7], the simple evaluation of this product in terms of the point values of the basis functions is not precise enough. A better result may be achieved by performing a transformation to the wave function coefficients, which allows the calculation of the values of the wave functions on the fine grid, via a smoothed version of the basis functions. This is the so-called magic filter transformation, which can be expressed as follows:

$$\Psi(\mathbf{r}_{i'_1,i'_2,i'_3}) = \sum_{j'_1 j'_2 j'_3} \omega_{i'_1-j'_1} \omega_{i'_2-j'_2} \omega_{i'_3-j'_3} s_{j'_1,j'_2,j'_3} \tag{6.19}$$

and allows expressing the potential application with higher accuracy. In other words, the point values of a given wave function $|\Psi\rangle$ are expressed as if $\Psi(\mathbf{r})$ would be the smoothest function that has the same Daubechies expansion coefficients of $|\Psi\rangle$. This procedure guarantees the highest precision ($\mathcal{O}(h^{16})$ in the potential energy) and can be computationally expressed by a 3D separable convolution in terms of the filters ω_i. After application of the local potential (pointwise product), a transposed magic filter transformation can be applied to obtain Daubechies expansion coefficients of $V_L|\Psi\rangle$.

The above-described operations must be combined together for the application of the local Hamiltonian $\left(-\frac{1}{2}\nabla^2 + V_L(\mathbf{r})\right)$, which is depicted in Figure 6.3. The complete Hamiltonian contains also the nonlocal part of the pseudopotential which, thanks to the orthogonality of Daubechies wavelets, can be directly applied in the compressed form. All the other operations can also be performed in the compressed form. In particular, the overlap matrices needed for the orthogonality constraints, and their manipulations, are implemented by suitable applications of the BLAS and LAPACK routines.

6.2.5 Poisson Solver

The local potential V_L can be obtained from the local density ρ by solving Poisson's equation and by calculating the exchange–correlation potential $V_{xc}[\rho]$. These operations are performed via a Poisson solver based on interpolating scaling functions [12], which is a basis set tightly connected with Daubechies functions, optimal for electrostatic problems, and which allows for mixed boundary conditions. A description of this Poisson solver can be found in [13–15].

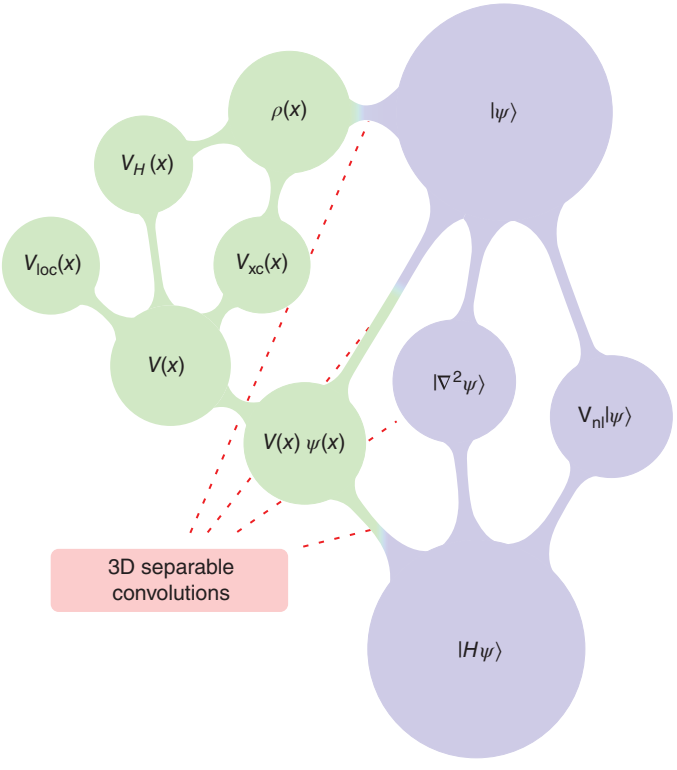


Figure 6.3 *Schematic representation of the application of the Hamiltonian in the BigDFT formalism*

6.3 Parallelization

Two data distribution schemes are used in the parallel version of our program. In the orbital distribution scheme, each processor works on one or a few orbitals. The processor holds its scaling function coefficients and wavelet coefficients. In the coefficient distribution scheme (see Figure 6.4), each processor holds a certain subset of the coefficients of all the orbitals. Most of the operations, such as applying the Hamiltonian on the orbitals and the preconditioning, are done in the orbital distribution scheme. This has the advantage that we do not have to parallelize these routines with MPI. The calculation of the Lagrange multipliers that enforce the orthogonality constraints onto the gradient as well as the orthogonalization of the orbitals is done in the coefficient distribution scheme (Figure 6.4). Switching back and forth between the orbital distribution scheme and the coefficient distribution scheme is done by the MPI global transposition routine MPI_ALLTOALL(V). For parallel computers where the cross-sectional bandwidth [16] scales well with the number of processors, this global transposition does not require much CPU time. Another time-consuming communication is the global reduction sum required to obtain the total charge distribution from the partial charge distribution of the individual orbital.

In the parallelization scheme of the BigDFT code, another level of parallelization was added via the OpenMP directive. In particular, all the convolutions and the linear algebra part can be executed in the multi-threaded mode. This adds further flexibility to the parallelization scheme. Several tests and improvements have been made to stabilize the behavior of the code in a multilevel MPI/OpenMP parallelization. At present, optimal performance can be reached by associating one MPI process per CPU (socket), or even one MPI process per compute node, depending on the network and MPI library performances. This has been possible also thanks to recent improvements of the OpenMP implementation of the compilers.

6.3.1 MPI Parallel Performance and Architecture Dependence

The parallellization scheme of the code has been continuously tested since its first version. Since MPI communications do not interfere with calculations, as long as the computational workload is more demanding than the time required for the communication, the overall efficiency remains high, also for simulations with a large number of processors [17].

We have evaluated the amount of time spent for a given operation on a typical run. To do this, we have profiled the different sections of the BigDFT code for parallel calculations. In Figure 6.5, we show the percentage of time that is dedicated to the operations described above for runs with two different architectures: the French CCRT Titane platform (Bull Novascale R422), and the Swiss

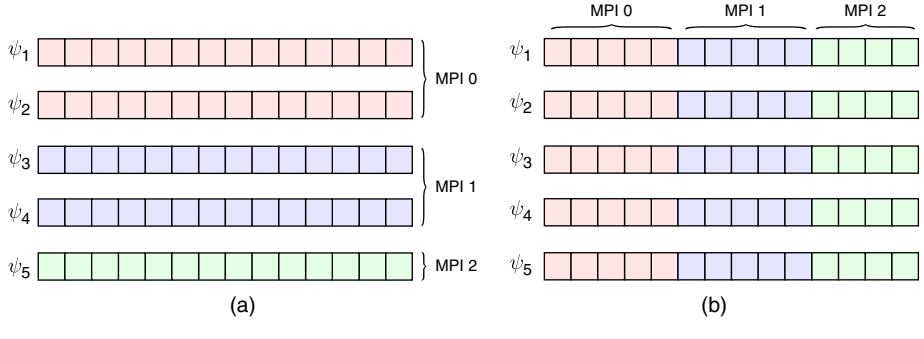


Figure 6.4 *Orbital (a) and coefficient (b) distribution schemes*

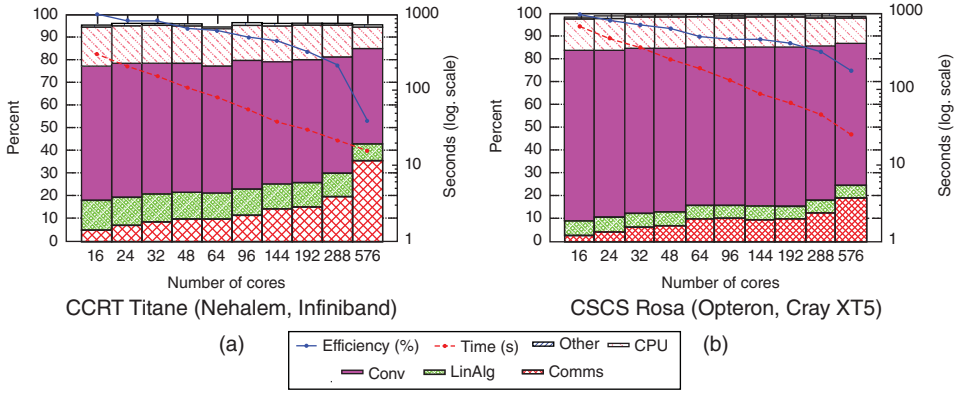


Figure 6.5 *Comparison of the performance of BigDFT on different platforms. Runs on the CCRT machine are worse in scalability but better in absolute performance per compute core than runs on the CSCS machine (1.6–2.3 times faster)*

Rosa Cray XT5 system. The latter shows better performance for the communication, and thus better scalability. However, in terms of absolute performance per compute core ("time-to-solution"), the former is 2 times faster. This is mainly related to better performance of the linear algebra libraries (Intel MKL compared to Istanbul linear algebra) and of the processor. These benchmarks are taken for a run of the BigDFT code with the same input files (a relatively small benchmark system of 128 atoms of ZnO), starting form the same sources. It is worth pointing out that, in this case, Figure 6.5 shows that parallel efficiency is not always a useful parameter to evaluate the performance of a code.

6.4 GPU Architecture

An overview of GPU architecture and introduction to GPU programming is provided in the first two chapters of this book. As explained in those chapters, GPUs are quite different from CPUs in many aspects, and understanding the peculiarities of their architecture is essential to write efficient GPU code. Here we first reiterate a few central points that are important to keep in mind before explaining details of the algorithms that are used to accelerate DFT calculations with wavelets basis using GPUs and analyzing the performance of the code in comparison to the CPU-based MPI/OpenMP implementation.

The first particularity of GPUs is that they are coprocessors controlled exclusively by a CPU. GPUs cannot be used alone and must have a CPU associated with them, thus forming a so-called hybrid architecture. GPUs can only access data that lie in their dedicated memory.[1] Data must be explicitly copied by the programmer between CPU memory and GPU memory. The GPU and the CPU are typically connected through the PCI-express link, which has a relatively small bandwidth and high latency.

In a GPU, execution units are organized into groups that form a multiprocessor. All the execution units in a multiprocessor share a control unit, so they perform the same instruction at the same time. To exploit the highly parallel nature of the processors, GPU programs use fine-grained threads or tasks. These are quite different from CPU threads, which are typically used when programming in parallel environments like OpenMP. In a GPU, the strategy is to have many more threads than execution units

[1] However, the specifications are continuously evolving and configurations with common memory may become a standard in the near future.

while assigning each thread a very small amount of work. As the GPU can switch threads without cost, it can hide the latency of the operations of one thread by processing other threads while waiting.

Memory access is also particular in GPUs. CPUs only have one type of memory and rely on caches to speed up the access to it. GPUs do not always have a cache, but instead they have a fast local memory shared by all the execution units in a multiprocessor. This local or private memory must be explicitly used by the programmer to store data that needs to be accessed frequently. Main memory access also needs to be done carefully. The memory bandwidth is higher than for a CPU, but latency is also higher. Moreover, to obtain maximum memory transfer rates, the execution units in a multiprocessor must access sequential memory locations.

In summary, the following rules of thumb need to be followed when developing a GPU implementation:

- Because of the high latency of the communication, the programmer should try to limit the data transfer between CPU and GPU as much as possible.
- The calculation workload should be parallelized in many little chunks, which perform the same operations on different data.
- Data locality is of great importance to achieve good performance, since different multiprocessors have different local memories.
- Memory access patterns should be as regular and homogeneous as possible.

6.4.1 GPU Implementation Using the OpenCL Language

A major issue concerning GPU programming is that the programming paradigms are sensitive to different architectures. In other words, a code should be reprogrammed to run on a different GPU. Different vendors provide programming languages suitable to their specific architecture, and the CUDA architecture and programming language of Nvidia [18] is undoubtedly the most advanced in terms of functionality and maturity.

We initially developed our GPU acceleration with CUDA [19]. However, when the OpenCL specification was released, we ported our code to this language. At the time of writing, the OpenCL BigDFT code is better optimized and complete than the older CUDA version.

OpenCL is an open standard defined by the *Khronos Group* [20]. It is aimed at cross-platform parallel computing, and GPUs are among the several types of devices defined by the OpenCL standard. The GPU device in OpenCL consists of several address spaces and a set of multiprocessors. OpenCL is aimed at data-parallel tasks and describes the work to be performed in terms of work groups that are composed of work items. When executing an OpenCL function (termed *kernel*), work items execute the same code. The difference between work items from different work groups is the visibility of the address spaces. The four address spaces are *global*, *local*, *private*, and *constant*. Each of these address spaces corresponds to a specific usage and has distinct characteristics. The *global* address space is shared among every work group. This address space is usually large, as it corresponds to the device on-board RAM. Accessing this address space is expensive, as latency is high, and should be done linearly rather than randomly as contiguous accesses can be coalesced into a single access. Synchronization using *global* memory can be achieved through the use of atomic operations but is expensive and should be avoided. The *local* address space is shared among all work items of a work group. *Local* memory is very fast (compared to *global*), and is organized in banks. Read and writes to local memory are simultaneous as long as they access different memory banks. Read operations are also simultaneous if work items read from the same address. Multiprocessors have only a small amount of shared memory, so it should be used wisely. Access to *private* memory is restricted to one work item, and is used for local variables and parameters. *Constant* memory is visible by all work groups, and is optimized for simultaneous reading from all work items. Content of *constant* memory cannot be modified after initialization.

It must be pointed out that, while OpenCL code can be executed without changes on different platforms (a GPU, a CPU, or other OpenCL supported device), this does not necessarily mean that the code optimized for one platform will run efficiently on other platforms. However, the appeal of a multiplatform language is evident: as OpenCL kernels are compiled *on board* and *at runtime*, the application has the freedom to choose the optimal kernel implementation that is available. This means that a single executable is able to be executed in parallel environments in which the machines contain different accelerators. As an example, we have executed a test run with BigDFT on two Intel Nehalem quad-core machines with one GPU each, one Nvidia Fermi and one ATI Radeon HD 2600 XT. Performance results are summarized in Table 6.1. The code runs as expected, exploiting both MPI parallelism and acceleration by the GPUs according to the particular behavior of the architecture. Clearly, different architectures have different features, and the compute kernels should be optimized in a different way for each. For the time being, the Nvidia optimized kernels are used in BigDFT, even for the ATI runs. However, since OpenCL kernels are compiled at runtime, some typical parameters can be extracted before generating the OpenCL binaries. This includes parameters such as the maximum number of threads per kernel execution (CUDA blocks), which are extracted at compilation time. The code execution thus becomes really heterogeneous, that is, each machine has at the end a different binary code.

6.4.2 Implementation Details of the Convolution Kernel

Since the convolution filters are separable, it can be shown that all the fundamental BigDFT convolutions can be brought to the following 1D kernel:

$$K_p(I,a) = \sum_j f_j G_{p-1}(a, I-j) + \alpha K_{p-1}(a,I) \quad \forall a, I \ , \tag{6.20}$$

possibly combined with a data transposition $G_p(I,a) = G_{p-1}(a,I)$. The filter f_j and the coefficient α are equal to T_j and 1 for the kinetic operator and ω_j and 0 for the magic filter, respectively.

From the GPU parallelism point of view, there is a set of N independent convolutions to be computed. Each of the lines of n elements to be transformed is split in chunks of size N_e. Each multiprocessor of the graphic card computes a group of N_ℓ different chunks and parallelizes the calculation on its computing units. The upper panel of Figure 6.6 shows the data distribution on the grid of blocks during the transposition. Input data (upper left panel) are ordered along the N-axis, while output (upper right panel) is ordered in n-axis direction, see Eq. (6.20). When executing the GPU convolution kernel, each block (i,j) of the execution grid is associated with a set of N_ℓ (N-axis) times N_e (n-axis) elements. The size of the data feed to each block is identical (so as to avoid block-dependent treatment); hence when N and n are not multiples of N_ℓ and N_e, some data treated by different blocks may overlap. This is indicated by the filled patterns in the figure. The light gray filling behind the block with the label (i,j) indicates the portion of data that needs to be copied to the shared memory for treating the data in this block.

The lower panel shows the data arrangement in shared memory for a given block. The number of lines N_ℓ is chosen to be a divisor of the half-warp size. Data are then treated in units of the

Table 6.1 *Parallel run of BigDFT on a hybrid heterogeneous architecture*

MPI +	1	1 +	4 +	1 +	4 +	4 + 4 +
Nvidia (NV)/ATI		NV	NV	ATI	ATI	NV + ATI
Execution time (s)	6200	300	160	347	197	109
Speedup	1	20.7	38.8	17.9	31.5	56.9

Two quad-core Intel Nehalem nodes are connected to two different GPU cards, one ATI and one Nvidia. The system is a four-carbon atom surface supercell with 52 k-points

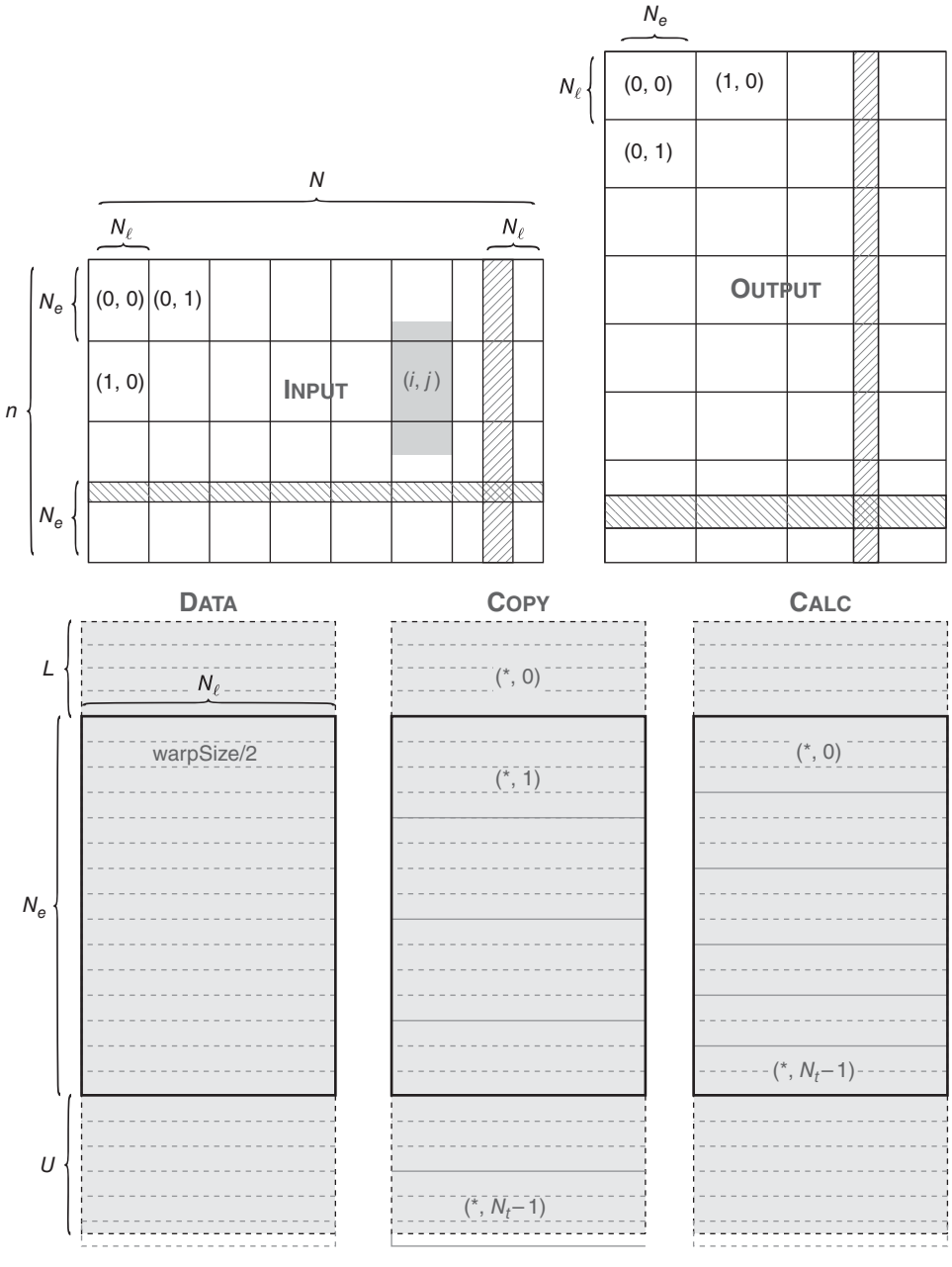


Figure 6.6 *Upper panel: Data distribution for 1D convolution+transposition on the GPU. See Section 6.4.2 for details. Lower panel: Reproduction of the portion of the input data highlighted in gray in the upper panel*

`warpSize`, which is the number of threads that can be executed simultaneously (in CUDA terminology). The thread index has (`warpSize/2`, N_t) elements, with $N_t \leq 16$ (left panel). Each group of threads $(*, i)$ of the half-warp i treats a definite number of elements, either for copying the data (center panel) or for performing the calculation (right panel). This data arrangement ensures that bank conflicts during shared memory access are avoided. For calculating the convolution, two buffers of sizes $N_\ell L$ and $N_\ell U$ must be created in shared memory.

In order to achieve the best performance with the GPU, it is strongly recommended to transfer data from the global to the shared memory of the multiprocessors. The shared memory must contain buffers to store the data needed for the convolution computations. The desired boundary conditions (periodic in our case) are implemented in shared memory during the data transfer. Each thread computes the convolution for a subset of N_e elements associated with the block. This data distribution is illustrated in Figure 6.6.

6.4.3 Performance of the GPU Convolution Routines

We have evaluated the performance of the GPU port of the 1D convolutions needed for the wavelet implementation of the local Hamiltonian operators and their 3D counterpart. For these evaluations, we used a computer with an Intel Xeon Processor X5472 (3 GHz) and a Nvidia Tesla S1070 card. The CPU version of BigDFT is highly optimized with optimal loop-unrolling and compiler options. The CPU code is compiled with the Intel Fortran Compiler (10.1.011) and the most aggressive compiler options (`-O2 -xT`). With these options, the magic filter convolutions run at about 3.4 GFLOPS, similar to what we have shown above. All benchmarks are performed with double-precision floating-point numbers.

The GPU versions of the 1D convolutions are 1–2 orders of magnitude faster than their CPU counterparts. We can then achieve an effective performance rate of the GPU convolutions of about 40 GFLOPS, by also considering the data transfers in the card. We are not close to peak performance since, on GPU, due to transpositions, a considerable fraction of time is still spent in data transfers rather than in calculations. This happens because data should be transposed between the input and the output array, and the arithmetic needed to perform convolutions is not heavy enough to hide the latency of all the memory transfers. However, we will later show that these results are really satisfactory for our purposes.

The performance graphs for the above-mentioned convolutions, together with the compression–decompression operator, are presented in Figure 6.7 as a function of the size of the corresponding 3D array. In addition, all required linear algebra operations can be executed on the GPU thanks to the CUBLAS routines or dedicated OpenCL kernels.

To build a three-dimensional operation, one must chain the corresponding one-dimensional GPU kernels 3 times. In this way, we obtain the three-dimensional wavelet transformations as well as the kinetic operator and the magic filter transformation (direct and transposed). The GPU speedup of the local density construction as well as the application of the local Hamiltonian and of the preconditioning operation is shown on the right in Figure 6.7 as a function of the compressed wave function size.

6.4.4 Three-Dimensional Operators, Complete BigDFT Code

Let us now examine the overall performance of the code. In Figure 6.8 we present an optimal case for GPU acceleration. For this example, most of the operations can be GPU-accelerated. The overall speedup of the full code due to GPUs is of about one order of magnitude and scaling with MPI processes is good, making it possible to achieve a speedup of two orders of magnitude using multiple nodes as compared to a single node run. This is really encouraging for challenging simulations on bigger hybrid machines.

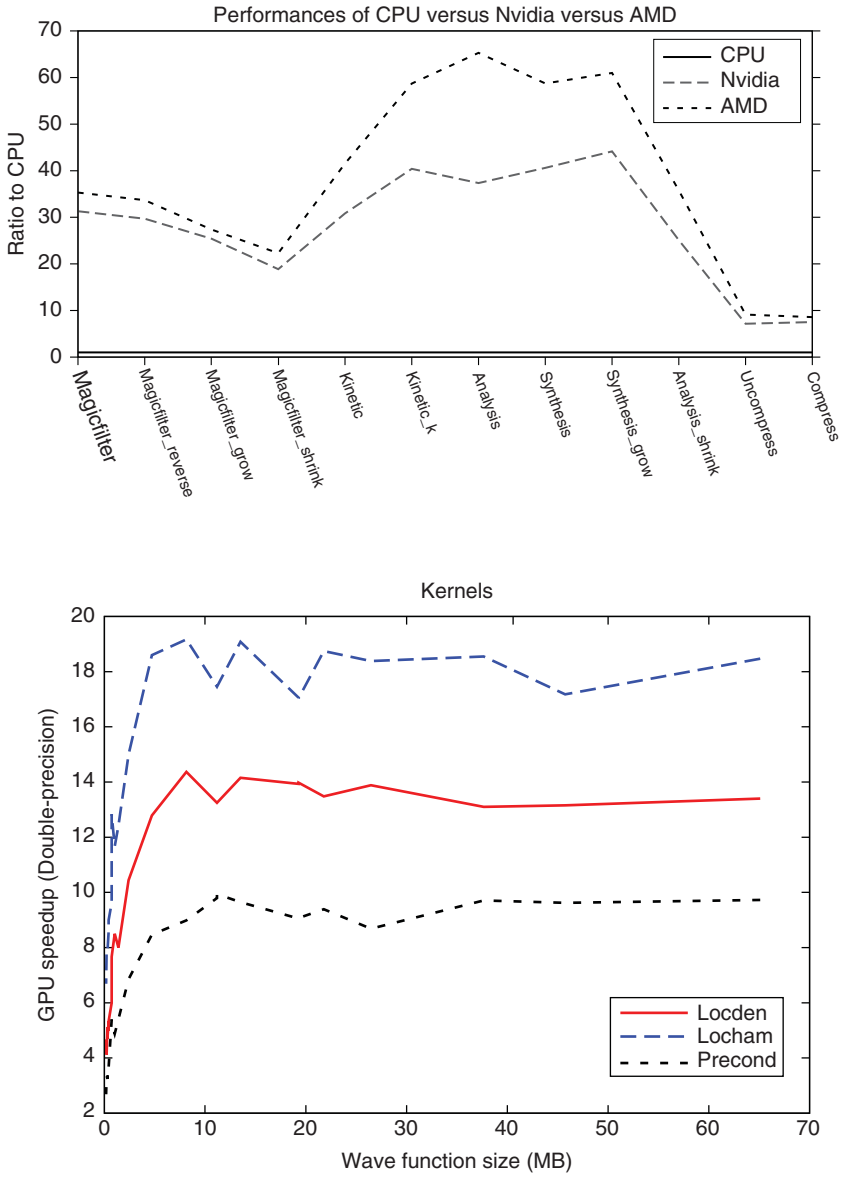
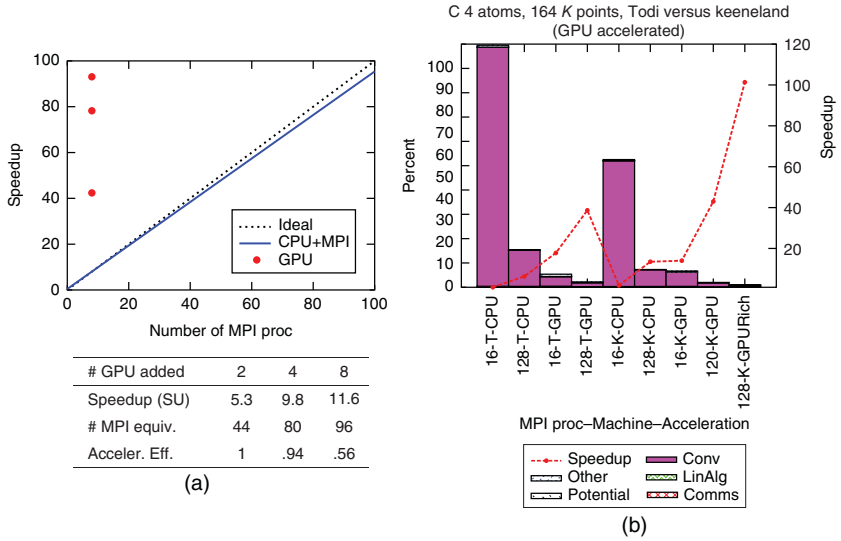


Figure 6.7 *Left panel: Speedup for the GPU version of the fundamental operations on the wave functions. Right panel: Double-precision speedup for the GPU version of the 3D operators used in the BigDFT code as a function of the single wave function size*

In Figure 6.9, simulations of different sizes have been run under different conditions. In particular, what has been tested is the response of the code in the case of an under-dimensioned calculation, where the amount of communication is of the same order as the computation. This may happen if the simulated system is too small, or if the ratio between the runtime GFLOPS of the computations and the cross-sectional bandwidth of the network is high.



# GPU added	2	4	8
Speedup (SU)	5.3	9.8	11.6
# MPI equiv.	44	80	96
Acceler. Eff.	1	.94	.56

(a) (b)

Figure 6.8 *(a) Speedup of the BigDFT code for a four-carbon-atom supercell (graphene) with 164 K-points. The calculation is performed with eight MPI processes on the CEA-DAM INTI machine, based on Westmere processors and Nvidia Fermi. For each run, the number of equivalent MPI processes is indicated, given that the parallel efficiency of this run is 98%. Also, the efficiency of the GPU acceleration is presented.(b) Speedup of the same run on different hybrid architectures in combination with MPI runs*

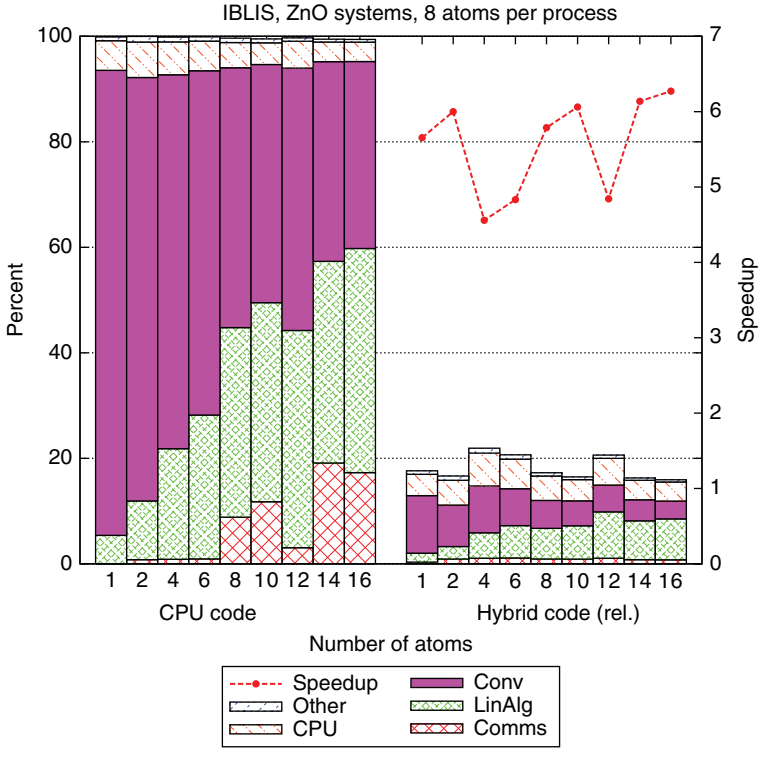


Figure 6.9 *Relative speedup of the hybrid DFT code with respect to the equivalent pure CPU run. Different runs for simulations of increasing system size have been performed on an Intel X5472 3 GHz (Harpertown) machine, with a Fermi GPU card*

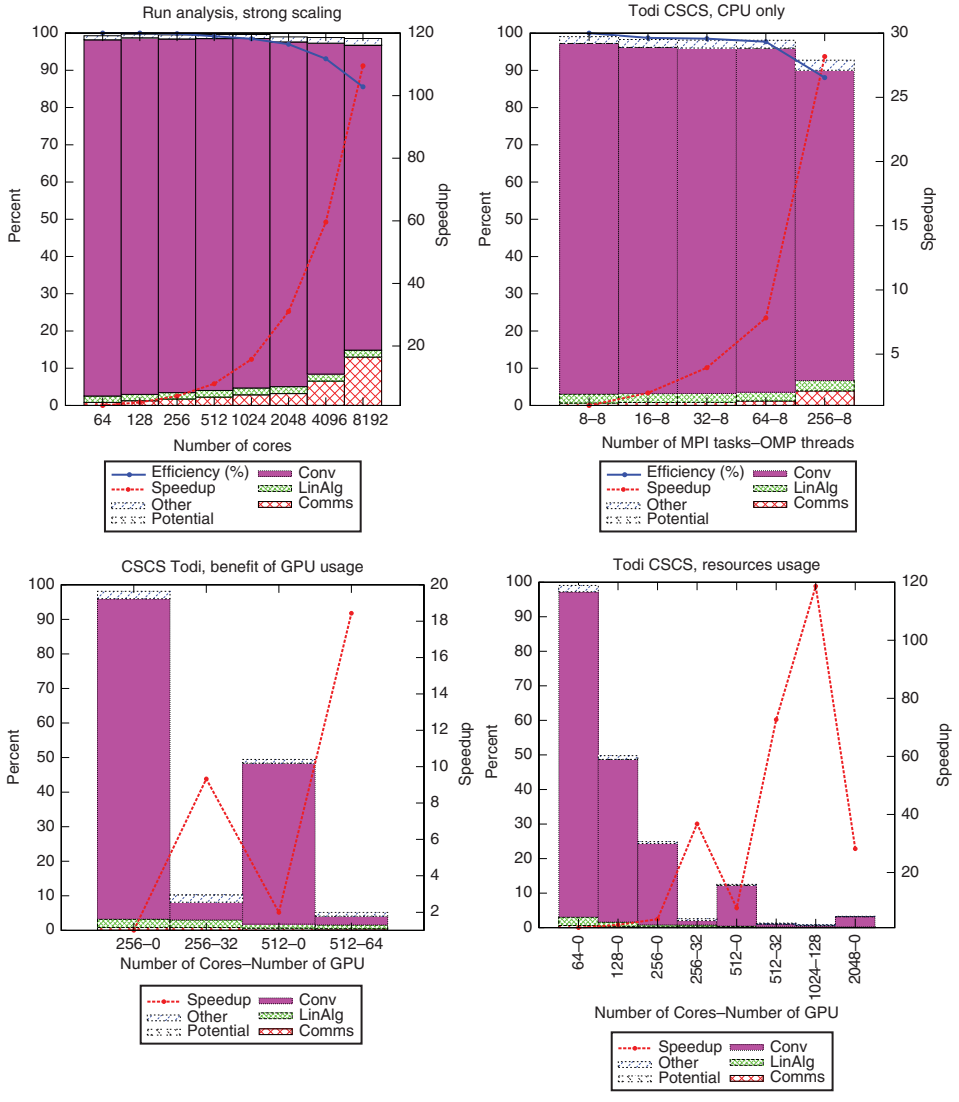


Figure 6.10 *Massively parallel runs for a cobalt metalloporphyrin on graphene sheet (265 atoms system) with surfaces BC. In the bottom panel, the simulations have been accelerated with Kepler GPU cards. Interesting speedups can be achieved*

Another interesting benchmark for a cobalt metalloporphyrin on a graphene surface, with and without GPU acceleration, is presented in Figure 6.10. In any case, the code appears to have a robust behavior even under nonoptimal conditions for the GPU acceleration. This is interesting because, for a basic usage, the end user is not required to understand in depth the optimal dimensioning of the simulated system for a given architecture.

6.4.5 Other GPU Accelerations

At present, our code accelerates not only the convolutions. We have already anticipated that BigDFT linear algebra operations can be ported to GPUs either explicitly (via OpenCL) or by using the CUBLAS library. In addition, the fast Fourier transforms (FFTs) used for the Poisson solver have recently been accelerated by a dedicated kernel based on the CuFFT library [15]. This is of great importance for advanced features such as the evaluation of the exact exchange operator. Strategies of automatic code generation at runtime for OpenCL FFTs are under investigation.

6.5 Conclusions and Outlook

DFT calculations using a wavelet basis set have been implemented for CPU and hybrid GPU/CPU systems in the BigDFT code, which is routinely used for production runs with applications in both physical and chemical sciences. In Table 6.2 we present the advantages of accelerating the BigDFT code in a multilevel parallelization framework, by giving the number of steps (and thus timescales) that are accessible in one day for an *ab initio* molecular dynamics simulations of 32 water molecules. Exploiting the power of GPUs on top of MPI and OpenMP parallelization significantly extends the accessible timescales.

As presented in this chapter, the numerical operations required for DFT calculations with wavelet basis sets are well suited for GPU acceleration. Indeed, on one hand the computational nature of 3D separable convolutions allows us to write efficient routines that benefit from the computational power of GPUs. On the other, the parallelization scheme of the BigDFT code is optimal in this sense: GPUs can be used without affecting the nature of the communications between the different MPI processes. This is in the same spirit of the multilevel MPI/OpenMP parallelization. Porting critical code sections to GPUs has been achieved within Kronos' OpenCL standard, which allows for multi-architecture acceleration. We therefore have at hand a multilevel parallelized code, combining MPI, OpenMP, OpenCL, and CUDA (the latter used for the FFT and linear algebra), which can work on state-of-the-art hybrid supercomputers. Thanks to the use of OpenCL, even heterogeneous architectures with different types of GPU accelerators can be exploited.

6.5.1 Evaluation of Performance Benefits for Complex Codes

Based upon our experience, we can express some general guidelines that should be of interest to someone who might want to use GPUs for scientific computing. For a code with the complexity of BigDFT, the evaluation of the benefits of using a GPU-accelerated code must be performed at three different levels.

Initially, one has to evaluate the effective speedup provided by the GPU kernels with respect to the corresponding CPU routines that perform the same operations. This is the "bare" speedup, which, of course, for a given implementation depends of the computational power provided by the device. It

Table 6.2 *Effect of MPI and OpenMP (OMP) parallelization and GPU acceleration on the complete molecular dynamics of 32* H_2O *molecules*

MPI*OMP + GPU	1*1	32*1	64*1	128*1	32*6	128*6	128*1 + 128
SCF iteration (s)	170	7.2	3.8	2.0	1.5	.44	.30
Force evaluation (s)	2,210	93.6	49.4	26.0	19.5	5.72	3.92
AIMD steps/day	40	923	1,749	3,323	4,430	15,104	22,040
MD ps/day	0.02	0.461	0.874	1.661	2.215	7.552	11.02

has to be kept in mind that vendors who do not know about the details of the full code *are able to provide only bare speedups*. For BigDFT, such results can be found in Figure 6.7.

At the second level, the "complete" speedup has to be evaluated; the performance of the complete hybrid CPU/GPU code should be analyzed with respect to execution on the reference CPU implementation. Clearly, this depends on the actual importance of the ported routines in the context of the whole code (i.e., following Amdahl's Law). This is the first reliable result of the actual performance enhancement of the GPU porting of the code. For a hybrid code that originates from a serial CPU program, this is the last level of evaluation.

However, for a parallel code, there is still another step that has to be evaluated. This is the behavior of the hybrid code in a parallel (or massively parallel) environment. Indeed, for parallel runs the picture is complicated by two aspects. The first is the management of the extra level of communication that is introduced by the PCI-express bus, which may interact negatively with the underlying code communication scheduling (MPI or OpenMP, for example). The second is the behavior of the code for a number of GPU devices, which is lower than the number of CPU processes that are running. In this case, the GPU resource is not homogeneously distributed and the management of this fact adds an extra level of complexity. The evaluation of the code at this stage yields the "user-level" speedup, which is the actual time-to-solution speedup, and thus the only speedup that is of relevance for practical purposes.

We believe these rules of thumb will be be useful for any developer of complex codes like the ones typically used in scientific computing nowadays.

References

1. Kohn, W. and Sham, L. (1965) Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, **140**, A1133–A1138.
2. Genovese, L., Neelov, A., Goedecker, S., Deutsch, T., Ghasemi, S.A., Willand, A. *et al*. (2008) Daubechies wavelets as a basis set for density functional pseudopotential calculations. *J. Chem. Phys.*, **129** (1), 014109.
3. Yang, J., Wang, Y. and Chen, Y. (2007) {GPU} accelerated molecular dynamics simulation of thermal conductivities. *J. Comput. Phys.*, **221** (2), 799–804.
4. Anderson, A.G., Goddard, W.A. III and Schröder, P. (2007) Quantum Monte Carlo on graphical processing units. *Comput. Phys. Commun.*, **177** (3), 298–306.
5. Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S.H.M., Grajewski, M. *et al*. (2007) Exploring weak scalability for {FEM} calculations on a GPU-enhanced cluster. *Parallel Comput.*, **33** (10–11), 685–699.
6. Goedecker, S. (1998) *Wavelets and their Application for the Solution of Partial Differential Equations in Physics*, Presses Polytechniques Universitaires et Romandes, Lausanne.
7. Neelov, A.I. and Goedecker, S. (2006) An efficient numerical quadrature for the calculation of the potential energy of wavefunctions expressed in the Daubechies wavelet basis. *J. Comput. Phys.*, **217** (2), 312–339.
8. Johnson, B.R., Modisette, J.P., Nordlander, P.J. and Kinsey, J.L. (1999) Quadrature integration for orthogonal wavelet systems. *J. Chem. Phys.*, **110** (17), 8309–8317.
9. Goedecker, S., Teter, M. and Hutter, J. (1996) Separable dual-space Gaussian pseudopotentials. *Phys. Rev. B*, **54**, 1703–1710.
10. Hartwigsen, C., Goedecker, S. and Hutter, J. (1998) Relativistic separable dual-space Gaussian pseudopotentials from H to Rn. *Phys. Rev. B.*, **58**, 3641–3662.
11. Krack, M. (2005) Pseudopotentials for H to Kr optimized for gradient-corrected exchange-correlation functionals. *Theor. Chem. Acc.*, **114** (1–3), 145–152.

12. Deslauriers, G. and Dubuc, S. (1989) Symmetric iterative interpolation processes. *Constr. Approx.*, **5** (1), 49–68.
13. Genovese, L., Deutsch, T., Neelov, A., Goedecker, S. and Beylkin, G. (2006) Efficient solution of Poisson's equation with free boundary conditions. *J. Chem. Phys.*, **125** (7), 074105.
14. Genovese, L., Deutsch, T. and Goedecker, S. (2007) Efficient and accurate three-dimensional Poisson solver for surface problems. *J. Chem. Phys.*, **127** (5), 054704.
15. Dugan, N., Genovese, L. and Goedecker, S. (2013) A customized 3D {GPU} Poisson solver for free boundary conditions. *Comput. Phys. Commun.*, **184** (8), 1815–1820.
16. Goedecker, S. and Hoisie, A. (2001) *Performance Optimization of Numerically Intensive Codes*, SIAM Publishing Company, Philadelphia, PA.
17. Genovese, L., Videau, B., Ospici, M., Deutsch, T., Goedecker, S. and Méhaut, J.F. (2011) Daubechies wavelets for high performance electronic structure calculations: the BigDFT project. *C. R. Mécanique.*, **339** (2–3), 149–164.
18. Nvidia CUDA Programming Guide, `http://www.nvidia.com/object/cuda_home.html` (accessed 18 September 2015).
19. Genovese, L., Ospici, M., Deutsch, T., Méhaut, J.F., Neelov, A. and Goedecker, S. (2009) Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *J. Chem. Phys.*, **131** (3), 034103.
20. Khronos Group (2009) The OpenCL Standard, `http://www.khronos.org/opencl/` (accessed 18 September 2015).

7

Plane-Wave Density Functional Theory

Maxwell Hutchinson[1], *Paul Fleurat-Lessard*[2,3], *Ani Anciaux-Sedrakian*[4], *Dusan Stosic*[5], *Jeroen Bédorf*[6] *and Sarah Tariq*[7]

[1]*Department of Physics, University of Chicago, Chicago, IL, USA*
[2]*Laboratoire de Chimie, Université de Lyon, ENS Lyon, Lyon, France*
[3]*ICMUB, Université de Bourgogne Franche-Comté, Dijon, France*
[4]*Mechatronics, Computer Sciences and Applied Mathematics Division, IFP Energies nouvelles, Rueil-Malmaison Cedex, France*
[5]*Department of Computer Science, Federal University of Pernambuco, Recife, Brazil*
[6]*Centrum Wiskunde & Informatica, Amsterdam, The Netherlands*
[7]*NVIDIA Corporation, Santa Clara, CA, USA*

In this chapter, we describe how density functional theory (DFT) calculations with plane-wave (PW) basis sets can be accelerated using GPUs. We review the steps that are required to obtain the electronic structure of condensed-phase systems using a plane-wave basis set both for standard, explicit density functionals and hybrid functionals with exact exchange. We then discuss the numerical implementation of the different energy components as well as ionic forces and stress tensors with an eye towards GPU and more general coprocessor architectures. Specific optimizations for GPUs suitable for hybrid multicore and multi-GPU platforms are discussed along with code samples. Benchmarks for typical simulation setups for energy, band structure, *ab initio* molecular dynamics and structure relaxations demonstrate the performance improvements that can be achieved with GPUs.

7.1 Introduction

DFT [1, 2] is one of the most widely used techniques for computing the electronic structure in physics [3] and chemistry, [4–7] and in particular of condensed matter systems. DFT implementations are realized on a variety of bases [3], which can be broadly categorized as orbital-like or grid-like. Orbital

Electronic Structure Calculations on Graphics Processing Units:
From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

bases, most commonly *linear combinations of atomic orbitals* (LCAO) [8], attempt to concisely represent the Hilbert space using physically motivated, atom-centered monoelectronic functions, for example, model solutions of the Schrödinger equation. Their primary advantages are their small size and sparse overlap, which facilitate the development of low-order methods. However, they run the risk of poorly representing unexpected processes in regions not covered by model solutions [9]. Grid bases, either in real or reciprocal space, systematically resolve the Hilbert space. Their primary advantage is uniform convergence. However, they are much larger than orbital bases and are often nonlocal. If the primary representation of wavefunctions is on a reciprocal grid, then the basis is referred to as *planewave* (PW).

Plane-wave density functional theory (PWDFT) is one of the most popular schemes for performing electronic structure calculations of condensed matter systems. In PWDFT, the kinetic energy operator is expressed as a diagonal matrix in plane-wave space (g-space), while the local potential is expressed as a diagonal matrix in real space (r-space). In most cases, pseudopotentials are used to smooth the wavefunctions near atom centers, reducing the required number of planewaves (g-vectors) at the cost of a low-rank nonlocal potential.

The combined r-space and g-space representation of the Hamiltonian necessitates the use of matrix-free eigensolvers that rely on the action of the Hamiltonian on a vector $|\phi\rangle$, $H|\phi\rangle$, rather than the elements of the Hamiltonian, $\langle r_1|H|r_2\rangle$ or $\langle g_1|H|g_2\rangle$. In computing the action on a vector $|\phi\rangle$, fast Fourier transforms (FFTs) are used to switch between r-space, $\langle r|\phi\rangle$, and g-space, $\langle g|\phi\rangle$, representations of the vector $|\phi\rangle$. The nonlocal potential is defined on subspaces that, in realspace, do not grow with the overall system size. This makes the action of the Hamiltonian formally $O(n \log n)$, instead of $O(n^2)$ as for a general matrix. This is the key feature of the PWDFT method.

Implementations of PWDFT can be further categorized by the type or types of pseudopotentials that they employ. Modern pseudopotentials come in three popular varieties: norm-conserving pseudopotentials (NCPP) [10], ultra-soft pseudopotentials (USPP) [11, 12], and projector-augmented wave (PAW) [13]. NCPPs only add a dense low-rank term to the Hamiltonian, making them the simplest. USPPs significantly reduce the grid resolution necessary to converge the electron density, but require the introduction of a nontrivial overlap matrix, which generalizes the eigenproblem and adds charge density dependence to the nonlocal potential. PAW is a generalization of USPP to arbitrary projector forms, but is very similar in practice and provides only modest improvements to the rate of grid convergence.

There are many implementations of PWDFT, including Quantum Espresso [14], VASP [15], CASTEP [16, 17], and Qbox [18, 19]. This chapter will draw examples from the GPU ports of VASP.

7.2 Theoretical Background

7.2.1 Self-Consistent Field

Kohn–Sham DFT is fundamentally an eigenproblem: $\hat{H}\psi = \epsilon\psi$, where $\hat{H}$ is the Hamiltonian operator, ψ is a single-particle wavefunction, and ϵ is the energy eigenvalue of ψ. In the pseudopotential framework, the Hamiltonian takes the form

$$\hat{H} = \hat{T} + \hat{V}^{\text{loc}} + \hat{V}^{\text{nl}}, \tag{7.1}$$

where $\hat{T} = -\frac{1}{2}\nabla^2$ is the kinetic energy operator, $\hat{V}^{\text{nl}}$ is the nonlocal potential, and $\hat{V}^{\text{loc}}$ is the local potential due to both ion–electron and electron–electron interactions.

Bloch's theorem decomposes the wavefunctions as

$$u_{nk} = \frac{1}{\sqrt{\Omega}} \sum_j c_{jnk} e^{ig_j \cdot r}, \psi_{nk} = e^{ik \cdot r} u_{nk} = \frac{1}{\sqrt{\Omega}} \sum_j c_{jnk} e^{i(k+g_j) \cdot r}, \tag{7.2}$$

where n is an index to order the wavefunctions, the vector k is a Brillouin zone (BZ) sample or *k-point*, g is a g-vector of a planewave, and Ω is the volume of the unit cell. For clarity's sake, the normalization factor Ω^{-1} will be omitted in the following. Using this decomposition, the Hamiltonian is diagonal with respect to the k-point. The k-dependent Hamiltonian is defined as a function of the k-block of the full Hamiltonian:

$$\hat{H}(k) = e^{-ik \cdot r} \hat{H} e^{ik \cdot r}, \tag{7.3}$$

which converts the eigenproblem into:

$$\hat{H}(k) u_{nk} = \epsilon_{nk} u_{nk} \,. \tag{7.4}$$

Writing in the bases reveals the k-point dependence:

$$H(k) = |g\rangle T(k) \langle g| + e^{-ik \cdot r} |r\rangle V^{\text{loc}} \langle r| e^{ik \cdot r} + e^{-ik \cdot r} |\beta'\rangle V^{\text{nl}} \langle \beta| e^{ik \cdot r} \tag{7.5}$$

where $H(k)$ is a representation of the operator $\hat{H}(k)$, g runs over the plane-wave basis, r runs over the real-space basis, and β, β' run over the pseudopotential projectors. The k-dependence of the local potential cancels. The k-dependence of the kinetic energy has a simple analytic form. The k-dependence of the nonlocal potential is wrapped into the projectors as $\beta(k)$. This results in the expression for the k-dependent Hamiltonian:

$$H(k) = \left| g \right\rangle \frac{1}{2} |g + k|^2 \left\langle g \right| + |r\rangle V^{\text{loc}} \langle r| + |\beta'(k)\rangle V^{\text{nl}} \langle \beta(k)|. \tag{7.6}$$

The local potential is a functional, $V^{\text{loc}}[\rho(r)]$, of the electron density:

$$\rho(r) \equiv \langle r|\rho|r\rangle = \sum_{nk} f_{nk} \langle r|u_{nk}\rangle \langle u_{nk}|r\rangle \tag{7.7}$$

where $f_{nk} \equiv f_{nk}[\epsilon]$, the *occupancy*, is a function of the entire eigenvalue spectrum ϵ_{nk}. Therefore, the electron density depends explicitly on all u_{nk} and ϵ_{nk}. The occupancy is a decreasing function in n, which is nonzero for only the order n_e lowest eigenpairs, where n_e is the number of electrons. The functionals used to produce the local and nonlocal potentials contain nonlinear terms. The application of the functionals themselves is generally not performance-critical, so they will not be further described here.

Explicitly including the dependence on dynamical variables yields

$$(|g + k|^2 + V^{\text{loc}}[\rho[u, \epsilon]] + V^{\text{nl}})|u_{nk}\rangle = \epsilon_{nk}|u_{nk}\rangle. \tag{7.8}$$

To resolve the cyclic dependence on the eigenpairs (u, ϵ), a *self-consistent* technique is employed: the calculation begins with a guess of the electron density ρ, produces the resulting Hamiltonian, diagonalizes that Hamiltonian, and uses the resulting wavefunctions and energies to refine the density. This back and forth operation between diagonalizing the Hamiltonian and producing a better estimate of the density continues until the electron density converges. Because there are decaying errors in the density from iteration to iteration, the diagonalization at each step need not be complete. This motivates the use of an iterative diagonalization scheme.

7.2.2 Ultrasoft Pseudopotentials

In order to reduce the number of plane waves, actual calculations resort to pseudopotentials: close to each ion, the exact wavefunction is replaced by pseudo-wavefunctions centered on the ion.

The radial part of the pseudo-wavefuntions is nodeless. These pseudo-wavefunction are localized in r-space and will be denoted by $\{|\tilde{\theta}_i\rangle\}$. Projection on these pseudo-wavefunctions are denoted by $\{\langle\beta_i|\}$, which are the dual vectors of $\{|\tilde{\theta}_i\rangle\}$. The β_i functions of each type of atom are defined with respect to the displacement from an atomic center and vanish outside of a core region:

$$\beta_i(r) = \beta(r - R_i), \quad \beta(r > r_c) = 0,$$

where R_i is the position of the ion that β_i is centered around and r_c is the cut-off radius.

With USPPs, the tensor Q connects diagonal real-space 2-tensors, such as the density ρ, to pairs of projectors [20]:

$$\langle\psi|r\rangle\langle r|\psi'\rangle \rightarrow \sum_{r'}\langle\psi|r'\rangle\left(\left\langle r'|r\right\rangle\langle r|r'\rangle + \sum_{i,j}\langle r'|\beta_i\rangle Q_{i,j}(r)\langle\beta_j|r'\right\rangle\right)\langle r'|\psi'\rangle, \tag{7.9}$$

where ψ, ψ' are arbitrary vectors. Integrating over r yields an overlap matrix

$$S = I + \sum_{i,j}|\beta_i\rangle Q_{i,j}\langle\beta_j|, \tag{7.10}$$

which defines the inner product $\langle\phi|\psi\rangle \rightarrow \langle\phi|S|\psi\rangle$, where $Q_{i,j} = \sum_r Q_{i,j}(r)$. The eigenproblem is therefore generalized to

$$H(k)|u_{nk}\rangle = \epsilon_{nk}S|u_{nk}\rangle. \tag{7.11}$$

The change in inner product leads to an additional term in the charge density:

$$\langle r|\rho|r\rangle = \sum_{n,k} f_{nk}\left(\left\langle r|u_{nk}\right\rangle\langle u_{nk}|r\rangle + \sum_{i,j}\langle u_{nk}|\beta_i\rangle Q_{i,j}(r)\langle\beta_j|u_{nk}\right\rangle\right) \tag{7.12}$$

It also leads to the density-dependent nonlocal potential $V^{\text{nl}}[\rho[u,\epsilon]]$, which adds to the cyclic dependence of Eq. (7.8). In the USPP scheme, V^{nl} is

$$V^{\text{nl}} = \sum_{i,j}|\beta_i\rangle D_{i,j}[\rho]\langle\beta_j|, \tag{7.13}$$

where the matrix $D_{i,j}$ depends on the density ρ.

The rest of this chapter operates with the generality of USPP. The NCPP expressions are recovered by letting $Q \rightarrow 0$, which implies $S \rightarrow I$ and reduces the generalized eigenvalue problem to a standard one.

7.2.3 Projector Augmented Wave (PAW) Method

The PAW scheme [13] is a generalization of the USPP scheme [12] introduced in Section 7.2.2. In the PAW scheme, the wavefunction is related to smooth pseudo-wavefunctions by a linear transformation [13]:

$$|\psi\rangle = |\tilde{\psi}\rangle + \sum_i(|\phi_i\rangle - |\tilde{\phi}_i\rangle)\langle\tilde{p}_i|\tilde{\psi}\rangle, \tag{7.14}$$

where the ϕ are generally taken to be solutions to the radial Schrödinger equation for an isolated atom and tildes mark pseudo-quantities. Computationally, and USPP take very similar forms. The relationship between the two is discussed more formally in the literature [12]. Here, we maintain consistency with the notation used for USPPs by letting $\tilde{p} \equiv \beta$.

7.2.4 Force and Stress

Forces are computed by differentiating the total energy function with respect to ionic positions:

$$F_a^\mu = -\frac{\partial E[\rho]}{\partial R_a^\mu}, \tag{7.15}$$

where μ indexes the spatial components and a indexes the atomic centers.

The total energy is equal to the sum of the electronic energy from the Hamiltonian in Eq. (7.1) and the electrostatic energy due to the ions. This last term will be denoted by $U\{R_I\}$:

$$\begin{aligned} E_{\text{tot}}(\{f\},\{u\},\{R_I\}) =& \frac{1}{N_k}\sum_{n,k} f_{nk}\langle u_{nk}|T+V^{\text{nl}}|u_{nk}\rangle + \frac{1}{2}\int\int dr\, dr' \frac{\rho(r)\rho(r')}{|r-r'|} \\ &+ E_{xc}[\rho] + \int dr\, V_{\text{ion}}^{\text{loc}}(r)\rho(r) + U\{R_I\}, \end{aligned} \tag{7.16}$$

where the local potential is separated into its electron–electron Coulombic (Hartree), electron–electron exchange–correlation, and electron–ion Coulombic contributions. Let us introduce $V_{\text{eff}}(r)$, $D_{i,j}^{\text{ion}}$, and the onsite density $\rho_{i,j}$:

$$V_{\text{eff}}(r) = \frac{\partial E_{\text{tot}}}{\partial \rho} = V_{\text{ion}}^{\text{loc}}(r) + \int dr' \frac{\rho(r')}{|r-r'|} + \frac{\partial E_{xc}[\rho]}{\partial \rho}, \tag{7.17}$$

$$D_{i,j}^{\text{ion}} = D_{i,j} + \int dr\, V_{\text{eff}}(r) Q_{i,j}(r), \tag{7.18}$$

$$\rho_{i,j} = \sum_{n,k} f_{nk}\langle u_{nk}|\beta_i\rangle\langle\beta_j|u_{nk}\rangle. \tag{7.19}$$

Using these definitions, the total energy can be written as

$$\begin{aligned} E_{\text{tot}}(\{f\},\{u\},\{R_I\}) =& \frac{1}{N_k}\sum_{n,k} f_{nk}\langle u_{nk}|T|u_{nk}\rangle \\ &+ \int dr\, V_{eff}(r)\rho(r) + \frac{1}{N_k}\sum_{i,j} D_{i,j}\rho_{i,j} + U\{R_I\}. \end{aligned} \tag{7.20}$$

leading to the following expression for the forces:

$$\begin{aligned} F_a^\mu =& -\frac{\partial U\{R_I\}}{\partial R_a^\mu} \\ &- \int dr\, \frac{\partial V_{\text{ion}}^{\text{loc}}(r)}{\partial R_a^\mu}\rho(r) \\ &- \int dr V_{\text{eff}}(r) \sum_{i,j} \frac{\partial Q_{i,j}(r)}{\partial R_a^\mu}\rho_{i,j} \\ &- \sum_{i,j} D_{i,j}^{\text{ion}} \sum_{n,k} f_{nk}\left[\left\langle u_{nk}|\frac{\partial \beta_i}{\partial R_a^\mu}\right\rangle\langle\beta_j|u_{nk}\rangle + \langle u_{nk}|\beta_i\rangle\left\langle\frac{\partial\beta_j}{\partial R_a^\mu}|u_{nk}\right\rangle\right], \end{aligned} \tag{7.21}$$

where we have used

$$\frac{\partial\rho(r)}{\partial R_a^\mu} = \sum_{n,k} f_{nk}\sum_{i,j}\frac{\partial Q_{i,j}(r)}{\partial R_a^\mu}\langle\beta_i|u_{nk}\rangle\langle u_{nk}|\beta_j\rangle + Q_{i,j}(r)\frac{\partial(\langle\beta_i|u_{nk}\rangle\langle u_{nk}|\beta_j\rangle)}{\partial R_a^\mu}. \tag{7.22}$$

Another interesting quantity is the stress tensor. Let us denote by $\mathbf{a_1}, \mathbf{a_2}$, and $\mathbf{a_3}$ the unit cell lattice vectors. We combine them in $\mathbf{h} = [\mathbf{a_1}, \mathbf{a_2}, \mathbf{a_3}]$. With these notations, the stress components are

$$\sigma_{\mu\nu} = -\frac{1}{\Omega} \sum_s \frac{\partial E_{\text{tot}}}{\partial h_{\mu s}} h^T_{s\nu} \tag{7.23}$$

where μ and ν denote the Cartesian indices.

As the interaction energy between electrons and ions depends only on their distances, one can recast the previous equation into [3]

$$\sigma_{\mu\nu} = -\left\langle \Psi \left| \sum_p \frac{1}{2m_p} \nabla_{p,\mu} \nabla_{p,\nu} - \frac{1}{2} \sum_{p \neq p'} \frac{\left(r_{pp'}\right)_\mu (r_{pp'})_\nu}{|r_p - r_{p'}|} \frac{\partial V}{\partial r_{pp'}} \right| \Psi \right\rangle, \tag{7.24}$$

where the sum over p and p' runs over *all* particles (nuclei *and* electrons) and $(r_{pp'})_\mu$ is the μ Cartesian coordinate of $r_{pp'}$, the vector joining particles p and p'.

The pseudopotentials introduce a dependence on the wavefunctions. The number of wavefunction terms grows with the system size, so the computation of those terms dominates the cost of computing forces and stresses.

Efficient evaluation of the stress tensor will not be further discussed here. Interested readers are redirected to, for example, [3] or [21].

7.2.5 Iterative Diagonalization

In the self-consistent scheme, the Hamiltonian itself is converging along with the charge density. Therefore, each self-consistent iteration does not require the exact diagonalization of the Hamiltonian so long as the eigensolutions are converging as quickly as the Hamiltonian. Furthermore, we are interested in the lowest N_e eigenvalues and associated eigenfunctions, as only the $N_e \gtrsim n_e$ lowest eigenfunctions contribute to the charge density. Directly diagonalizing the Hamiltonian would be a waste of effort.

Iterative procedures provide an elegant solution to this problem: the eigenproblem can be solved in a subset (generally called the *expansion set*) [22] that is much smaller than the plane-wave basis. Any eigensolutions of the full problem will also be eigensolutions in the expansion space. There exist several common iterative approaches for PWDFT:

- the conjugate gradient algorithm [23, 24],
- the blocked Davidson scheme [25–28], and
- the residual minimization scheme with direct inversion in iterative subspace (RMM-DIIS) [15, 22, 29].

Thorough reviews of commonly used iterative methods can be found in Refs [15, 22].

The expansion set is built iteratively so that the eigenvectors and eigenvalues in the expansion space are the best approximations of the eigenvectors and eigenvalues in the full space. One usually distinguishes the blocked algorithms that work on many bands at the same time from the unblocked algorithms that optimize each band sequentially. In either case, the key step is to compute the vectors that are added to the expansion set to ensure that the lowest eigensolutions of the expansion set converge to the actual eigensolutions. Let us denote by $|\hat{\phi}_n\rangle$ the approximation of the eigenvector $|\phi_n\rangle$. A central quantity of most iterative schemes is the residual vector $|R(\hat{\phi}_n)\rangle$:

$$|R(\hat{\phi}_n)\rangle = (H - \epsilon_{\text{app}} S)|\hat{\phi}_n\rangle,$$

where ϵ_{app} is the approximation of the eigenvalue, defined as the Rayleigh quotient

$$\epsilon_{\text{app}} = \frac{\langle \hat{\phi}_n | H | \hat{\phi}_n \rangle}{\langle \hat{\phi}_n | S | \hat{\phi}_n \rangle}. \tag{7.25}$$

The perfect correction to $\hat{\phi}_n$ is thus

$$|\delta\phi_n\rangle = -(H - \epsilon_{\text{app}} S)^{-1} |R(\hat{\phi}_n)\rangle. \tag{7.26}$$

because

$$|R(\hat{\phi}_n + \delta\phi_n)\rangle = 0. \tag{7.27}$$

However, computing $(H - \epsilon_{\text{app}} S)^{-1}$ is as complex as directly diagonalizing H, so one has to resort to approximate inverses:

$$|\delta\phi_n\rangle \approx K |R(\hat{\phi}_n)\rangle, \tag{7.28}$$

where K is called the preconditioning matrix and building $|\delta\phi_n\rangle$ from the residual vector $|R(\hat{\phi}_n)\rangle$ is called the preconditioning. For PWDFT, K is usually diagonal and therefore easy to compute and apply. There are many specific approaches [29–31], which will not be further discussed here.

7.2.5.1 Conjugate Gradient (CG)

The sequential conjugate gradient approach is an unblocked scheme in which the expansion set is restricted to only two vectors for each eigenfunction. Consider the minimization of eigenfunction n. At each iteration, the new vector, called the *search vector* for CG, is chosen to be orthogonal to the previous one. The Hamiltonian is then directly diagonalized in the expansion space composed of the approximate eigenvector $|\hat{\phi}_n\rangle$ and the search vector $|f_n\rangle$, producing a revised approximate eigenvector $|\hat{\phi}'_n\rangle$. A new search direction is then constructed:

$$|f'_n\rangle = |P(\hat{\phi}'_n)\rangle + \frac{\langle P(\hat{\phi}'_n) | R(\hat{\phi}'_n) \rangle}{\langle P(\hat{\phi}_n) | R(\hat{\phi}_n) \rangle} |f_n\rangle, \tag{7.29}$$

where $|P(\hat{\phi}_n)\rangle$ is the orthogonalized, preconditioned residual vector:

$$|P(\hat{\phi}_n)\rangle = \left(1 - \sum_m |\phi_m\rangle \langle \phi_m | S\right) K (H - \epsilon_{\text{app}} S) |\hat{\phi}_n\rangle, \tag{7.30}$$

where m runs over the previously computed lower energy eigenvectors. This orthogonalization is necessary to ensure that all bands do not converge to the eigenvector associated with the lowest eigenvalue.

7.2.5.2 Blocked Davidson

In the blocked Davidson scheme, a single expansion set, $\{|\varphi_i\rangle\}$, is used to compute all the lowest eigenvectors concurrently. The first step of each iteration is to construct the Hamiltonian and overlap matrices in the expansion space.

$$\hat{H}_{i,j} = \langle \varphi_i | H | \varphi_j \rangle, \hat{S}_{i,j} = \langle \varphi_i | S | \varphi_j \rangle. \tag{7.31}$$

The resulting expansion space eigenproblem is then solved directly to produce trial vectors and Rayleigh quotients. Diagonalizing the Hamiltonian in the expansion set is called the Rayleigh–Ritz Scheme [32] or "subspace rotation."

$$\hat{H}\hat{\phi}_n = \epsilon_{\text{app}}\hat{S}\hat{\phi}_n. \tag{7.32}$$

The residuals of the trial vectors with the lowest Rayleigh quotients are then computed in the full space. Based on the norm of the residual or the difference in the Rayleigh quotient from the last iteration, some or all of the residuals are preconditioned and added to the next expansion set.

The expansion set grows with each iteration, but only needs to be strictly larger than the desired number of eigenvectors. Therefore, many schemes limit the size of the expansion set to some small multiple of the number of desired eigenvectors [15, 29]. When adding preconditioned residuals would expand the set beyond that limit, the trial vectors with the highest Rayleigh quotients are removed to make room.

The blocked Davidson method does not require explicit orthonormalization because the trial vectors are always orthogonal in both the expansion space and the full space.

7.2.5.3 RMM-DIIS

The RMM-DIIS method optimizes one band at a time, but can be parallelized over bands. It is based on the original work of Pulay [33] applied here to minimize the norm of the residual vector. In this scheme, the expansion set is called the *iterative subspace* following Pulay. The iterative subspace consists of the previous approximations of the eigenvectors $\{|\hat{\phi}_n^i\rangle\}$, where the n index is over eigenvectors and the i index is over iterations. Each iteration begins by constructing a trial vector $|\hat{\phi}_n'\rangle$ as a linear combination of the previous approximate eigenvectors:

$$|\hat{\phi}_n'\rangle = \sum_i |\hat{\phi}_n^i\rangle\alpha_i. \tag{7.33}$$

The coefficients α_i are chosen in order to minimize the relative norm of the residual vector:

$$\epsilon = \frac{\langle R(\hat{\phi}_n')|R(\hat{\phi}_n')\rangle}{\langle\hat{\phi}_n'|S|\hat{\phi}_n'\rangle}. \tag{7.34}$$

The residual can be expanded with respect to the residuals of the previous eigenvectors:

$$|R(\hat{\phi}_n')\rangle = \sum_i |R(\hat{\phi}_n^i)\rangle\alpha_i \tag{7.35}$$

creating an eigensystem

$$\sum_j \langle R(\hat{\phi}_n^i)|R(\hat{\phi_n^j})\rangle\alpha_j = \epsilon\sum_j \langle\hat{\phi}_n^i|S|\hat{\phi}_n^j\rangle\alpha_j, \tag{7.36}$$

where the lowest eigenvector becomes the trial vector $|\hat{\phi}_n'\rangle$. A linear combination of the trial vector and its preconditioned residual is then added as the next approximate eigenvector: $|\hat{\phi}_n^{i+1}\rangle = |\hat{\phi}_n'\rangle + \lambda K|R(\hat{\phi}_n')\rangle$. The size of the step λ is an important factor for the convergence properties of the algorithm [29, 33].

As the RMM-DIIS does not include the explicit orthogonalization of the CG approach or the implicit orthogonalization of blocked Davidson, the band residual minimization is very fast. This also allows easy parallelization of this scheme for bands, as applying the RMM-DIIS scheme to one band does not require information from the other bands. However, because the final vectors are not forced to be orthogonal, applying only a few steps of RMM-DIIS can result in linear dependence. This requires

an additional orthogonalization step at the end of the procedure that will involve communications between *all* computational units.

The scheme has another disadvantage: approximate eigenvectors converge to the nearest eigenvector, so the initialization step is critical. One solution is to append a small number of Davidson steps to the random initialization. Davidson is very robust and strictly orthogonal, so it will separate the estimates of the eigenvectors.

Additionally, each self-consistent diagonalization step is performed in the following order:

1. direct diagonalization in the space of initial approximate eigenfunctions,
2. RMM-DIIS minimization, and
3. explicit orthonormalization of eigenfunctions.

The subspace rotation is employed to separate the approximate eigenfunctions to lower the risk of two wavefunctions converging to the same eigenvector. Orthonormalization is added to ensure that the wavefunctions used to build the density do not overlap, which would invalidate the expression for the charge density given in Eq. (7.12). Even so, the RMM-DIIS approach is generally the most efficient scheme for large eigenproblems.

7.3 Implementation

Each iteration of the self-consistent solution of the eigenproblem Eq. (7.8) has four main steps. Beginning with an estimate of the density, ρ

1. Apply functionals to ρ to produce V^{loc}.
2. Compute eigenvalues, ϵ_{nk}, and eigenfunctions, u_{nk}, of H.
3. Use ϵ_{nk} to produce occupancies f_{nk}.
4. Use u_{nk} and f_{nk} to produce ρ.
   And after the self-consistent solution converges
5. Compute forces and stresses.

Note that in the case of insulators the occupancies are known *a priori*, so step 3 can be omitted. The first step is not generally performance-critical. The computation of forces and stresses are optional, dependent on the application (e.g., molecular dynamics or relaxation).

7.3.1 Transformations

Before describing the details of these steps, it will be useful to discuss common components: the Fourier transformation and the projection.

7.3.1.1 Fourier Transforms

The Fourier transform used in PWDFT is slightly different from the conventional unitary discrete Fourier transform. The first difference is that the Fourier basis is defined as only the g-vectors that lie within an inscribed sphere, offset by the k-point:

$$\{|g\rangle : |g + k|^2 < E_{\text{cut}}\}. \tag{7.37}$$

This follows from the physical argument that the contribution of a mode to the wavefunctions should be related to its kinetic energy, with higher energy modes being less represented. Note as well that the transformation does not include the k offset implicit in the g-vectors. Thus before taking real-space inner products, one must be sure that they are either at the same k-point or that a factor of $\exp\ [i(k_1 - k_2) \cdot r]$ is added explicitly.

The second difference is that r-space must be sampled at a higher resolution, at least by a factor of 2, to faithfully represent the potential. These two differences lead to an overall inverse transformation that has three steps: pad, transform, interpolate. Note that the r-space representation is thus about $(2)^3 6/\pi \approx 15$ times larger than g-space.

To take advantage of the reduced number of g-vectors, many PWDFT codes have implemented their own parallel 3D FFTs on top of 1D and 2D standard library calls. That way, the unused corners of g-space do not need to be communicated during the transpose steps. If targeting a single-socket system, it is recommended that full 3D FFTs are used instead, and many codes will fall back to it automatically. For parallel FFTs across many GPU nodes, the behavior of the CPU implementation can be mimicked, but scalability will suffer because of the relatively strong compute performance of the GPU-enabled nodes. To mitigate this, one should set up the run to process each band on the smallest possible number of nodes, which may require reducing the number of bands that are concurrently processed. Putting different bands from the same k-point in different memory spaces can also cause problems because of the need for inner products between test functions, which will be discussed shortly.

7.3.1.2 Projection

Projection is the process of computing the inner products $\langle \beta(k) | \phi_{nk} \rangle$, where β are the pseudopotential projectors, and k is a k-point. The projectors connect to a space of pseudo-wavefunctions, $\tilde{\theta}$, with the completeness relation:

$$I_i = \sum_l |\tilde{\theta}_{i,l}\rangle\langle\beta_{i,l}|, \tag{7.38}$$

where I_i denotes identity only in the pseudo-wave space of ion i, and l indexes angular momentum. Projection can be done in either r-space or g-space. In g-space, the projection is computed as:

$$\langle \beta(k) | \phi_{nk} \rangle = \sum_g \langle \beta | g \rangle \langle g | \phi_{nk} \rangle. \tag{7.39}$$

The matrix representation $\langle g | \beta(k) \rangle$ is dense, so projection is a matrix–vector product. Technically, this corresponds to a LAPACK call of the `ZGEMV('C', ...)` function, that has been ported to the GPU [34].

In r-space, each projector is nonzero only within a sphere centered at an ionic center. Let those r-space basis vectors for which the projector is non-zero be $\{r'_i\}$. Including the ion and angular momentum indices

$$\langle \beta(k)_{i,l} | \phi_{nk} \rangle = \sum_{r,r'_i} \langle \beta_{i,l}(0) | r'_i \rangle \langle r'_i | r \rangle e^{i(k \cdot r)} \langle r | \phi_{nk} \rangle, \tag{7.40}$$

where the sums over r_i are independent of total system size, and exp $[i(k \cdot r)]$ is included because the real-space projectors are defined at $k = 0$. The transformation described by $\langle r'_i | r \rangle$ is best described as a gather operation, as it collects r-space data that is distributed noncontiguously in memory. The angular momentum components on each atom share r-space subsets, so the number of gathers is the number of ions, not the total number of projectors. An example kernel implementing the projection is found in Listing 1.

The inverse transformation in r-space has a similar form:

$$\langle r | \tilde{\phi}_{nk} \rangle = \sum_{i,l,r'_i} e^{-i(k \cdot r)} \langle r | r'_i \rangle \langle r'_i | \beta_{i,l} \rangle \langle \tilde{\theta}_{i,l} | \phi_{nk} \rangle. \tag{7.41}$$

Here, the transformation $\langle r | r'_i \rangle$ is a scatter operation, with similar issues to the gather. An example kernel implementing the inverse projection can be found in Listing 2.

Both methods of projection are equally valid methodologically, so the choice is purely based on performance. The advantage of the more complex r-space projection is the independence on total system size. The disadvantages are that (1) r-space is about 15 times larger to begin with, and (2) gather operations can be inefficient on memory designed for sequential access.

7.3.2 Functionals

For nontrivially sized systems, the construction of the potentials V^{loc} and V^{nl} are not performance-critical. They consist of $O(n)$ operations applied to the charge density and two Fourier transforms used for a Poisson solve. Section 7.6 provides an indirect treatment under the constraint $a = b$, which specifies the two particle potential in exact exchange to a single particle Coulomb potential. It omits the exchange–correlation functional.

7.3.3 Diagonalization

The Hamiltonian is block-diagonal with respect to the index k, as seen in Eq. (7.6). Each k-point is an independent eigenproblem, and can be parallelized over freely. In the rest of this subsection, we will omit the k label.

Only the $n \approx n_e$ eigenfunctions corresponding to the lowest eigenvalues are generally required, as only they will correspond to nonzero occupancies. Implementing fast iterative matrix diagonalization is one of the key points to attaining a high performance and robust computation. Common choices were described in Section 7.2.5.

All solvers require the implementation of the action $H|\phi\rangle$ and an inner product $\langle\phi'|\phi\rangle$, for arbitrary test functions $|\phi\rangle, |\phi'\rangle$. In some cases, it may be easier to compute $\langle\phi'|H|\phi\rangle$ in a single step, rather than $\langle\phi'|(H|\phi\rangle)$, so we will treat this as a special case.

In PWDFT codes, the test functions are generally stored in the g-space, $\langle g|\phi\rangle$. Evaluating the action of the Hamiltonian further requires the r-space, $\langle r|\phi\rangle$, and β-space representations, $\langle\beta|\phi\rangle$. If there is sufficient memory, these quantities can be computed once per test function and stored for reuse. This is frequently done for $\langle\beta|\phi\rangle$, because it is much smaller than $\langle g|\phi\rangle$. The r-space representation can be a factor of $48/\pi$ larger than the g-space representation, so it is less frequently stored. The r-space and g-space representations are complete, so inner products can be computed in either r-space or g-space. Because of the larger size of r-space, g-space is always preferable. Inner products should be blocked together into matrix–matrix products whenever possible.

7.3.3.1 Kinetic Energy

The action of the kinetic energy $\hat{T}$ consists in a simple multiplication of the test function by the vector $|g + k|^2$. If the g-vectors are in order, this can be easily done by computing $|g + k|^2$ in the kernel based on the indices. If the g-vectors are out of order, then one should first precompute a vector $|g + k|^2$, and then perform an element-wise vector product.

As element-wise products are bandwidth-bound, it should be used carefully to reduce loads. For example, when computing the kinetic energy of a set of test functions $|\phi_i\rangle$, then a loop over products will reload the prefactors and thus reduce the performance. A better approach is to cast this operation into a matrix–matrix multiplication, where the first matrix is diagonal. Although BLAS does not include such an operation, cuBLAS does as ZDGMM [34]. Furthermore, the prefactor here is real while the test function is complex. Thus, a custom kernel can load reals for half the bandwidth. Finally, the operation can be performed in-place. Such a kernel is found in Listing 3. This functionality is not currently available in standard libraries.

7.3.3.2 Local Potential

To compute the action of the local potential, $\langle r|V^{\text{loc}}|\phi\rangle$, the same operation is performed in r-space, with $V(r)$ replacing $|g+k|^2$ as the prefactor. Again, a single test function corresponds to an element-wise product while a set of test functions is a ZDGMM call or custom kernel. As in the kinetic energy case, the prefactors are real. If the desired result is in g-space, then the r-space action should be Fourier-transformed.

7.3.3.3 Nonlocal Potential

The nonlocal potential in the projector space is either diagonal, in the case of NCPP, or block-diagonal, in the case of USPP. For NCPP, the action of the nonlocal potential proceeds as for the kinetic and local terms, but with a sum over the projections. For USPP, the blocks correspond to atoms, so the action looks like

$$\langle\tilde{\theta}_{i,l}|V^{\text{nl}}|\phi\rangle = \sum_{l'}\langle\tilde{\theta}_{i,l}|V^{\text{nl}}|\tilde{\theta}_{i,l'}\rangle\langle\beta_{i,l'}|\phi\rangle. \tag{7.42}$$

The l, l' angular momentum indices take a small set of values, about 10, so these matrix products are very small and simply looping over matrix–matrix calls is likely to be kernel-launch-bound. If batched calls, discussed in greater depth in Chapter 2, are available, they should be used. Otherwise, a custom kernel can achieve moderate performance, such as the one given in Listing 4.

As with the kinetic and local potential, the nonlocal potential is real. If the desired result is in g-space but the r-space projection was used, the result can be added to $\langle r|V|\phi\rangle$ prior to the Fourier transformation back to g-space, removing the need for an additional Fourier transform.

In the case of USPPs, the action of the overlap matrix, $S|\phi\rangle$, is needed. This is computed as the nonlocal potential, but for an overlap operator $Q_{i,j}$, and then added to the identity:

$$S|\phi\rangle = |\phi\rangle + \sum_{i,l,l'}|\beta_{i,l}\rangle Q_{i,l,l'}\langle\beta_{i,l'}|\phi\rangle, \tag{7.43}$$

where Q has the same structure as V^{nl}. If both $H|\phi\rangle$ and $S|\phi\rangle$ must be computed, the projections $\langle\beta|\phi\rangle$ can be reused to great effect.

Expansion-set techniques. In all the iterative methods, there will be direct solutions of small eigenproblems expressed in the expansion set. They can be solved on the GPU using libraries, such as MAGMA [35]. When their size is small, they can be low performance. Fortunately, in that case, they contribute little to the overall run time of the method.

For RMM-DIIS, the final orthonormalization can be performed efficiently using a Cholesky decomposition. First, the overlap matrix $S(i,j) = \langle\psi_i|\psi_j\rangle$ is computed. Next, the overlap matrix is decomposed as $S = LU$, which can be performed using LAPACK or MAGMA. Finally, the orthonormalized orbitals are given by $|\bar{\psi}\rangle = U^{-1}|\psi\rangle$, where the inverse can also be evaluated using LAPACK or MAGMA.

The eigenproblems in subspace methods are built with the matrix elements $\langle\phi'|H|\phi\rangle$ and, in the case of USPP and PAW, $\langle\phi'|S|\phi\rangle$. In this case, the inner product can be split into different bases:

$$\begin{aligned}\langle\phi'|H|\phi\rangle = &\sum_{g}\langle\phi'|g\rangle\langle g|T|g\rangle\langle g|\phi\rangle \\ &+ \sum_{r}\langle\phi'|r\rangle\langle r|V^{\text{loc}}|r\rangle\langle r|\phi\rangle \\ &+ \sum_{i,j}\langle\phi'|\beta_i\rangle\langle\tilde{\theta}_i|V^{\text{nl}}|\tilde{\theta}_{i'}\rangle\langle\beta_{i'}|\phi\rangle,\end{aligned} \tag{7.44}$$

where $\langle\tilde{\theta}|$ are the duals of $|\beta\rangle$. If the projections $\langle\beta|\phi\rangle$ and $\langle\beta|\phi'\rangle$ are stored, this method can reuse them to avoid a transformation back to g-space or a scatter–gather. If the r-space representations $\langle r|\phi\rangle$ are stored, this method can reuse them to avoid an FFT. The benefit is mitigated, however, by the large size of r-space, generally resulting in a slower method for small and medium sized problems.

7.3.4 Occupancies

The occupancies, or weights, of the wavefunctions are computed as a function of the entire energy spectrum over all the k-points. This computation is not expensive and should be performed on the CPU using existing code. It does, however, effectively introduce a synchronization step between diagonalization and building the electron density. If necessary, this synchronization can be partially avoided by making the safe assumption that the lowest energy bands are fully occupied. That is, for some percentage of the bands, α, assume that $f_{n,k} = 1 \forall n < \alpha n_e$. This assumption should be made conservatively and exposed to the user, as there are exceptions.

Note that this step is unnecessary for insulators: the occupancies of the n_e wavefunctions with lowest energy at each k-point are unity and the rest are zero.

7.3.5 Electron Density

After the occupancies are formed, the plane-wave contribution to the electron density, the first term of Eq. (7.12), is computed by Fourier-transforming the wavefunctions back to r-space and squaring them element-wise, weighted by their occupancy:

$$\langle r|\rho|r\rangle = \sum_{nk} f_{nk} \left| \sum_{g} \langle r|g\rangle \langle g|u_{nk}\rangle \right|^2 . \tag{7.45}$$

Low-occupancy wavefunctions can be ignored entirely. On the CPU side, this is done by cycling the loop over wavefunctions for small occupancies. On GPUs, it can be beneficial to transform multiple wavefunctions concurrently, so wavefunctions with non-negligible weight should be contiguous in memory. Listing 5 shows a schematic kernel for this task. The behavior of the occupancies is governed by the *smearing*. For Gaussian and Fermi–Dirac smearing, $f_{n+1,k} < f_{n,k}$, so the memory is already contiguous. For Methfessel–Paxton [36] and Marzari–Vanderbilt [37] smearing, this is not always the case and rearrangement may be necessary.

For ultrasoft and PAW calculations, the second term of Eq. (7.12), the ultrasoft or *augmentation* charge must be added. Because the projection $\langle\beta_i|u_{nk}\rangle$ is much smaller than the r-space wavefunctions $\langle r|u_{nk}\rangle$, this term is inexpensive relative to the plane-wave contribution. Therefore, we will not discuss it in depth here. The augmentation charge in its two-particle form is discussed in Section 7.6.1. The one-particle form needed here is recovered by equating the two band indicies, $a = b$, in Eq. (7.54).

7.3.6 Forces

The expressions for forces given in Eq. (7.21) are easily computed in the Fourier space. The electrostatic nuclear interaction is a low-cost operation and will not be discussed here. In g-space, the derivative of the local potential is diagonal:

$$\int dr \frac{\partial V_{\text{ion}}^{\text{loc}}(r)}{\partial R_a^{\mu}} \rho(r) = \sum_{g} igV_{\text{ion}}^{\text{loc}}(g)\rho(g), \tag{7.46}$$

where $V_{\text{ion}}^{\text{loc}}(g)$ and $\rho(g)$ are the Fourier transforms of the vector representations $V_{\text{ion}}^{\text{loc}}(r)$ and $\rho(r)$, respectively. Computationally, this is very similar to the local energy evaluation described in Section 7.3.3, but only only a single vector $\rho(g)$ and with a complex potential $-gV(g)$.

The derivatives of the $Q_{i,j}$ are computed similarly using its Fourier transform $Q_{i,j}(g)$.
The last terms are the projections $\langle u_{nk}|\beta_i\rangle$ and their derivatives:

$$\left\langle u_{nk} \left| \frac{\partial \beta_i}{\partial R_a^\mu} \right\rangle = \sum_g ig\langle \beta_i|g\rangle\langle g|u_{nk}\rangle. \right. \tag{7.47}$$

Evaluating the derivative terms can be done using the same strategies as those used to compute the projection in Section 7.3.1. Real-space projection will require the Fourier transformation of $g\langle g|\beta_i\rangle$ back to real space, but this needs to be computed only once for the entire calculation.

7.4 Optimizations

Two key issues in developing high-performance implementations for heterogeneous architectures are

1. maximizing the occupancy of the GPU, and
2. minimizing the high cost of transferring data between the GPU and CPU node(s).

The first issue boils down to expressing enough parallelism to feed the tens of thousands of GPU threads. The second issue is addressed by keeping data resident on the GPU for as long as possible. This can mean porting many intermediate, non-performance-critical methods to the GPU and carefully choosing and designing low-communication algorithms.

7.4.1 GPU Optimization Techniques

As explained in Chapters 1 and 2, the GPU is a massively parallel architecture capable of running several tens of thousands of threads concurrently. In order to use the GPU to its full potential, we need to express computations as large enough tasks so that every thread has work to do.

Nearly every step in PWDFT involves a loop over bands. The easiest way to generate large, GPU-ready tasks is to group bands together and express computations on those larger groups. We call those groups blocks and this technique *blocking*. The example kernels given in Section 7.3 all block over a single band index, seen by loops of size `nband`. Blocking can be especially effective when the constituent calculations share data. For example, the action of the kinetic energy is an element-wise multiplication by the same $|g + k|^2$ vector for every band. Sharing this data allows the application to read fewer memory and hence have higher arithmetic intensity and higher performance.

Sometimes, blocking over a single index, such as bands, is not enough. This could be because the number of available bands is small or the amount of work per band is small. The projection is a good example of the latter: the number of arithmetic operations per ion per band is 100 and does not scale with the problem size. In order to fully occupy the GPU, multiple ions must be processed concurrently as well.

Each ion requires a different scatter/gather of the band data, so parallelizing is not as simple as adding a band index. The next solution would be to use *streaming* to concurrently execute multiple kernels, with each kernel working on a single ion. The newest Nvidia GPUs (Kepler at the time of writing) support up to 32 concurrent kernels executing on a device, so this can be an effective source of added parallelism. Kernels issued in the same stream have implicit first-in first-out (FIFO) dependency ordering, but the ordering between kernels in different streams needs to be enforced through the use of explicit synchronization. In this case, the data dependence is intra-ion, so each ion is assigned to a stream. If starting from unstreamed code, adding streams often requires the expansion of temporary buffers to handle more data in-flight.

GPU kernel launches are not free, and if launching the kernel to the GPU takes more time than the duration of the kernel, streams will offer no acceleration. More generally, the kernel launch latency cuts into stream performance if each streamed kernel is not long-lived. The amount of work per ion

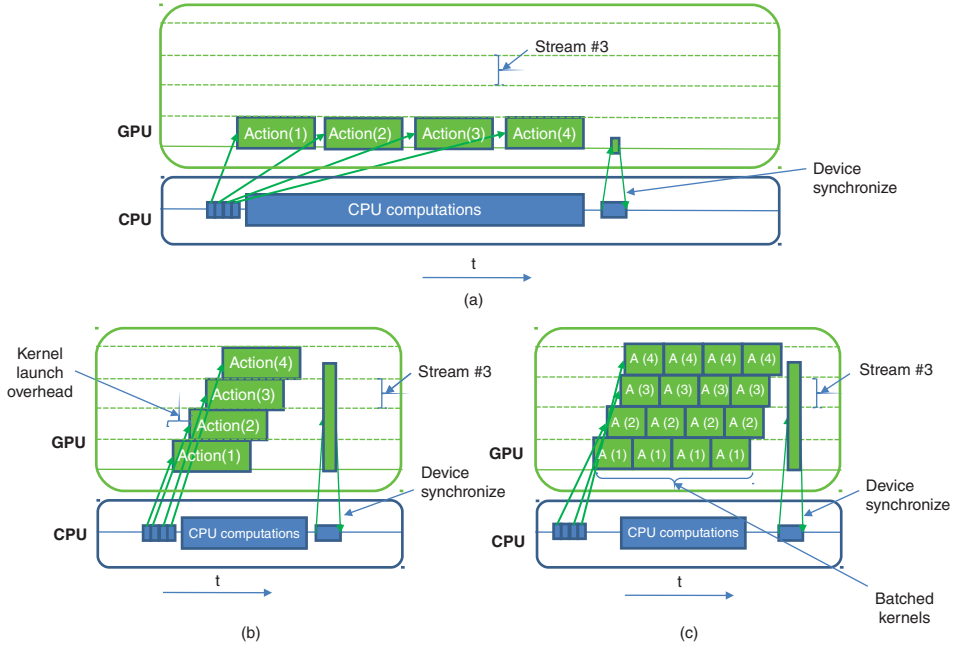


Figure 7.1 *Schematic representation of three approaches for porting loops on GPUs: (a) asynchronous kernel launches; (b) streaming kernel launches; and (c) batched kernel launches*

per band is fixed, so if the number of bands is not increasing, then the projection kernels duration will frequently be near or even below the launch latency.

In order to hide the launch latency for such small kernels, we can use *batching*: a process where we launch a single kernel that internally differentiates into multiple tasks. In the case of projection, the loop over ions would be pushed into the kernel. In a sense, batching is a more complex version of blocking where the tasks do not generally share data. It should be mentioned that batching custom kernels can lead to messy and inefficient code. Fortunately, the cuFFT and cuBLAS libraries include batched versions of the most popular kernels, which help sensure the performance of the kernels. These library routines generally impose homogeneous problem sizes across elements of the batch, so zero-padding is sometimes necessary if the batches span ionic types.

Figure 7.1 depicts both streaming and batching, which are more thoroughly described in Section 2.4. In addition to the projection, streams and batches can be used to improve the occupancy of the nonlocal action and some parts of RMM-DIIS diagonalization.

7.4.1.1 Reduce Communication Cost to/from the GPU

While GPUs feature fast on-chip memory (the latest K40 GPUs have 288 GB/s of bandwidth to the global memory), any data that has to travel between the CPU and GPU currently has to go through the PCIe bus, which is relatively low bandwidth (PCIe gen 3, e.g., has a theoretical maximum of 16 GB/s per direction). Hence, in order to achieve the full performance advantage of the GPU, we need to minimize the cost of the transfer of data between the GPU and the CPU. There are a couple of strategies to reduce the time spent waiting for data transfers.

- Reduce the need to copy data back and forth between the GPU and CPU by porting additional functions to the GPU, even if the computation of the functions themselves is not a bottleneck.
- Overlap data transfers with computations using streams and asynchronous memory copies.

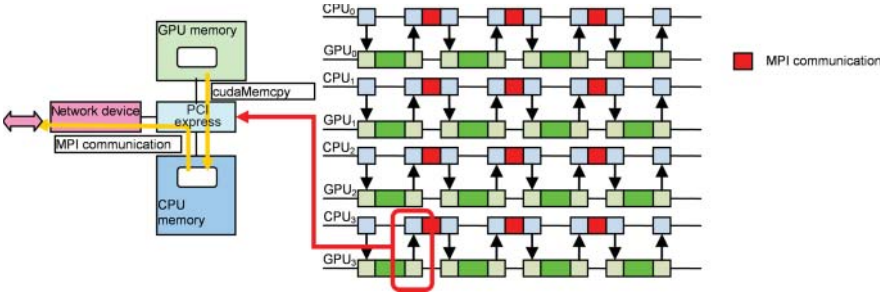


Figure 7.2 *Using multiple process accelerated by GPUs communicating with MPI. (See insert for colour representation of this figure)*

The direct diagonalization of the RMM eigenproblem is an example of a computationally inexpensive operation that should be ported to prolong GPU data residency. The expansion set for each band generally has only a few vectors, resulting in very small eigenproblems that have no impact on overall performance. There is one such eigenproblem per band per RMM-DIIS iteration, so copying data from and to the GPU memory space each time would be prohibitively expensive. To prevent these frequent and costly data movements, MAGMA routines should be used, even though they are highly inefficient for such small problem sizes.

7.4.2 Parallel Optimization Techniques (Off-Node)

7.4.2.1 Multiple Host CPUs and Multiple GPUs

In order to address larger physical systems, one needs to deal with platforms composed of multiple GPUs and multicore CPUs. There are two general parallel setups:

1. Each GPU receives kernels from exactly one CPU process
2. Each GPU receives kernels from multiple CPU processes

The first setup can be further divided into the case where only one CPU process runs per GPU and the case where many CPU processes run per GPU, but only one is designated for kernel launch. As we will see, recent improvements to the GPU runtime have all but deprecated the latter setup.

There are two advantages to a one-process-per-GPU setup. First, this setup minimizes the number of messages that must be sent between processes, which reduces the overall share of the communication time. These messages can be seen in Figure 7.2. Because GPU-enabled nodes are generally much more arithmetically capable than CPU-only counterparts, GPU implementations are more sensitive to the communication overhead. Using streams to overlap communication and computation can help mitigate this effect. Second, this setup maximizes the potential for blocking and batching by dividing the data as little as possible. GPU can run only a fixed number of kernels concurrently, presently 32. If the problem is decomposed beyond the point at which 32 kernels occupy the device, then further parallelism will reduce occupancy and degrade performance.

There are two advantages to a multiple processes per the GPU setup. The first is that using multiple processes will accelerate the CPU portion of the work. For PWDFT, we have described how to port the majority of the workload, so this is less of an issue. The second advantage is implicit streaming with the multiprocess server (MPS). MPS allows kernels from different CPU processes to run concurrently. By default, each CPU process submits kernels to a different stream, so using multiple processes is like streaming over the entire calculation, excepting barriers. This can be a quick and easy way to improve GPU occupancy without writing streams or batches into the code.

7.4.3 Numerical Optimization Techniques

7.4.3.1 Mixed Precision

PWDFT has traditionally been performed entirely in double precision, as are most scientific numerical methods. However, the sensitivity to numerical precision is present only in certain parts of the method. For example, the accumulation of the electron density, Eq. (7.45), is a sum of small numbers into a much larger one, and is therefore susceptible to numerical roundoff errors. On the other hand, the diagonal parts of the action of the Hamiltonian are sum-less, and thus require much fewer digits of precision. Furthermore, the subspace diagonalization steps at the beginning of the self-consistent iteration are expected to produce only approximate solutions in a small number of iterations, diminishing the importance of numerical stability.

We can group the steps into three categories: insensitive, sensitive, and situational.

Insensitive	**Situational**	**Sensitive**
Kinetic energy	Fourier transformation	Density accumulation
Local potential	Projection	(Orthonormalization)
	Subspace diagonalization (early)	Subspace diagonalization (late)

It is the authors' opinion that the insensitive steps can be freely cast to single precision, the situational steps should provide a switch to the user, and the sensitive steps should not be altered. The VASP implementation of exact exchange performs Fourier transformations in single precision by default and has yet to experience an accuracy issue [38]. Fourier transforms were selected as a testbed due to their combination of high overall cost and relative code isolation.

7.5 Performance Examples

The steps above have been ported to the GPU in the Quantum Espresso [39] and VASP codes with a special focus on the block Davidson algorithm [40] or the RMM-DIIS one [41]. In this work, we have used a new GPU acceleration of VASP based on the CPU version 5.3.5 [42]. Our RMM-DIIS work has been further optimized, and all steps of the block Davidson algorithm have been ported to GPU. Moreover, this version also includes the GPU acceleration of the exact exchange part of VASP from one of us [38].

7.5.1 Benchmark Settings

7.5.1.1 Timed Sections

The runtime of the calculation is divided into categories based on the steps discussed in Section 7.3:

- **Fourier**: transform between r-space and g-space.
- **Projection**: computation of $\langle\beta|\psi\rangle$.
- $\hat{\boldsymbol{T}}$ **and V**: action of the kinetic energy and local potential.
- **Nonlocal**: action of the non-local potential.
- **Sub diag**: sub-space diagonalization, including orthonormalization.
- $\boldsymbol{\rho}$**-sum**: sum of electron density, but not transformation.

7.5.1.2 Node Configuration

The examples are demonstrated on the Nvidia PSG cluster. Each node contains two 10-core Intel Ivy-Bridge CPUs, 128 GB of RAM, and four Nvidia Tesla K40m GPUs. The CPUs are clocked at 3 GHz

and have 60 GB/s peak memory bandwidth.[1] The GPUs have 1.43 TFlop/s peak double-precision performance and 288 GB/s peak memory bandwidth (see also Chapter 1).

However, to better approximate common production configurations at the time of writing, we under-subscribe the node slightly: we only use eight CPU cores and one GPU per socket. When using both sockets, the two GPUs are selected from different PCIe busses to avoid contention.

Consider the maximum theoretically achievable speedup, or just *max speedup*, based on the raw hardware and assuming an optimal CPU implementation. GPU ports running near the max speedup should be considered successful and are unlikely to be further accelerated. When the max speedup is exceeded, it indicates a suboptimal CPU implementation. For compute-bound operations, the max speedup is 7.4× faster for a GPU than eight CPU cores on our test system. For memory-bound operations, the max speedup is 4.8×. Switching to single precision on the GPU (but keeping double precision on CPU) improves the compute- and memory-bound max speedup to 22.1× and 9.6×, respectively.

The cluster is equipped with Intel compilers at version 14, the CUDA toolkit at version 6.0, and runs the Nvidia multi-processes server (MPS). Version 1.4.1 of the MAGMA library was used [35].

7.5.1.3 Runtime Configuration

We run the code on six hardware configurations:

- Single core, no GPU.
- Single socket, no GPU.
- Dual socket, no GPU.
- Single core, single GPU.
- Single socket, single GPU.
- Dual socket, dual GPU.

The configurations with more CPU cores than GPUs are constrained to use eight or fewer CPU processes per GPU, specified in the tables for each calculation. It is worth emphasizing that GPU performance is generally optimized using four CPU processes.

7.5.1.4 Types of Runs

We illustrate the performance of GPU implementations of PWDFT in several contexts: the band structure of bulk silicon (see Figure 7.3); *ab initio* molecular dynamics (AIMD) of bulk gold (see Figure 7.4); and structural relaxation of elemental boron (see Figure 7.5). The self-consistent

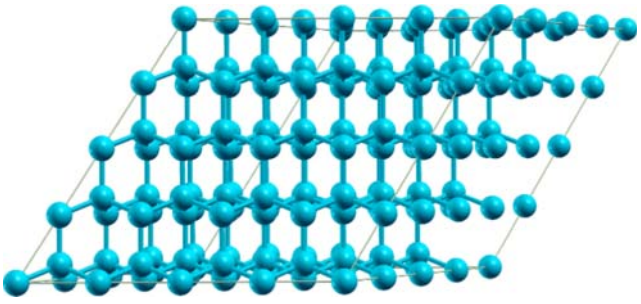


Figure 7.3 $4 \times 4 \times 4$ *supercell of crystalline Si*

[1] http://ark.intel.com/products/75279/Intel-Xeon-Processor-E5-2690-v2-25M-Cache-3_00-GHz.

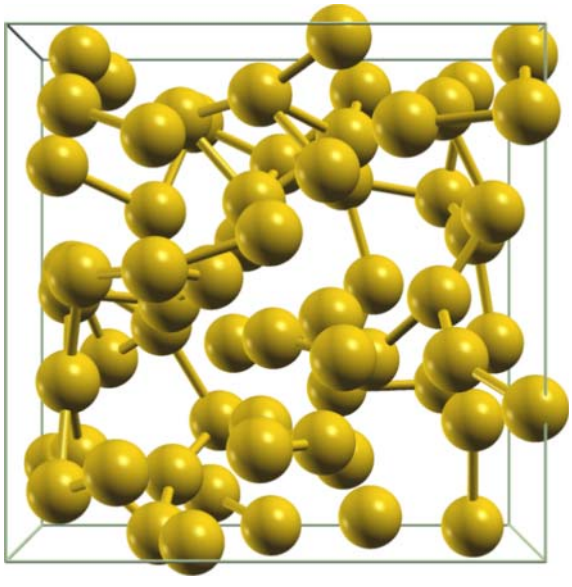


Figure 7.4 *Gold MD snapshot at 1600 K*

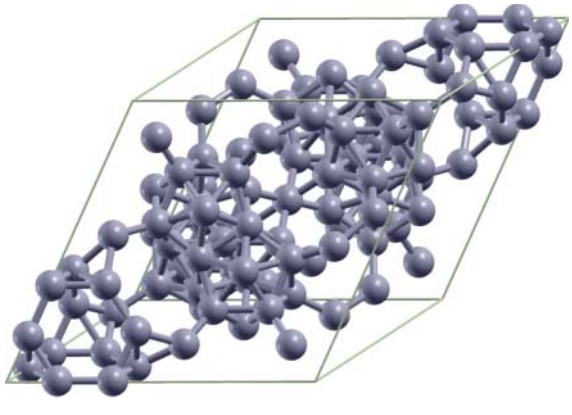


Figure 7.5 *Boron in hR105 (β-rhombohedral) structure*

calculation of the charge density is the first step of all PWDFT calculations. Band structure calculations take the self-consistent charge density as an input and then compute the energy eigenvalues on a larger set of *k*-points, generally arranged in high symmetry paths. AIMD consists of a series of self-consistent calculations interspersed with the classical motion of the ions. Structural relaxation consists of a series of self-consistent calculations interspersed with small perturbations of the ionic structure.

In all cases, the Brillouin zone integration was conducted with a *k*-point mesh generated with the Monkhorst–Pack algorithm [43]. The atomic cores were described with the projector augmented wave method [13] as implemented in VASP [12].

7.5.2 Self-Consistent Charge Density

The self-consistent computation is performed with a $4 \times 4 \times 4$ k-point mesh. The generalized gradient approximation was used in the formulation of Perdew and Wang PW91 [44, 45], with a plane-wave cutoff energy at 245.4 eV. In order to compare the GPU acceleration for different system sizes, calculations were conducted on three systems containing 128, 256, and 512 atoms. The 128 and 512 atom cases use the block Davidson diagonalizer. The 256 atom case is conducted with both the block Davidson and RMM-DIIS diagonalizers for comparison. For the 512 atoms system, only three electronic steps were done to reduce the computation time. The block Davidson results are given in Tables 7.1–7.3 for 128, 256, and 512 atoms, respectively. The RMM-DIIS results for 256 atoms are given in Table 7.4.

7.5.2.1 *Profile*

The costs using block Davidson are dominated by subspace diagonalization and Fourier transforms, which take 45 and 30% of the single-core CPU time, respectively. The next most expensive step is projection, at around 15%. The remaining costs can be considered insignificant.

Increasing the system size shifts effort further towards subspace diagonalization. This makes sense, as diagonalization has the highest order complexity.

Table 7.1 *Self-consistent electron density calculation of bulk Si with a supercell of 128 atoms using block Davidson diagonalization ($N_{ion} = 128$, $N_{band} = 320$, $N_{pw} = 216,000$)*

CPU cores/ GPUs	1/0	1/1	x	8/0	8/1	x	16/0	16/2	x
Fourier	1122.0	117.7	9.5	150.5	108.1	1.4	101.0	54.4	1.9
Projection	457.9	26.7	17.1	95.1	27.4	3.5	78.0	19.2	4.1
K and V	140.3	12.8	11.0	27.6	8.6	3.2	27.2	4.8	5.7
Nonlocal	6.6	7.6	0.9	1.3	2.0	0.7	1.0	2.0	0.5
Sub diag	1494.3	87.4	17.1	215.0	77.9	2.8	117.5	59.1	2.0
ρ-sum	19.9	1.4	14.2	2.6	0.8	3.3	1.7	0.5	3.4
Other	68.8	36.0	1.9	57.7	61.6	0.9	30.8	57.7	0.5
Total	3309.7	289.6	11.4	549.7	286.4	1.9	357.2	197.7	1.8

Times in seconds, GPU speedup denoted by *x*.

Table 7.2 *Self-consistent electron density calculation of bulk Si with a supercell of 256 atoms using block Davidson diagonalization ($N_{ion} = 256$, $N_{band} = 644$, $N_{pw} = 432,000$)*

CPU cores/ GPUs	1/0	1/1	x	8/0	8/1	x	16/0	16/2	x
Fourier	10,110.6	916.8	11.0	1,528.3	845.6	1.8	1,036.2	410.6	2.5
Projection	4,332.3	211.1	20.5	861.0	224.2	3.8	636.6	126.7	5.0
K and V	1,086.3	82.6	13.2	246.1	68.7	3.6	212.8	34.4	6.2
Non-local	52.1	57.2	0.9	9.7	15.0	0.6	7.2	14.9	0.5
Sub diag	23,469.1	889.8	26.4	3,351.2	769.5	4.4	1,652.0	587.3	2.8
ρ-sum	160.6	10.0	16.1	23.4	6.4	3.7	18.2	3.9	4.7
Other	399.5	166.0	2.4	450.8	384.4	1.2	204.9	354.9	0.6
Total	39,610.6	2,333.5	17.0	6,475.0	2,313.8	2.8	3,769.6	1,532.7	2.5

Times in seconds, GPU speedup denoted by *x*.

Table 7.3 *Self-consistent electron density calculation of bulk Si with a supercell of 512 atoms using block Davidson diagonalization (N_{ion} = 512, N_{band} = 1284, N_{pw} = 864,000)*

CPU cores/ GPUs	1/0	1/1	x	8/0	8/1	x	16/0	16/2	x
Fourier	7,307.6	589.0	12.4	1,199.7	527.3	2.3	770.9	246.4	3.1
Projection	4,437.5	142.8	31.1	604.9	151.1	4.0	428.8	90.6	4.7
K and V	853.1	43.6	19.6	144.3	41.3	3.5	124.7	19.1	6.5
Nonlocal	34.7	34.8	1.0	5.6	9.0	0.6	4.3	8.8	0.5
Sub diag	34,050.8	542.7	62.7	3,958.2	432.4	9.2	1,779.5	260.8	6.8
ρ-sum	46.3	2.5	18.5	7.0	1.8	3.9	5.0	1.0	5.0
Other	418.2	275.5	1.5	369.8	379.3	1.0	237.2	351.1	0.7
Total	47,148.1	1,630.9	28.9	6,289.6	1,542.2	4.1	3,350.5	977.8	3.4

Times in seconds, GPU speedup denoted by *x*.

Table 7.4 *Self-consistent electron density calculation of bulk Si with a supercell of 256 atoms using RMM-DIIS diagonalization (N_{ion} = 256, N_{band} = 644, N_{pw} = 432,000)*

CPU cores/ GPUs	1/0	1/1	x	8/0	8/1	x	16/0	16/2	x
Fourier	10,760.5	1,001.3	10.7	1,692.6	611.1	2.8	1,182.0	284.7	4.2
Projection	4,487.8	201.6	22.3	722.6	215.5	3.4	552.2	107.1	5.2
K and V	2,448.8	107.7	22.7	401.9	394.6	1.0	320.2	159.6	2.0
Nonlocal	81.8	59.1	1.4	11.5	36.7	0.3	7.7	19.0	0.4
Sub diag	10,619.1	507.7	20.9	1,596.0	747.0	2.1	1,059.2	438.8	2.4
ρ-sum	146.2	9.1	16.1	21.0	19.1	1.1	16.3	6.8	2.4
Other	734.2	568.8	1.3	473.3	635.8	0.7	236.9	469.2	0.5
Total	29,278.5	2,455.3	11.9	4,918.9	2,659.8	1.8	3,374.6	1,485.2	2.3

Times in seconds, GPU speedup denoted by *x*.

Switching from block Davidson to RMM-DIIS evens the distribution of costs between Fourier transforms and subspace diagonalization at 35% each. Projection remains the next largest cost at 15%, followed by the local action at around half that.

7.5.2.2 Single-Socket

For the 128-atom system, single-socket GPU acceleration is 2×. The subspace diagonalization, projection, and local action steps are about 3× faster but the Fourier transforms only exhibit 40% improvement.

For the 256-atom system, single-socket performance with GPU improves to nearly 3×. The improvement can be attributed to two factors: the performance-relevant sections each exhibit improved performance and the overall effort in the calculation has shifted towards subspace diagonalization, which performs better on the GPU than the Fourier transforms. Larger problems will generally take greater advantage of the GPU both on a kernel by kernel basis and by spending more time in highly efficient dense linear algebra.

Single-socket RMM-DIIS GPU performance trails block Davidson with respect to both absolute time and the comparison to the CPU socket. The absolute time gap is around 5 min in favor of block Davidson, compared to 25 min in favor of RMM-DIIS on the CPU. The difference is mostly in the efficiency of the subspace diagonalization. The larger subspace problems in block Davidson do a better

job of occupying the GPU. This factor is mitigated somewhat by the shift of effort from subspace diagonalization to Fourier transforms.

7.5.2.3 Scaling

Scaling of the GPU implementation over multiple GPUs is comparable to that of the CPU implementation over multiple cores. The 128- and 256-atom systems scale from one to two sockets at 72 and 75% efficiency, respectively, compared to 76 and 85% for the CPU. The Fourier transforms actually scale nearly perfectly, improving their standing compared to the CPU significantly. However, MPI communication involved in the redistribution over bands or plane waves in block Davidson drags the subspace diagonalization and other sections down considerably.

By comparison, RMM-DIIS scales with 90% parallel efficiency on the GPU versus only 73% on the CPU. This brings the dual-socket GPU advantage up from 1.8× to 2.3×. Further, the GPU implementation of RMM-DIIS outperforms block Davidson on two GPUs in absolute terms, in contrast to the block Davidson advantage on a single GPU. The reason for this is the same as that for the single-socket behavior: RMM-DIIS creates loosely coupled small subproblems, while block Davidson creates tightly coupled larger subproblems. On a single GPU, the larger subproblems in Davidson achieve higher occupancy and therefore better performance. On multiple GPUs, the looser coupling in RMM-DIIS improves the communication to computation ratio and therefore parallel efficiency.

7.5.3 Band Structure

The non-self-consistent computation receives the electron density from the self-consistent calculation, constructs the corresponding Hamiltonian, and performs a single diagonalization, steps 1 and 2, at each k-point along a path through the Brillouin zone. Because the non-self-consistent procedure assumes an accurate starting density, the first and only diagonalization must be performed to high accuracy, greatly increasing the number of subspace steps in comparison to the partial diagonalization found in the self-consistent procedure. The runtime parameters for the band structure calculation are identical to those of the self-consistent charge density calculation, other than the number of k-points. The results are given in Table 7.5.

7.5.3.1 Profile

Overall, the non-self-consistent computation takes longer than the self-consistent counterpart but distributes the runtime in very similar proportions to the self-consistent calculation.

Table 7.5 *Non-self-consistent bulk Si band structure calculation ($N_{ion} = 128$, $N_{band} = 320$, $N_{pw} = 216,000$)*

CPU cores/ GPUs	1/0	1/1	x	8/0	8/1	x	16/0	16/2	x
Fourier	2934.0	210.1	14.0	375.6	201.1	1.9	251.5	97.4	2.6
Projection	1450.0	57.7	25.1	254.7	63.3	4.0	210.2	53.7	3.9
K and V	421.6	26.9	15.7	73.6	17.9	4.1	73.1	9.7	7.5
Nonlocal	18.3	16.0	1.1	3.3	4.3	0.8	2.5	4.4	0.6
Sub diag	3985.8	173.2	23.0	572.2	144.9	3.9	312.0	104.4	3.0
ρ-sum	8.2	0.5	16.4	1.0	0.3	3.3	0.7	0.2	3.5
Other	148.1	49.4	3.0	133.9	129.5	1.0	66.4	130.1	0.5
Total	8966.0	533.8	16.8	1414.5	561.3	2.5	916.5	399.9	2.3

Times in seconds, GPU speedup denoted by *x*.

7.5.3.2 Single-Socket

The GPU implementation performs better in the non-self-consistent calculation than before. The GPU time is divided in similar proportions but the FFT, projection, and subspace diagonalization outperform their self-consistent counterparts. The kernels are not parallelized over k-points, so that the difference is unlikely to have had an effect. The longer path to convergence likely increased the expansion set sizes, which therefore increased the size of the blocks and batches, improving occupancy.

7.5.3.3 Scaling

The scaling is very similar to the self-consistent case: FFTs scale very well, but communication in subspace diagonalization and other reduces the efficiency. The negative effects are somewhat lesser than in the self-consistent case; for the same reason, the single-socket performance is better: larger expansion set sizes improve the communication to computation ratios.

7.5.4 AIMD

Born–Oppenheimer MD is the classical movement of the ionic positions under forces derived from the quantum solution of the electron density. It consists of alternating electronic minimization and ionic motion steps, the former dominating the latter with respect to computational cost. To demonstrate the implementations on AIMD, we consider 64 atoms of bulk gold at 1600 K. One full ionic step is taken, using the RMM-DIIS diagonalizer for electronic minimization. The electronic minimization is performed using a $4 \times 4 \times 4$ k-point grid, with the PBE functional [46]. The plane-wave cutoff energy was set at 229.9 eV. The results are given in Table 7.6.

7.5.4.1 Profile

Fourier transforms, subspace diagonalization, and projection are the dominant steps, taking 35%, 35%, and 15% of the runtime, respectively. The increase in importance of Fourier transforms and the associated decrease for subspace diagonalization are due to the use of the faster RMM-DIIS diagonalizer.

7.5.4.2 Single-Socket

Single-socket performance is 2× faster on the GPU than the CPU-only system. The Fourier transforms, projection, and local action all exhibit better performance than in silicon, which is significant

Table 7.6 *Ab-initio molecular dynamics of 64 gold atoms at 1600 K, seen in Figure 7.4 ($N_{ion} = 64$, $N_{band} = 422$, $N_{pw} = 110,592$)*

CPU cores/ GPUs	1/0	1/1	x	8/0	8/1	x	16/0	16/2	x
Fourier	2686.6	164.4	16.3	363.7	164.5	2.2	215.5	99.8	2.2
Projection	1160.1	77.9	14.9	211.6	53.6	4.0	141.3	42.8	3.3
K and V	604.4	37.5	16.1	89.3	22.2	4.0	63.3	14.8	4.3
Nonlocal	86.7	54.4	1.6	12	14.6	0.8	7.1	14.7	0.5
Sub diag	2495.6	208.9	12.0	433.9	160.7	2.7	312.1	120.1	2.6
ρ-sum	51.7	4.1	12.6	6.7	1.9	3.5	3.4	1.4	2.4
Other	191.1	163.4	1.2	99.6	173.8	0.6	65.3	164.8	0.4
Total	7278.4	710.6	10.2	1216.8	591.3	2.1	808.1	458.4	1.8

Calculation takes only one step and uses the RMM-DIIS diagonalizer.

given this problem's smaller size. The subspace diagonalization, however, does not experience the same performance gains as the Davidson diagonalizer used earlier because much less compute time is spent on dense linear algebra operations, for example, matrix multiplications.

The projection, local action, nonlocal action, and ρ-sum sections all improve significantly when multiple CPU cores are used with a single GPU. These operations are composed of many small kernels, so the implicit streaming introduced by greater CPU-side parallelism improves occupancy, as discussed in Section 7.4.2. This trend is present to a lesser degree in silicon, but is emphasized here by the small size of the system.

7.5.4.3 Scaling

From one to two sockets, the GPU scales at 64% efficiency, compared to 75% for the CPU. The GPU scaling is hampered by subspace diagonalization and projection, each of which only sees 25% improvements with double resources. This is mostly because the problem size is small, which causes both the CPU and GPU to scale poorly. The overall GPU advantage remains steady around 2×. Note that the Fourier transforms scale just as well as for Si and it is the CPU that is makes up ground thanks to the significantly increased number of bands.

7.5.5 Structural Relaxation

To demonstrate the implementations on structural relaxation, we consider the boron allotrope hR105. In the next section, hybrid functionals are demonstrated on this structure. Because of the high cost of hybrid functionals, this structure is chosen to be particularly small to make the hybrid calculations tractable.

Structural relaxation is the minimization of the energy with respect to the ionic degrees of freedom. As in MD, at each step of this process, forces are evaluated and used to update the ionic coordinates. Depending on the complexity of the considered systems, many algorithms have been designed to reduce the number of forces evaluations, ranging from the steepest descent that simply follow the forces to the elaborate limited memory Broyden–Fletcher–Goldfarb–Shanno that make use of the second derivative of the energy with respect to the ionic positions [47]. Because the ionic positions converge to a minimization solution, the acceleration of the electronic minimization step due to charge density reuse is even more pronounced than in MD.

The block Davidson algorithm was used for electronic minimization with the PBE functional [46]. A 2×2×2 k-point mesh and a plane-wave cutoff energy of 318.6 eV were used. The results are given in Table 7.7.

7.5.5.1 Profile

The dominant steps are Fourier transformation, subspace diagonalization, and projection. Here, projection takes slightly longer and diagonalization slightly shorter than in the other examples presented

Table 7.7 *Structural relaxation of hR105 boron ($N_{ion} = 105$, $N_{band} = 224$, $N_{pw} = 110,592$)*

CPU cores/ GPUs	1/0	1/1	*x*	8/0	8/1	*x*	16/0	16/2	*x*
Fourier	442.2	42.6	10.4	61.7	33.8	1.8	34.0	26.4	1.3
Projection	279.9	20.8	13.5	57.9	19.5	3.0	44.9	16.7	2.7
K and V	40.0	6.0	6.7	6.9	3.1	2.2	7.4	6.8	1.1
Nonlocal	3.5	4.3	0.8	0.6	1.2	0.5	0.5	1.5	0.3
Sub diag	338.3	36.0	9.4	53.6	27.9	1.9	35.0	26.5	1.3
ρ-sum	9.4	0.7	13.4	1.2	0.4	3.0	0.6	0.7	0.9
Other	73.8	63.0	1.1	32.5	47.2	0.7	20.9	41.6	0.5
Total	1187.0	173.4	6.8	214.5	133.1	1.6	143.2	120.2	1.2

above. This is due to the rapid electronic convergence at the end of the structural relaxation when the ionic positions do not move much from ionic step to ionic step, which reduces the size of the expansion sets and therefore subspace diagonalization.

7.5.5.2 Single-Socket

The single-socket performance is similar to that of silicon, but generally worse. For projections and Fourier transforms, the degradation is due to there being half as many plane waves and therefore half as much work to parallelize over. Some performance is recovered by running with multiple CPUs per GPU, as in the case of gold. For subspace diagonalization, the rapid electronic convergence near the end of the structural relaxation uses smaller expansion sets and therefore small subspace problems, which are less efficient on the GPU.

7.5.5.3 Scaling

This system, with only 105 atoms and 110,592 plane waves, is simply not large enough to scale well. Even so, the multi-GPU performance is commensurate with two CPU sockets.

7.6 Exact Exchange with Plane Waves

Traditionally, PWDFT has been employed with explicit density functionals. Recently, extensions to hybrid functionals employing exact (Hartree–Fock-like) exchange have been made. In hybrid functionals, the exchange energy E_x is approximated as a linear combination [3, 48] of an explicit density functional, E_x^{EF} and the exact Hartree–Fock exchange term E_x^{HF}:

$$E_x = \alpha E_x^{\mathrm{HF}} + (1-\alpha)E_x^{\mathrm{EF}}, \tag{7.48}$$

where the explicit functional term can similarly be a linear combination of different explicit terms, such as a local density or generalized gradient approximation. The Hartree–Fock exchange is given by

$$K_{nk,mq} = \int dr_1 \int dr_2 \frac{\psi_{nk}^*(r_1)\psi_{mq}^*(r_2)\psi_{nk}(r_2)\psi_{mq}(r_1)}{|r_1 - r_2|}, \tag{7.49}$$

$$E_x^{\mathrm{HF}} = \frac{1}{2}\sum_{\substack{nk \\ mq}} f_{nk} f_{mq} K_{nk,mq}, \tag{7.50}$$

where f_{nk} is the occupancy of the nth band of the kth reciprocal lattice point, and ψ are the full single-particle wavefunctions.

The double sum over reciprocal lattice samples and bands adds an order in N to the action of the Hamiltonian, leading to an $\mathcal{O}(N^3 \log N)$ PWDFT method. Therefore, in calculations involving hybrid functionals, the computation of the Hartree–Fock exchange dominates the resource requirements. Furthermore, the iterative diagonalization method should emphasize minimization of the number of Hamiltonian evaluations rather than orthonormalization, which is only $\mathcal{O}(N^3)$ with a smaller prefactor. This encourages the use of the Davidson scheme rather than RMM-DIIS.

The wavefunction-dependent part of the action of the Hartree–Fock exchange operator is

$$\langle r|K[\tilde{b}]|\tilde{\psi}_a\rangle = \langle r|V_{a,b}|r\rangle\langle r|\tilde{\psi}_b\rangle + \sum_{i,j}\langle r|\beta_i\rangle D_{i,j}[V_{a,b}]\langle\beta_j|\tilde{\psi}_b\rangle, \tag{7.51}$$

$$\langle r|\tilde{K}|\tilde{\psi}_a\rangle = \sum_b \langle r|\tilde{K}[b]|\tilde{\psi}_a\rangle, \tag{7.52}$$

where we have compressed the indices $a \equiv (n,k), b \equiv (m,q)$. $V_{a,b}$ is a complex two-particle potential given by

$$\langle r|V_{a,b}|r\rangle = f_b \int \frac{\langle r'|\rho_{a,b}|r'\rangle}{|r-r'|} dr' \tag{7.53}$$

$\rho_{a,b}$ is a two-particle density given by

$$\langle r|\rho_{a,b}|r\rangle = \langle r|\tilde{\psi}_a\rangle\langle\tilde{\psi}_b|r\rangle + \sum_{i,j} Q_{i,j}(r)\langle\beta_j|\tilde{\psi}_a\rangle\langle\tilde{\psi}_b|\beta_i\rangle \tag{7.54}$$

and D is a matrix functional of $V_{a,b}$ given by

$$D_{i,j}[V_{a,b}] = \sum_r Q_{i,j}(r)\langle r|V_{a,b}|r\rangle. \tag{7.55}$$

We have written the sum over r' as an integral in Eq. (7.53) to highlight it as a Poisson solve. As in the case of USPPs, Q is the nontrivial part of the overlap matrix $S - I$, and D is a nonlocal potential $D[V_{a,b}] \equiv V^{\text{nl}}_{a,b}$.

The decomposition into a sum over $\tilde{K}[b]$ characterizes Hartree–Fock exchange as an all wavefunction, or *orbital-dependent*, energy functional. It can be useful to think of the computation as being over pairs of band indices a, b, akin to Eq. (7.50).

The Hartree–Fock exchange term in VASP [15, 49] has been ported to the GPU [38]. We use this port to demonstrate the key features of exact exchange on the GPU.

7.6.1 Implementation

The computation of exact exchange can be divided into two phases: the construction phase and the action phase of the two-particle potential. The construction phase produces the two-particle potential in local and nonlocal forms, Eqs. (7.53) and (7.55), respectively. The action phase, Eq. (7.51), is the same operation as the action of the conventional single-particle potentials from Section 7.2.1, the last two terms of Eq. (7.6). The techniques presented in Section 7.3 can be reused. For that reason, this section will focus on the construction of the two-particle potential.

7.6.1.1 Two-Sided Projection

As with conventional PWDFT, basis transformations are a large part of exact-exchange calculations. Here, however, we are also interested in the projection of 2-tensor quantities using $Q_{i,j}(r)$, which was introduced in Section 7.3.1. $Q_{i,j}(r)$ is a 2,2-tensor that connects diagonal real-space 1,1-tensors to pairs of projections. The forward transformation is

$$D_{i,j}[\phi] = \sum_r Q_{i,j}(r)\phi(r), \tag{7.56}$$

where ϕ is an arbitrary diagonal real-space 1,1-tensor. The reverse transformation is

$$\hat{\phi}_{ab}(r) = \sum_{i,j} \langle r|r'\rangle Q_{i,j}(r')\langle\beta_i|\phi_a\rangle\langle\phi_b|\beta_j\rangle, \tag{7.57}$$

where ϕ_a, ϕ_b are arbitrary vectors, and $\hat{\phi}_{ab}$ is their transformation. We will call these two-sided projections the *tensor projection* and the procedure described in Section 7.3.1 *vector projection*.

While this procedure is formally correct, the tensor Q has additional structure that can be exploited to factorize the computation. Considering that the i,j indices of the projectors run over angular

momenta, Q can be given additional indices that run over the total angular momentum LM of the sum of individual angular momenta i and j:

$$\hat{\phi}_{ab}(r) = \sum_{i,j,L,M} \langle r|r'\rangle Q_{i,j}^{L,M}(r')\langle \beta_i|\phi_a\rangle\langle\phi_b|\beta_j\rangle. \tag{7.58}$$

where $Q_{i,j}^{LM}(r)$ has an analytic form that separates the i,j and r dependencies:

$$Q_{i,j}^{LM}(r) = q_{i,j}^{LM} g_L(|r-R|)Y_{LM}(r-R), \tag{7.59}$$

where R is the position of the ionic center, $q_{i,j}$ are constants, and the Y_{LM} functions are the standard spherical harmonics.

The reverse two-sided projection can be rewritten as

$$\hat{\phi}_{ab}(r) = \sum_{r'}\langle r|r'\rangle \left[\sum_{L,M} g_L(|r'-R|)Y_{LM}(r'-R)\left(\sum_{i,j} q_{i,j}^{LM}\left\langle \beta_i|\phi_a\right\rangle\langle\phi_b|\beta_j\rangle\right)\right]. \tag{7.60}$$

In this context, the two-sided reverse projection is broken into three steps: the transformation from i,j-space to LM-space, the reverse projection from LM-space to r'-space, and the scatter from r'-space to the full r-space. The forward two-sided projection can be treated similarly:

$$D_{i,j}[\phi] = \sum_{L,M} q_{i,j}^{LM}\left[\sum_{r'} g_L(|r'-R|)Y_{LM}(r'-R)\left(\sum_{r}\left\langle r'|r\right\rangle \phi(r)\right)\right], \tag{7.61}$$

where we have simply reversed the individual steps.

The scatter/gather and projection from r' to LM space is exactly of the vector projection from Section 7.3.1. Therefore, we can reuse the vector projection kernels Listings 1 and 2. The Q_{ij}^{LM} factorization has turned nested loops over (i,j,r) into sequential loops over (i,j) and (r) and facilitated reuse of the vector projection code. For this reason, we will focus on this method of tensor projection: that is, projection to LM and transformation from LM to i,j.

7.6.1.2 Two-Particle Pseudo-density

The first term in Eq. (7.54) is an element-wise product in real space. Much like the element-wise products present in diagonal products for kinetic energy and local potential, this operation is bandwidth-bound. The number of loads can be reduced by constructing multiple potentials in blocks, which reuse the wavefunctions. This is demonstrated in Listing 6.

7.6.1.3 Two-Particle Augmentation Density

The second term in Eq. (7.54) is the two-particle reverse projection. Using the form of Eq. (7.60), one need only compute the transformation given by $q_{i,j}^{LM}$:

$$\hat{\rho}_{ab}(LM) = \sum_{i,j}\langle u_b|\beta_i\rangle q_{i,j}^{LM}\langle\beta_j|u_a\rangle. \tag{7.62}$$

This is like computing a generalized inner product for each LM: $\langle u_b|q(LM)|u_a\rangle$. Fortunately, there are only about 10 angular momentum indices i,j,LM per block of Q, so each inner product is small. Further, $q_{i,j}^{LM}$ is real.

Because of the small size and mixed-type nature, this operation is hard to perform efficiently using existing libraries. Instead, a custom kernel that blocks over $|u_a\rangle$ can achieve reasonable performance. An example of such a kernel can be found in Listing 7.

7.6.1.4 Two-Particle Local Potential

Given the two-particle density $\rho_{a,b}$, the evaluation of the two-particle local potential $V_{a,b}$ is a Poisson solve Eq. (7.53). The Poisson solve in real space is an element-wise product with the vector $1/|G|^2$ in reciprocal space:

$$V_{nk,mq}(r) = f_b \sum_{g,r'} \langle r|g\rangle \left(\frac{\langle g|r'\rangle \, \rho_{nk,mq}(r')}{|g+k-q|^2} \right), \tag{7.63}$$

where we have reintroduced the explicit n, k, m, q indices and let $\langle r|V|r\rangle \equiv V(r)$ and $\langle r'|\rho|r'\rangle \equiv \rho(r')$ to simplify notation. As before, the FFT is used to pass from real to reciprocal space and back again, $\langle g|r\rangle$ and $\langle r|g\rangle$, respectively. Multiplication by the kernel takes the same form as the multiplication by $|g+k|^2$ in the kinetic energy or $V^{\text{loc}}(r)$ in the local potential. The custom kernel found in Listing 3 can be reused. The FFTs, of which there are $2N_b^2$, are very time consuming.

7.6.1.5 Two-Particle NonLocal Potential

The construction of the nonlocal two particle potential $D_{a,b}$, Eq. (7.55), is the forward tensor projection:

$$D_{i,j}[V_{a,b}] = \sum_{r'} Q_{i,j}(r')\langle r'|r\rangle\langle r|V_{a,b}|r\rangle\langle r|r'\rangle. \tag{7.64}$$

As with the two-particle augmentation density, one can use the intermediate LM transformation to factor the projection, as in Eq. (7.61). Using the forward vector projection to go from r-space to LM-space, the remaining operation is

$$D_{i,j}[V_{a,b}] = \sum_{LM} D_{LM}[V_{a,b}] q_{i,j}^{LM}, \tag{7.65}$$

which is the inverse of Eq. (7.62). Computationally, this is a weighted sum over matrices. Because of the small size and mixed type, a custom kernel is preferable over library calls. An example of such a kernel can be found in Listing 8.

7.6.2 Optimization

The key feature of an efficient parallel implementation of exact exchange on the GPU is a flexible loop-blocking scheme. Such a scheme must facilitate the launch of large data-parallel kernels to fully occupy a single GPU while providing sufficient decomposition for task parallelism over multiple GPUs.

When inserted into an iterative diagonalizer, such as those discussed in Section 7.2.5, the action of the exact-exchange operator must be computed on each band at each k-point. We use the decomposition given by Eq. (7.52) to recognize this as computing the table of pairwise actions $\langle r|\tilde{K}[b]|\tilde{\psi}_a\rangle$ and then summing down the columns.

	$\tilde{\psi}_1$	$\tilde{\psi}_2$	$\tilde{\psi}_3$	$\tilde{\psi}_4$	…
$\tilde{K}[1]$	$\langle r\|\tilde{K}[1]\|\tilde{\psi}_1\rangle$	$\langle r\|\tilde{K}[1]\|\tilde{\psi}_2\rangle$	$\langle r\|\tilde{K}[1]\|\tilde{\psi}_3\rangle$	$\langle r\|\tilde{K}[1]\|\tilde{\psi}_4\rangle$	…
$\tilde{K}[2]$	$\langle r\|\tilde{K}[2]\|\tilde{\psi}_1\rangle$	$\langle r\|\tilde{K}[2]\|\tilde{\psi}_2\rangle$	$\langle r\|\tilde{K}[2]\|\tilde{\psi}_3\rangle$	$\langle r\|\tilde{K}[2]\|\tilde{\psi}_4\rangle$	…
$\tilde{K}[3]$	$\langle r\|\tilde{K}[3]\|\tilde{\psi}_1\rangle$	$\langle r\|\tilde{K}[3]\|\tilde{\psi}_2\rangle$	$\langle r\|\tilde{K}[3]\|\tilde{\psi}_3\rangle$	$\langle r\|\tilde{K}[3]\|\tilde{\psi}_4\rangle$	…
$\tilde{K}[4]$	$\langle r\|\tilde{K}[4]\|\tilde{\psi}_1\rangle$	$\langle r\|\tilde{K}[4]\|\tilde{\psi}_2\rangle$	$\langle r\|\tilde{K}[4]\|\tilde{\psi}_3\rangle$	$\langle r\|\tilde{K}[4]\|\tilde{\psi}_4\rangle$	…
⋮	⋮	⋮	⋮	⋮	
$\sum$	$\langle r\|\tilde{K}\|\tilde{\psi}_1\rangle$	$\langle r\|\tilde{K}\|\tilde{\psi}_2\rangle$	$\langle r\|\tilde{K}\|\tilde{\psi}_3\rangle$	$\langle r\|\tilde{K}\|\tilde{\psi}_4\rangle$	…

We would like to block together elements of the double loop used to compute the table. The two natural options are to block together rows or columns. As written, columns would seem to reuse $|\tilde{\psi}_a\rangle$ when computing the action and accumulate into $\tilde{K}|\tilde{\psi}_a\rangle$, making them the better choice. However, the two-particle potentials in exact exchange are applied to the b-index wavefunction as opposed to the a-index wavefunction for conventional potentials. That is, $\tilde{K}[b]|\tilde{\psi}_a\rangle$ leads to potentials that act on $|\tilde{\psi}_b\rangle$, not $|\tilde{\psi}_a\rangle$. Blocking by column will reuse the accumulation destination, while blocking by row will reuse the b-index wavefunction when computing the action.

The general solution is to block in both: break each row into blocks, but process the blocks by column.

	$\tilde{\psi}_1$	$\tilde{\psi}_2$	$\tilde{\psi}_3$	$\tilde{\psi}_4$	…
$\tilde{K}[1]$	$\langle r\|\tilde{K}[1]\|\tilde{\psi}_1\rangle$	$\langle r\|\tilde{K}[1]\|\tilde{\psi}_2\rangle$	$\langle r\|\tilde{K}[1]\|\tilde{\psi}_3\rangle$	$\langle r\|\tilde{K}[1]\|\tilde{\psi}_4\rangle$	…
$\tilde{K}[2]$	$\langle r\|\tilde{K}[2]\|\tilde{\psi}_1\rangle$	$\langle r\|\tilde{K}[2]\|\tilde{\psi}_2\rangle$	$\langle r\|\tilde{K}[2]\|\tilde{\psi}_3\rangle$	$\langle r\|\tilde{K}[2]\|\tilde{\psi}_4\rangle$	…
$\tilde{K}[3]$	$\langle r\|\tilde{K}[3]\|\tilde{\psi}_1\rangle$	$\langle r\|\tilde{K}[3]\|\tilde{\psi}_2\rangle$	$\langle r\|\tilde{K}[3]\|\tilde{\psi}_3\rangle$	$\langle r\|\tilde{K}[3]\|\tilde{\psi}_4\rangle$	…
$\tilde{K}[4]$	$\langle r\|\tilde{K}[4]\|\tilde{\psi}_1\rangle$	$\langle r\|\tilde{K}[4]\|\tilde{\psi}_2\rangle$	$\langle r\|\tilde{K}[4]\|\tilde{\psi}_3\rangle$	$\langle r\|\tilde{K}[4]\|\tilde{\psi}_4\rangle$	…
⋮	⋮	⋮	⋮	⋮	
$\sum$	$\langle r\|\tilde{K}\|\tilde{\psi}_1\rangle$	$\langle r\|\tilde{K}\|\tilde{\psi}_2\rangle$	$\langle r\|\tilde{K}\|\tilde{\psi}_3\rangle$	$\langle r\|\tilde{K}\|\tilde{\psi}_4\rangle$	…

This scheme results in a blocked action and a local accumulation.

The indices a, b are composite indices over (n, k) and (m, q), respectively. Each pair of BZ indices, (k, q), defines data that is common to the computation of the exchange between wavefunctions at those BZ points, for example, the g-space vector $|g + k - q|^{-2}$ and the phase factors $\exp\,[ik \cdot r]$ and $\exp\,[iq \cdot r]$. Further, PWDFT codes tend to colocate wavefunctions from the same k-point to facilitate blocking and parallelism in conventional calculations. Therefore, it makes sense to parallelize fully over the (k, q) pairs. This strategy conflicts with the aforementioned blocking scheme only if the ideal block size is smaller than the number of wavefunctions per k-point. Such a situation generally corresponds to a small, computationally cheap calculation, so it should not be the focus of a GPU implementation.

The kernels provided in Section 7.6.1 are blocked over one band index: a. It should now be clear why this corresponds to the aforementioned row-blocking scheme. The action kernels in Listings 3 and 4 need to be modified slightly to block over potentials instead of wavefunctions. Additionally, the two-particle potentials are generally complex. These modifications are straightforward, and left to the reader.

The optimizations described for the vector projection in Section 7.3.1 should be reused. Because the $LM \leftrightarrow i, j$ transformations given here batch over ions, the streams from vector projection must be terminated. That is, the vector projection at all ions must be computed first, and then the transformation from LM to i, j (or vice versa). This separation is, of course, artificial and could be removed by inlining the $LM \leftrightarrow i, j$ transformation into the vector projection. We have not found this step necessary, as the $LM \leftrightarrow i, j$ transformation, timed as $\boldsymbol{\rho}_{\mathbf{a},\mathbf{b}}$ and $\mathbf{D}[V_{a,b}]$ in Section 7.6.3, are inexpensive compared FFT and vector projection.

7.6.3 Performance/Examples

The techniques described in Sections 7.6.1 and 7.6.2 have been applied to the VASP code [38].

The profile categories used for the conventional calculation are not all appropriate for exact exchange. The application of the exchange operator dominates the non-exchange action, subspace diagonalization, and the ρ sum. The construction of the two-particle densities and potential, along

Table 7.8 *Structural relaxation of hR105 boron with exact exchange ($N_{ion} = 105$, $N_{band} = 224$, $N_{pw} = 110,592$)*

CPU cores/ GPUs	1/0	1/1	x	8/0	8/1	x	16/0	16/2	x
Fourier	14,669.1	492.8	29.8	2,116.0	235.5	9.0	1,151.2	215.7	5.3
Projection	6,429.1	934.2	6.9	1,262.0	957.4	1.3	690.4	550.5	1.3
$\rho_{a,b}$	1,233.5	220.5	5.6	208.8	119.3	1.7	117.9	116.9	1.0
$V_{a,b}$	367.9	39.5	9.3	66.6	12.6	5.3	39.0	16.7	2.3
$D[V_{a,b}]$	219.6	15.9	13.8	31.2	7.4	4.2	17.6	6.9	2.6
Action	789.4	194.1	4.1	145.3	53.6	2.7	83.6	74.1	1.1
Other	2,188.2	83.5	26.2	399.1	94.2	4.2	1,244.8	72.3	17.2
Total	25,896.6	1,980.5	13.1	4,228.9	1,480.0	2.9	2,344.4	1,053.0	2.2

Times in seconds, GPU speedup denoted by *x*.

with their action, is more informative. The projection category is extended to include the two-sided projection. Fourier transforms are used to compute the two-particle potential, and continue to be a major consumer of computational resources.

- **Fourier**: transform between *r*-space and *g*-space.
- **Projection**: computation of $\langle\beta|\psi\rangle$.
- $\rho_{\mathbf{a},\mathbf{b}}$: computation of two-particle density.
- $\mathbf{V}_{\mathbf{a},\mathbf{b}}$: computation of two-particle local potential.
- $\mathbf{D}[V_{\mathbf{a},\mathbf{b}}]$: transformation of two-particle nonlocal potential.
- **Action**: application of potentials to trial vector.

7.6.3.1 Setup

To demonstrate the performance of exact exchange on the GPU, we will revisit the boron structural minimization calculation from Section 7.5.5, but with exact exchange. Exact exchange is significantly more costly than conventional calculations, so only a single ionic step is taken. Further, a WAVECAR file is used to initialize the wavefunctions, greatly accelerating self-consistent convergence and therefore reducing the overall computational cost. In this case, only three electronic steps are needed. The performance profile is given in Table 7.8.

7.6.3.2 Profile

Fourier transformation and projection are still the two dominant steps, consuming 60% and 25% of the single-core CPU time, respectively. The computation of the two-particle density and the action each take around 10% of the runtime. The balance is spent computing the two-particle local and nonlocal potentials, which is reasonable given that those times exclude Fourier transformation and projection.

7.6.3.3 Single-Socket

The FFT performs nearly 9× faster on the GPU than a CPU socket. Considering that the GPU performs the FFT in single precision, this is very close to the 9.6× max speedup for mixed-precision memory-bound computations. On the other hand, projection is only 1.3× faster on the GPU. The performance with one and eight cores is comparable, and the projection is streamed, so this is unlikely due to occupancy. More likely, the very sparse scatter/gather memory access patterns are unable to take advantage of high-bandwidth coalesced memory accesses. The performance of the two-particle density and action kernels is poor due to low occupancy. The two-particle potentials

perform reasonably well because of the large kernel size, but do not make up a significant fraction of the overall run time.

7.6.3.4 Scaling

The Fourier transforms on the GPU scale favorably from one core to eight cores, considering that the GPU resources are fixed. This indicates that the Fourier transforms from a single core are not sufficient to saturate the GPU. Adding a second GPU doesn't improve performance, so the Fourier transformation for this problem likely fits within a single GPU.

Projection, on the other hand, does not scale from one to eight cores, but does scale with additional GPUs. Adding a second GPU doubles the memory bandwidth for the scatter/gather, improving performance by 1.7×. The other operations are too small to reasonably expect to benefit from spreading over multiple GPUs. Fortunately, they take up only a small fraction of the overall runtime, so scalability is predominantly limited by Fourier transforms.

7.7 Summary and Outlook

The principal steps taken to obtain the electronic structure of a system using a plane-wave basis set have been reviewed for standard and hybrid DFT. Evaluation of the kinetic, local, and nonlocal energy components has been detailed mathematically. The numerical implementation of the energy components has been discussed, bearing in mind their use on GPGPU and more general coprocessor architectures. The optimization of actual routines from PWDFT for GPU has been described with a special focus on parallel optimization both on multi-CPU and multi-GPU platforms. Illustration of this methodology to accelerate the plane-wave VASP code concludes our work: it has been shown that with proper tuning, a single GPU is generally faster than 16 CPU cores. As the amount of computation increases, either by increasing the size of the simulated system or by using more costly hybrid functionals, so does the efficiency of GPUs. Adding two GPUs to a dual socket node can improve absolute performance by 2–3×.

The methods described in this chapter can provide a base for future implementations of PWDFT on GPU and coprocessor architectures, but must be supplemented by deep profiling to account for platform-specific resources, bandwidths, and latencies. GPU technology is progressing quickly, offering larger and faster memory, more processors, and increased functionality. The future thus promises even better code performance on GPUs.

Acknowledgments

The authors would like to warmly thank S. Steinman, M. Widom, T. Guignon for thoughtful comments, as well as all the members of the Nvidia VASP team, who continue to support and advance VASP on the GPU.

References

1. Hohenberg, P. and Kohn, W. (1962) Inhomogeneous electron gas. *Phys. Rev.*, **155**, 1964.
2. Kohn, W. and Sham, L.J. (1965) Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, **140**, A1133–A1138.
3. Martin, R.M. (2004) *Electronic Structure: Basic Theory and Practical Methods*, Cambridge University Press.
4. Dedieu, A. (2000) Theoretical studies in palladium and platinum molecular chemistry. *Chem. Rev.*, **100** (2), 543–600.

5. Cundari, T.R. (ed.) (2001) *Computational Organometallic Chemistry*, Marcel Dekker, New York.
6. Sousa, S.F., Fernandes, P.A. and Ramos, M.J. (2007) General performance of density functionals. *J. Phys. Chem. A*, **111** (42), 10439–10452.
7. Bachrach, S.M. (2007) *Computational Organic Chemistry*, Wiley-Interscience, Hoboken, NJ.
8. Slater, J. and Koster, G. (1954) Simplified LCAO method for the periodic potential problem. *Phys Rev.*, **94**, 1498–1524.
9. Feenstra, R.M., Srivastava, N., Gao, Q., Widom, M., Diaconescu, B., Ohta, T. *et al*. (2013) Low-energy electron reflectivity from graphene. *Phys. Rev. B*, **87** (4), 041406.
10. Hamann, D.R., Schlüter, M. and Chiang, C. (1979) Norm-conserving pseudopotentials. *Phys. Rev. Lett.*, **43**, 1494–1497.
11. Vanderbilt, D. (1990) Soft self-consistent pseudopotentials in a generalized eigenvalue formalism. *Phys. Rev. B*, **41**, 7892–7895.
12. Kresse, G. and Joubert, D. (1999) From ultrasoft pseudopotentials to the projector augmented-wave method. *Phys. Rev. B*, **59** (3), 11–19.
13. Blöchl, P.E. (1994) Projector augmented-wave method. *Phys. Rev. B*, **50** (24), 17953–17979.
14. Giannozzi, P., Baroni, S., Bonini, N., Calandra, M., Car, R., Cavazzoni, C. *et al*. (2009) QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. *J. Phys. Condens. Matter*, **21** (39), 395502.
15. Kresse, G. and Furthmüller, J. (1996) Efficiency of Ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. *Comput. Mater. Sci.*, **6** (1), 15–50.
16. Segall, M., Lindan, P.J., Probert, M., Pickard, C., Hasnip, P., Clark, S. *et al*. (2002) First-principles simulation: ideas, illustrations and the CASTEP code. *J. Phys. Condens. Matter*, **14** (11), 2717.
17. Clark, S.J., Segall, M.D., Pickard, C.J., Hasnip, P.J., Probert, M.I., Refson, K. *et al*. (2005) First principles methods using CASTEP. *Z. Kristallogr.*, **220** (5–6), 567–570.
18. Gygi, F., Yates, R.K., Lorenz, J., Draeger, E.W., Franchetti, F., Ueberhuber, C.W. *et al*. (2005) Large-scale first-principles molecular dynamics simulations on the Bluegene/l platform using the Qbox code. Proceedings of the 2005 ACM/IEEE conference on Supercomputing, IEEE Computer Society, p. 24.
19. Gygi, F. (2008) Architecture of Qbox: a scalable first-principles molecular dynamics code. *IBM J. Res. Dev.*, **52** (1.2), 137–144.
20. Vanderbilt, D. (1990) Soft self-consistent pseudopotentials in a generalized eigenvalue formalism. *Phys. Rev. B*, **25** (23), 4228.
21. Marx, D. and Hutter, J. (2009) *Ab Initio Molecular Dynamics: Basic Theory and Advanced Methods*, Cambridge University Press, New York.
22. Wood, D.M. and Zunger, A. (1985) A new method for diagonalising large matrices. *J. Phys. A: Math. Gen.*, **18** (9), 1343.
23. Teter, M.P., Payne, M.C. and Allan, D.C. (1989) Solution of Schrödinger's equation for large systems. *Phys. Rev. B*, **40**, 12255–12263.
24. Bylander, D.M., Kleinman, L. and Lee, S. (1990) Self-consistent calculations of the energy bands and bonding properties of $B_{12}C_3$. *Phys. Rev. B*, **42**, 1394–1403.
25. Davidson, E.R. (1975) The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *J. Comput. Phys.*, **17** (1), 87–94.
26. Davidson, E.R. (1978) in *Report on the Workshop "Numerical Algorithms in Chemistry: Algebraic Methods"* (eds C. Moler and I. Shavitt), University of California, Lawrence Berkeley Laboratory, p. 15.
27. Liu, B. (1978) in *Report on the Workshop "Numerical Algorithms in Chemistry: Algebraic Methods"* (eds C. Moler and I. Shavitt), University of California, Lawrence Berkley Laboratory, p. 49.

28. Davidson, E.R. (1983) in *Methods in Computational Molecular Physics* (eds G.H.F. Diercksen and S. Wilson), Plenum Publishing Corporation, New York, p. 95.
29. Kresse, G. and Furthmüller, J. (1996) Efficient iterative schemes for Ab initio total-energy calculations using a plane-wave basis set. *Phys. Rev. B*, **54** (16), 11169–11186.
30. Mostofi, A.A., Haynes, P.D., Skylaris, C.K. and Payne, M.C. (2003) Preconditioned iterative minimization for linear-scaling electronic structure calculations. *J. Chem. Phys.*, **119** (17), 8842–8848.
31. Teter, M.P., Payne, M.C. and Allan, D.C. (1989) Solution of Schrödinger's equation for large systems. *Phys. Rev. B*, **40**, 12255–12263.
32. Parlett, B.N. (1980) *The Symmetric Eigenvalue Problem*, Prentice Hall, Englewood Cliffs, NJ.
33. Pulay, P. (1980) Convergence acceleration of iterative sequences. The case of SCF iteration. *Chem. Phys. Lett.*, **73** (2), 393–398.
34. CUDA BLAS Library, Available from `http://developer.nvidia.com/cublas` (accessed 30 September 2015).
35. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J. *et al*. (2009) Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *J. Phys. Conf. Ser.*, **180**, 012037.
36. Methfessel, M. and Paxton, A. (1989) High-precision sampling for Brillouin-zone integration in metals. *Phys. Rev. B*, **40** (6), 3616–3621.
37. Marzari, N., Vanderbilt, D., Vita, A.D. and Payne, M. (1999) Thermal contraction and disordering of the Al (110) surface. *Phys. Rev. Lett.*, **82** (16), 3296–3299.
38. Hutchinson, M. and Widom, M. (2012) VASP on a GPU: application to exact-exchange calculations of the stability of elemental boron. *Comput. Phys. Commun.*, **1**, 1–5.
39. Spiga, F. and Girotto, I. (2012) phiGEMM: a CPU-GPU library for porting Quantum ESPRESSO on hybrid systems. 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP).
40. Maintz, S., Eck, B. and Dronskowski, R. (2011) Speeding up plane-wave electronic-structure calculations using graphics-processing units. *Comput. Phys. Commun.*, **1**, 1–7.
41. Hacene, M., Anciaux-Sedrakian, A., Rozanska, X., Klahr, D., Guignon, T. and Fleurat-Lessard, P. (2012) Accelerating VASP electronic structure calculations using graphic processing units. *J. Comput. Chem.*, **33** (32), 2581–2589.
42. Tariq, S., Bédorf, J., Stosic, D., Anciaux-Sedrakian, A., Hutchinson, M., Widom, M. *et al*. (2015) GPU Acceleration of the Block Davidson, RMM-DIIS and Exact-Exchange algorithms of VASP, to be submitted.
43. Monkhorst, H.J. and Pack, J.D. (1976) Special points for Brillouin-zone integrations. *Phys. Rev. B*, **13**, 5188–5192.
44. Perdew, J.P. and Wang, Y. (1992) Accurate and simple analytic representation of the electron-gas correlation energy. *Phys. Rev. B*, **45**, 13244.
45. Perdew, J.P., Chevary, J.A., Vosko, S.H., Jackson, K.A., Pederson, M.R., Singh, D.J. *et al*. (1992) Atoms, molecules, solids, and surfaces: applications of the generalized gradient approximation for exchange and correlation. *Phys. Rev. B*, **46**, 6671–6687.
46. Perdew, J.P., Burke, K. and Ernzerhof, M. (1996) Generalized gradient approximation made simple. *Phys. Rev. Lett.*, **77**, 3865–3868.
47. Nocedal, J. and Wright, S.J. (2006) *Numerical Optimization*, Springer Series in Operations Research, 2nd edn, Springer-Verlag, New York.
48. Becke, A. (1993) A new mixing of Hartree-Fock and local density?functional theories. *J. Chem. Phys.*, **98** (2), 1372–1377.
49. Paier, J., Hirschl, R., Marsman, M. and Kresse, G. (2005) The Perdew-Burke-Ernzerhof exchange-correlation functional applied to the G2-1 test set using a plane-wave basis set. *J. Chem. Phys.*, **122** (23), 234102.

Appendix A: Definitions and Conventions

$\hat{H}$	Hamiltonian operator
$\hat{T}$	Kinetic energy operator
$H(k)$	k-Dependent Hamiltonian matrix
$\hat{V}^{\text{loc}}$	Local potential
$\hat{V}^{\text{nl}}$	Non-local potential
$\tilde{\Psi}$	Pseudo-wavefunction (in PAW)
$\hat{\phi}_n$	Approximate wavefunction in iterative diagonalization
n_e	Number of electrons
N_{pw}	Number of plane waves
N_{band}	Number of bands included in calculation
N_e	Bands with non zero occupation. $N_e \gtrsim n_e$
Ω	Unit Cell Volume
$f_{n,k}$	Occupancy of monoelectronic wavefunction ψ_{nk}
ψ_{nk}	Monoelectronic wavefunction
u_{nk}	Periodic part of the monoelectronic function. As the wavefunction ψ_{nk} is closely related to u_{nk} by the Bloch theorem, in the following, we will often use an approximate shortcut and refer to u_{nk} as the wavefunction.

Appendix B: Example Kernels

We provide here reference implementations of common kernels in plane-wave DFT. These kernels can be found in source form on Github.[2].

```
static __global__ void proj_forward_k(cuDoubleComplex *phase,
                                       cuDoubleComplex *u_r,
                                       cuDoubleComplex *u_rp,
                                       int *perm,
                                       int size,
                                       int p_size,
                                       int nband
                                      ){
  const int tdx     = threadIdx.x + blockIdx.x * blockDim.x;
  const int nthread = blockDim.x * gridDim.x;
  int idx;
  cuDoubleComplex phase_local;

  for (int i = tdx; i < p_size; i += nthread){
    idx         = perm[i] - 1;
    phase_local = phase[i];
    for (int j = blockIdx.y; j < nband; j += gridDim.y){
      u_rp[i + j*p_size].x = phase_local.x * u_r[idx + j*u_size].x
                           - phase_local.y * u_r[idx + j*u_size].y;
      u_rp[i + j*p_size].y = phase_local.x * u_r[idx + j*u_size].y
                           + phase_local.y * u_r[idx + j*u_size].x;
    }
  }
}
```

Listing 1 Forward projection in real-space

[2] https://github.com/maxhutch/pwdft_supplement.

```
static __global__ void proj_reverse_k(cuDoubleComplex *phase,
                                      cuDoubleComplex *u_r,
                                      cuDoubleComplex *u_rp,
                                      int *perm,
                                      int u_size,
                                      int p_size,
                                      int nband
                                     ){
  const int tdx     = threadIdx.x + blockIdx.x * blockDim.x;
  const int nthread = blockDim.x * gridDim.x;
  int ind;
  cuDoubleComplex phase_l;

  for (int i = tdx; i < p_size; i += nthread){
    idx     = perm[i] - 1;
    phase_l = phase[i];
    for (int j = blockIdx.y; j < nband; j += gridDim.y){
      u_r[idx + j*u_size].x += phase_l.x * u_rp[i + j*p_size].x
                             + phase_l.y * u_rp[i + j*p_size].y;
      u_r[idx + j*u_size].y += phase_l.x * u_rp[i + j*p_size].y
                             - phase_l.y * u_rp[i + j*p_size].x;
    }
  }
}
```

Listing 2 Reverse projection in real-space

```
static __global__ diagonal_kernel(
                                  double *g2_or_V,
                                  cuDoubleComplex *u,
                                  int u_size,
                                  int num
                                 ){
  const int tdx     = threadIdx.x + blockIdx.x * blockDim.x;
  const int nthread = blockDim.x * gridDim.x;

  for (int i = tdx; i < u_size; i+= nthread){
    for (int j = blockIdx.y; j < num; j+= gridDim.y){
      u[i + j*u_size] = u[i + j*u_size] * g2_or_V[i];
    }
  }
}
```

Listing 3 Diagonal action kernel

```
static __global__ vnl_uspp_kernel(double *Vnl,
  /* <\beta_i|u_nk> */              cuDoubleComplex *betaU,
  /* <\beta_j|V^{nl}|u_nk> */       cuDoubleComplex *VnlBetaU,
                                    int nion,
                                    int N_l,
                                    int num
                                   ){
  __shared__ Vnl_s[ N_l*N_l ];
  cuDoubleComplex VnlBetaU_local;

  // one block per ion
  for (i = blockIdx.x; i < nion; i+=gridDim.x){
    // Store Vnl in shared per block
    for (j = threadIdx.x; j < N_l*N_l; j++)
      Vnl_s[j] = Vnl[j + i * N_l * N_l];
    __syncthreads();

    // one thread per projection
    for (j = threadIdx.x; j < N_l*num; += blockDim.x){
      l1 = j % N_l;
      VnlBetaU_local = 0.;
      for (l2 = 0; l2 < N_l; l2++){
        VnlBetaU_local += Vnl[l1+N_l*l2] * betaU[l2 + i*N_l + (j/N_l)*nion*N_l];
      }
      VnlBetaU[l1 + i*N_l + (j/N_l)*nion*N_l] = VnlBetaU_local;
    }
  }
}
```

Listing 4 Nonlocal potential kernel

```
static __global__ square_sum_kernel(cuDoubleComplex *u_r,
                                    double *rho,
                                    double *occ,
                                    int size,
                                    int num
                                   ){
  const int idx      = threadIdx.x + blockIdx.x * blockDim.x;
  const int nthreads = blockDim.x * gridDim.x;
  double rho_l;

  for (i = idx; i < size; i+=nthreads){
    rho_l = 0.;
    for (j = threadIdx.y; j < num; j+= blockDim.y){
      rho_l += occ[j] *
               ( u_r[i+j*size].x * u_r[i+j*size].x
               - u_r[i+j*size].y * u_r[i+j*size].y );
    }
    atomicAdd(rho + i, rho_l);
  }
}
```

Listing 5 Density accumulation kernel

```
static __global__ void el_prod_k(
                                  cuDoubleComplex *uA_r,
                                  cuDoubleComplex *uB_r,
                                  cuDoubleComplex *rhoAB,
                                  int n,
                                  int nband
                                 ){
  const int nthreads = blockDim.x * gridDim.x;
  const int idx = threadIdx.x + blockIdx.x * blockDim.x;
  cuDoubleComplex uA_l;

  // Loop over spatial index
  for (int i = idx; i < n; i+= nthreads){
    uA_l = uA_r[i]; // load to register

    // Loop over B
    for (int j = blockIdx.y; j < nband; j+= gridDim.y){
     // Compute <r_i|u_A> <u_B|r_i>
     rhoAB[i+j*n].x = uB_r[i+j*n].x * uA_l.x + uB_r[i+j*n].y * uA_l.y;
     rhoAB[i+j*n].y = uB_r[i+j*n].y * uA_l.x - uB_r[i+j*n].x * uA_l.y;
    }
  }
}
```

Listing 6 Plane-wave two-particle density kernel

```
static __global__ void IJ_trans_LM_k(
                        cuDoubleComplex *uA_b, //!< block of wavefunctions
                        cuDoubleComplex *uB_b, //!< one wavefunction
                        double* q_ijlm, //!< transformation matrix
                        cuDoubleComplex *rhoAB_lm, //!< output
                        int nion, //!< number of ions of this type
                        int ij_max, //!< lmmax for wavefunction of this type
                        int npro_w, //!< size of wavefunction projection
                        int lm_max, //!< lmmax for projectors of this type
                        int npro_p, //!< total projectors per band
                        int q_dim, //!< extent of first 2 dims of q
                        int nband //!< number of bands
                                  ){

  cuDoubleComplex tmp;

  cuDoubleComplex *uA_p, *rhoAB_p;
  int i, j, a, lm, comp, band;

  /* Blocks loop over bands */
  for (band = blockIdx.x; band < nband; band += gridDim.x){
    uA_p = uA_b + band * npro_w; rhoAB_p = rhoAB_lm + band * npro_p;
    /* Threads loop over (ions, LM numbers) */
    for (comp = threadIdx.x; comp < nion*lm_max; comp += blockDim.x){
      a = comp / lm_max; lm = comp % lm_max;
      /* Inner product < uB | q(LM) | uA >  */
      for (j = 0; j < ij_max; j++){
        for (i = 0; i < ij_max; i++){
          rhoAB_p[comp].x += q_ijlm[lm*Q_dim*Q_dim + j*Q_dim + i]
                           * (uA_p[a*ij_max + j].x * uB_b[a*ij_max + i].x
                            + uA_p[a*ij_max + j].y * uB_b[a*ij_max + i].y
                             );
          rhoAB_p[comp].y += q_ijlm[lm*Q_dim*Q_dim + j*Q_dim + i]
                           * (uA_p[a*ij_max + j].y * uB_b[a*ij_max + i].x
                            - uA_p[a*ij_max + j].x * uB_b[a*ij_max + i].y
                             );
        }
      }
    }
  }
}
```

Listing 7 $i,j \rightarrow LM$ transform

```
static __global__ void LM_trans_IJ_k(
                        cuDoubleComplex* Dij, //!< (i,j) non-local potential
                        cuDoubleComplex* Dlm, //!< (LM) non-local potential
                        double* q_ijlm, //!< q_{i,j}^{LM} transformation
                        int nion, //!< number of ions of this type
                        int ij_max, //!< number of (i,j) angular momentum
                        int lm_max, //!< number of LM total angular momentum
                        int q_dim, //!< dimesion of D (q_dim >= ij_max)
                        int size_ij, //!< size of each band's D_{i,j}
                        int size_lm, //!< size of each band's D_{LM}
                        int nband //!< number of bands
                                  ){
  cuDoubleComplex *Dlm_p, Dije;
  int a, i, j, lm, k, band;

  /* Blocks loop over bands */
  for (band = blockIdx.x; band < nband; band += gridDim.x){
    Dlm_p = Dlm + band * size_lm;
    for (a = 0; a < nion; a++){
      /* Threads loop over i,j angular momenta (padded to q_dim) */
      for (ij = threadIdx.x; ij < q_dim*ij_max; ij += blockDim.x){
        Dije = Dij[ij + a * q_dim * q_dim + band*size_ij];
        /* Weighted sum of q_ijlm */
        for (lm = 0; lm < lm_max; lm++){
           Dije.x += Dlm_p[lm + a * lm_max + band * size_lm].x
                  * q_ijlm[ij + lm * q_dim * q_dim];
           Dije.y += Dlm_p[lm + a * lm_max + band * size_lm].y
                  * q_ijlm[ij + lm * q_dim * q_dim];
        }
        Dij[ij + a * q_dim * q_dim + band*size_ij] = Dije;
      }
    }
  }
}
```

Listing 8 $LM \rightarrow i,j$ transform

8

GPU-Accelerated Sparse Matrix–Matrix Multiplication for Linear Scaling Density Functional Theory

Ole Schütt[1], *Peter Messmer*[2,3], *Jürg Hutter*[4] *and Joost VandeVondele*[1]

[1]*Department of Materials, ETH Zürich, Zürich, Switzerland*
[2]*NVIDIA, Zürich, Switzerland*
[3]*NVIDIA Co-Design Lab for Hybrid Multicore Computing, Zürich, Switzerland*
[4]*Institute of Physical Chemistry, University of Zürich, Zürich, Switzerland*

This chapter discusses how GPUs can be exploited to accelerate sparse matrix–matrix multiplications as required to solve the self-consistent field equations in linear scaling density functional theory calculations. We present the DBCSR sparse matrix multiplication library and describe the algorithms that are used to achieve maximum performance on distributed multicore and hybrid CPU/GPU architectures. Our numerical results demonstrate the efficiency of this linear scaling GPU-based implementation on supercomputers for very large simulations. This paves the way for scientific applications based on three-dimensional models consisting of 10,000 atoms or more.

8.1 Introduction

8.1.1 Linear Scaling Self-Consistent Field

With the steady increase in computer power available, larger and larger systems are simulated using electronic structure methods. These large simulations expose the asymptotic scaling of the traditional algorithms used in electronic structure calculations. Many of the traditional algorithms have a cubic or higher scaling with system size, effectively blocking the path to very large scale simulations.

Electronic Structure Calculations on Graphics Processing Units:
From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

However, it is known that effective interactions are short-ranged for many systems, which can be exploited in linear scaling methods [1, 2]. Consequently, much effort has been spent in developing algorithms with a computational cost that scales linearly with system size. Originally, most applications and benchmarks of these methods were restricted to systems with a quasi-one-dimensional structure, where the prefactor of linear scaling methods is very favorable. Now, the huge increase in computational power and the refinement of the algorithms have made it possible to study scientifically relevant three-dimensional (3D) systems. Basis sets of good quality and tight numerical thresholds can be employed, essentially allowing an accuracy that is identical to that of the cubic scaling methods.

For the important class of mean field methods, for example, Hartree–Fock and Kohn–Sham (KS) density functional theory (DFT), linear scaling methods have to address the buildup of the Hamiltonian matrix (KS matrix) and the solution of the self-consistent field (SCF) equations (see Figure 8.1). Many different algorithms for these two tasks have been proposed, and a detailed discussion can be found in a recent review [2]. Here, we concentrate on methods for the solution of the KS equation, that is, replacements for the KS matrix diagonalization, that directly calculate the one-particle density matrix and are implemented in the CP2K simulation package [3, 4]. The impact of such methods on the computational cost of 3D systems can be inferred from the curves in Figure 8.2, where the simulation times for the calculation of the electronic structure of bulk liquid water for conventional cubically scaling and linear scaling methods are compared.

The most basic algorithm investigated is based on the matrix sign function, which can be defined as

$$\mathrm{sign}(A) = A(A^2)^{-(1/2)}. \tag{8.1}$$

For diagonalizable A, the eigenvectors of A are the eigenvectors of $\mathrm{sign}(A)$, with the eigenvalues of $\mathrm{sign}(A)$ being -1 or $+1$ for negative or positive eigenvalues of A, respectively. Various simple iterative

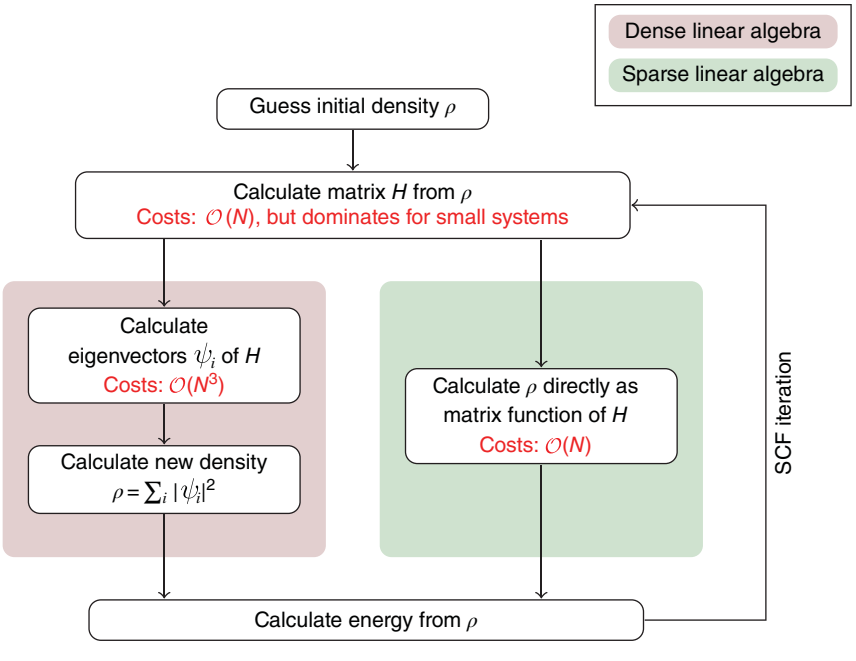


Figure 8.1 *Workflow for a self-consistent electronic structure calculation, illustrating both the use of traditional $O(N^3)$ as well as $O(N)$ methods*

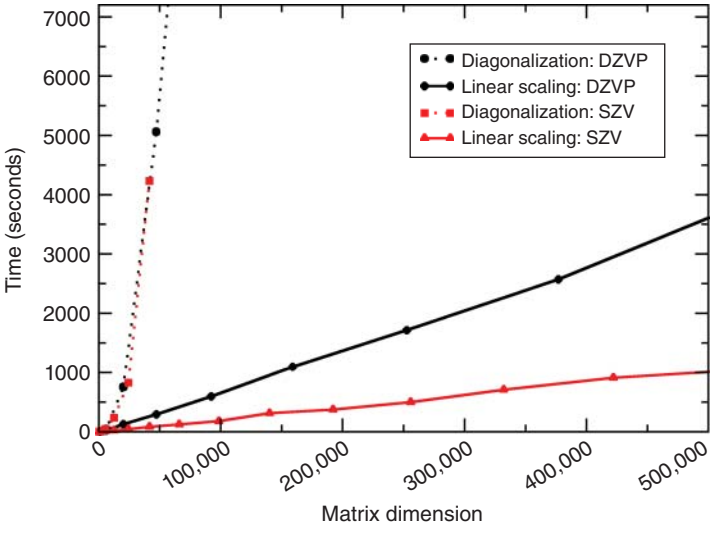


Figure 8.2 *Direct comparison of the time needed for calculations on bulk liquid water using linear scaling and diagonalization-based SCF procedures. For matrices larger than 20,000 basis functions, a speedup is observed (filtering threshold 10^{-5}). Note that for linear scaling approaches, the time needed depends not only on the matrix size but also on the sparsity pattern, and hence better quality basis sets typically have a larger relative cost*

algorithms are available to compute the matrix sign function [5], and these approaches have found early application [6, 7]. These algorithms converge super-linearly and are numerically stable. The simplest form, which only requires two matrix multiplies per iteration, is (I is the identity matrix)

$$X_{n+1} = \frac{1}{2} X_n (3I - X_n^2). \tag{8.2}$$

For $X_0 = cA$ and $c < \|A\|^{-1}$, this iteration converges quadratically to $X_\infty = \text{sign}(A)$. The convergence criterion employed terminates the iteration at X_{n+1} if $\|I - X_n^2\|_F < \sqrt{\epsilon_{\text{filter}}}\|X_n^2\|_F$, where $\|.\|_F$ is the Frobenius norm. Since the algorithm is quadratically convergent, near convergence each iteration will approximately double the number of correct digits in the solution.

Linear scaling results from the fact that all matrix operations are performed on sparse matrices, which have a number of nonzero entries per row that is independent of system size. In order to retain sparsity during the iterations, a threshold (ϵ_{filter}) is employed to set small entries to zero after multiplication, thereby reducing the data volume and speeding up the following multiplies.

The density matrix P corresponding to a given Hamiltonian matrix H, overlap matrix S and chemical potential μ can be computed as

$$P = \frac{1}{2}(I - \text{sign}(S^{-1}H - \mu I))S^{-1}. \tag{8.3}$$

The important idempotency (PSPS = PS) and commutativity (SPH-HPS=0) conditions, equivalent to wave function orthonormality, are automatically satisfied. The number of electrons N_{el} is determined by the chemical potential μ, and can be obtained from $N_{\text{el}} = \text{trace}(PS)$. S^{-1} is computed conveniently using $S^{-1} = S^{-(1/2)}S^{-(1/2)}$ where the square root and inverse square root can be obtained from

$$\text{sign}\left(\begin{bmatrix} 0 & \text{A} \\ I & 0 \end{bmatrix}\right) = \left(\begin{bmatrix} 0 & A^{(1/2)} \\ A^{-(1/2)} & 0 \end{bmatrix}\right). \tag{8.4}$$

A stationary solution of the self-consistent equations can be obtained by a simple mixing approach:

$$P_{n+1} = \frac{1}{2}(I - \mathrm{sign}(S^{-1}\hat{H}_n - \mu_n I))S^{-1},$$
$$\hat{H}_{n+1} = (1-\alpha)\hat{H}_n + \alpha H_{n+1},$$

where α is a mixing parameter between 0 and 1, and $\hat{H}_n$, an auxiliary matrix. The fixed point implies that $\hat{H}_n = H_n$ and thus $SP_nH_n - H_nP_nS = 0$. For each iteration, the total electronic energy (E_n) and Hamiltonian matrix (H_n) are computed from the density matrix P_n. The value of the chemical potential μ_n is determined by bisecting a suitable interval until $|\mathrm{trace}(P_{n+1}S) - N_{\mathrm{el}}| < \frac{1}{2}$, for a given N_{el}. Note that the trace($P_{n+1}S$) is integer-valued unless finite accuracy is employed in the calculation of the sign function. For a given SCF threshold (ϵ_{SCF}), the convergence criterion employed is $E_n - E_{n-1} < \epsilon_{\mathrm{SCF}}N_{\mathrm{el}}$.

More advanced algorithms that still use fix point iterations exist. They include the optimization of the chemical potential [8] as part of the density matrix computation, and achieve faster convergence by relaxing absolute trace conservation [9]. These methods represent a significant advantage over the sign matrix iteration, if the chemical potential is not known in advance, as the cumbersome bisection can be omitted. Also trace resetting (TRS) purification starts from a normalized Hamiltonian matrix (X_0, eigenvalues in the interval [0, 1]). The algorithm then calls for iterations where the update depends on the value of the quantity $\gamma_n = (N - \mathrm{trace}(\mathcal{F}(X_n)))/\mathrm{trace}(\mathcal{G}(X_n))$.

γ_n	Update
$\gamma_n > \gamma_{\max}$	$X_{n+1} = 2X_n - X_n^2$
$\gamma_n < \gamma_{\min}$	$X_{n+1} = X_n^2$
$\gamma_n \in [\gamma_{\min}, \gamma_{\max}]$	$X_{n+1} = \mathcal{F}(X_n) + \gamma_n\mathcal{G}(X_n)$

The choice of the polynomial functions $\mathcal{F}$ and $\mathcal{G}$ is not unique, but an efficient algorithm (TRS4) is achieved by using

$$\mathcal{F}(x) = x^2(4x - 3x^2), \tag{8.5}$$
$$\mathcal{G}(x) = x^2(1-x)^2. \tag{8.6}$$

For this choice of polynomials the values for $\gamma_{\min}$ and $\gamma_{\max}$ are 0 and 6, respectively.

Another class of algorithms aims at a direct minimization of the energy functional, avoiding the self-consistent mixing, and thus adding robustness. To achieve this, the constraints on the density matrix have to be included into the algorithm. In the work of Li *et al.* [10] this was achieved by using an extended energy functional. Helgaker *et al.* [11]. proposed a parameterization of the density matrix, that conserves idempotency. Within this curvy-step method [12, 13], starting from an idempotent P_0, as obtained for example from the TRS method, one performs updates of the form

$$P_{n+1} = e^{-\Delta S}P_n e^{\Delta S}, \tag{8.7}$$

where Δ is an anti-Hermitian matrix, $\Delta^\dagger = -\Delta$. This unitary transformation is evaluated using the Baker–Campbell–Hausdorff expansion

$$P_{n+1} = P_n + [P_n, \Delta]_S + \frac{1}{2}[[P_n, \Delta]_S, \Delta]_S + \frac{1}{6}[[[P_n, \Delta]_S, \Delta]_S, \Delta]_S + \cdots, \tag{8.8}$$

where the commutator within a nonorthogonal basis is

$$[X, \Delta]_S = XS\Delta - \Delta SX. \tag{8.9}$$

In the minimization, the matrix elements of the curvy-step matrix Δ are the free variables and are calculated from the energy gradient

$$\frac{\partial E}{\partial \Delta} = [H, P_n]_S \tag{8.10}$$

using, for example, a steepest descent, conjugate gradient, or a Newton–Raphson method.

All of the above algorithms have in common that matrix multiplication is the dominant operation. The performance of the underlying sparse matrix multiplication routines is of paramount importance for the overall computational efficiency.

8.1.2 DBCSR: A Sparse Matrix Library

The linear scaling SCF implementation in CP2K is centered around sparse matrix–matrix multiplication [3, 4]. This choice is motivated by the fact that matrix multiplication is a basic primitive that is suitable for a high performance parallel implementation. Furthermore, this operation can be used to compute matrix functionals, such as, for example, inv, sqrt, sign, and exponential. Surprisingly, no established software library is available that performs a parallel sparse matrix–matrix multiplication. Such a library should, in the context of quantum chemistry, exploit the concept of sub-matrix blocks, rather than individual elements for a description of the sparsity pattern. These sub-matrix blocks, also named atomic blocks as they correspond to basis functions of an atom, are small (typical numbers are 5, 13, 23), and exploiting them is key to achieve good performance. Furthermore, as most calculations are currently performed near the cross-over regime between dense and sparse, the library must be highly efficient for relatively high occupations (e.g., 50% non-zero elements), and 10,000s of non-zeros per row, while optimal performance for very sparse matrices (<1–5% non-zero elements) will become more important in the future. In order to address these needs, a general purpose sparse matrix library has been developed [14]. This library is currently distributed as part of the CP2K package, but it is our aim to provide a general purpose sparse matrix library that can ultimately be made available as a fully independent tool. The name of the library is DBCSR, which is an abbreviation for Distributed Blocked Compressed Sparse Row or Distributed Blocked Cannon Sparse Recursive. The full names emphasize the storage format or the multiplication algorithm, respectively. Data is stored distributed over all processes, using a blocked variant of the compressed sparse row storage format. The parallel algorithm to perform the matrix–matrix multiplication is based on the Cannon scheme [15], which is optimal in the dense case, if memory is a limited resource. In particular, it guarantees that communication per process decreases with increasing process count, and is free from all-to-all communication. These properties guarantee strong scaling of the algorithm, and good performance in the dense limit. Nevertheless, sparse matrix multiplication has $O(N)$ flops and $O(N)$ data, and reaching peak performance is thus difficult. The ratio of flops to data does not depend on system size, but rather on the number of non-zeros per row. The latter depends typically on the accuracy of the calculation, such as tighter filtering thresholds and larger basis sets in our quantum chemical applications. We refer to Ref. [14] for an in-depth discussion. In the following, we focus on those aspects that are important in the context of GPU-acceleration, and on the recent developments that have enabled a significant increase in accelerated performance.

8.2 Software Architecture for GPU-Acceleration

In this section, we outline the various layers of the DBCSR matrix multiplication architecture. It has been designed to decouple the various steps of the calculation, and is schematically shown in Figure 8.3. As we go down the layers, the granularity of the data becomes smaller and

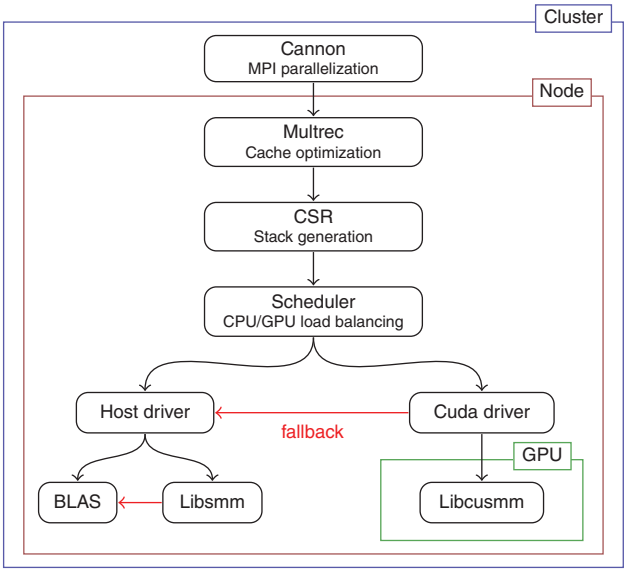


Figure 8.3 *A schematic representation of the software architecture employed in the GPU accelerated DBCSR library. The various layers correspond to key steps in the matrix multiplication algorithm. While the Cannon layer is essential for the parallelism between processes or on the cluster level, the lower layers deal with parallelism and acceleration on the node level*

the computational workload increases. The higher layers manage data transfers, optimize data access, and enable asynchronous progress. These steps are essential to fully benefit from the high performance that modern CPUs and GPUs offer.

8.2.1 Cannon Layer

The top-most layer deals with the parallelization of the matrix multiplication over the nodes of a cluster, and enables good parallel performance by managing the message passing between MPI processes. One MPI process can consist of several CPU threads, based on OpenMP, and can off-load to a dedicated or shared GPU. For the MPI-parallelization, the sparse matrices are divided into large sparse sub-matrices named panels. These panels are regular in shape, and by a suitable row and column permutation, the sparsity pattern has been homogenized so that all panels contain approximately the same amount of data, which is favorable for load-balancing the calculations. These panels are distributed over a regular 2D grid of processes and *Cannon's algorithm* [15] is used to communicate these panels between processes in a regular and ordered fashion, according to "ticks" in the Cannon metronome. In each tick of Cannon's algorithm, each process sends and receives two panels, multiplies the two panels that are available, and accumulates the results locally. As discussed below, messages can be processed asynchronously, that is, be in transit over the network, while computation takes place. The panel data are also uploaded to the GPU in this layer. Our approach to enable the asynchronous message passing and uploads will be discussed in more detail below. The following, lower layers deal with the node local multiplication of panels.

8.2.2 Multrec Layer

The *multrec* layer is a high level node-local layer that aims at optimizing memory access, in particular by exploiting the deep cache hierarchy of modern processors. Indeed, even for a standard dense matrix multiplication, optimal data reuse is essential to reach good performance. Usually, detailed knowledge of the hardware architecture can be combined with the well known data-access pattern of a dense matrix multiplication to optimally block matrices and to guarantee best cache reuse. However, due to the unknown sparsity pattern and relatively complicated data structures, this approach is not general enough in the sparse case. An alternative technique, also derived in the context of dense matrix multiplication is therefore employed, which instead uses a recursive approach to matrix multiplication [16]. Matrices are multiplied by recursively dividing the longest dimension of the matrix in two, until sufficiently small matrix dimensions have been obtained and all the data fits fully into a low-level cache. This *cache-oblivious algorithm* results in a near-optimal data access pattern for dense matrices, without explicit knowledge of the cache hierarchy, and is easily adapted to the sparse case.

8.2.3 CSR Layer

The compressed sparse row layer, or CSR layer, determines from the CSR data which blocks have to be multiplied. It is important to emphasize that the sparsity pattern of the result matrix is not fixed or known a priori, so that this processes is driven by the right-hand side of the equation ($C = AB$). In DBCSR, a two step approach, well suitable for GPUs, has been adopted. It separates performing the actual floating point operations from the indexing and book-keeping. The CSR layer performs the latter, on the host, deferring flops to lower layers. During the indexing, lists of needed block-multiplications, named "stacks" are generated, and passed on to the lower scheduling layers, that is, are flushed, as soon as the limited space of a stack is exhausted, or the end of a Cannon tick is reached. In order to allow for efficient processing, so-called homogeneous stacks are employed for the most common block sizes, these contain entries that have all the same block-dimensions, while a default stack contains the remaining cases. An important optimization has been introduced in this layer namely on-the-fly filtering. This optimization employs precomputed matrix block norms to decide if a given block product contributes to the final result significantly in comparison to the sparsity threshold, and skips negligible multiplications. In actual applications, even for matrices that are dense in data, this optimization can reduce the number of needed flops by a factor 2–4. The relative computational cost of the indexing operations depends strongly on the size of the basic blocks employed in the application calling the DBCSR library. It is significant if blocks are as small as 5×5, while it is clearly negligible if blocks are of size 23×23 or larger.

8.2.4 Scheduler and Driver Layers

The scheduler layer receives filled stacks and arranges for their processing by handing them off to one of the drivers. The host-driver is employed for CPU-processing and the CUDA-driver for GPU-processing. Both the host and device drivers are built on top of libraries that efficiently perform small matrix multiplications, libsmm and libcusmm for host and device, respectively. libsmm has been described in Ref. [14], while libcusmm is described in detail in a following section. Both libraries are significantly more efficient than standard matrix multiplication libraries for the small matrix sizes that are relevant for quantum chemical applications. The scheduler decides where the stack will be processed, and is currently based on a very simple scheme. The GPU is queried using the event based mechanism described below, and if buffer space is available on the GPU the stack will be handed over to the CUDA-driver, otherwise the host-driver processes the stack. The amount of buffer space

made available on the device is thus a mechanism to tune the host-device load balancing. Following this, the CUDA driver will check if highly tuned kernels are available in the libcusmm library for the particular matrix sizes in the homogeneous stack. If so, the stack is shipped to the GPU for processing, and otherwise is send to the host driver. The latter can deal with small matrix multiplications of all sizes, ultimately falling back to an optimized BLAS library calling DGEMM.

8.3 Maximizing Asynchronous Progress

Part of the challenge in writing efficient GPU-accelerated code is to exploit the asynchronous task based programming model. Whereas on a homogeneous system typically all processors execute the same program on different parts of the data in a lock-step fashion, on a hybrid system the CPU and GPU complement each other, and are partially independent. In order to fully utilize such a system, different programs need to be executed on the CPU and GPU. Typically, the host-CPU drives the GPU-device by handing over tasks, and while the GPU is executing these tasks, the CPU can perform other tasks on its own. In the following subsections, our approach to enable this asynchronous processing is explained.

8.3.1 CUDA Streams and Events

Once the host has submitted a task to the device, the CPU loses control over it and the GPU has significant freedom to schedule the task execution. However, dependencies between tasks might be present. For example, a task processing some data might depend on the completion of a prior task that copies this data from the host to the device. These dependencies have to be made explicit by the programmer. As discussed in Chapter 2, the CUDA programming environment provides two powerful mechanisms to enable further concurrency and to enforce dependencies: *streams* and *events*. Streams are a simple mechanism to establish dependencies and to enable concurrency. Tasks submitted to a given stream are processed in the order in which they are submitted, while tasks from different streams can be processed in any order or concurrently. Using multiple streams is essential to overlap computation with host-to-device or device-to-host transfers, and to enable concurrent task execution. Events can be used to express more general dependencies. Just like a task, an event can be created and submitted to a stream, and is processed after the previous task submitted to the same stream is completed. Furthermore, tasks can be submitted that wait for the completion of events in other streams. These "waiting-tasks" will block a stream until the referred event has occurred, and by submitting waiting-tasks prior to an actual task on the same stream, multiple cross-stream dependencies can be enforced.

In Figure 8.4, the scheme that is employed in the DBCSR library is illustrated. Stack buffers are transferred and processed in a number of independent streams, so that the stack buffer transfers can overlap with computations in other streams, and that concurrent stack processing is possible. The GPU can only process stacks if the panel and stack data is present, so that for each kernel dependencies on the completed transfer of the A, B, and C panels, taking place in different streams, and completion of the stack buffer transfer, in the same stream, must be present. Retaining the A, B, and C panels on the device while stack buffer processing is in progress is enforced with additional events. Notice that explicit synchronization between host and device is rarely needed, the host can query events to make sure that, for example, stack buffers have been uploaded before they are overwritten with new data. The host only has to wait for the device when the previous panel is still in use, and at the very end when the final results are downloaded from the device.

The CUDA API allows for an unlimited number of streams, but these are mapped to a limited number of *hardware queues*. Both the number of hardware queues and the mapping scheme are

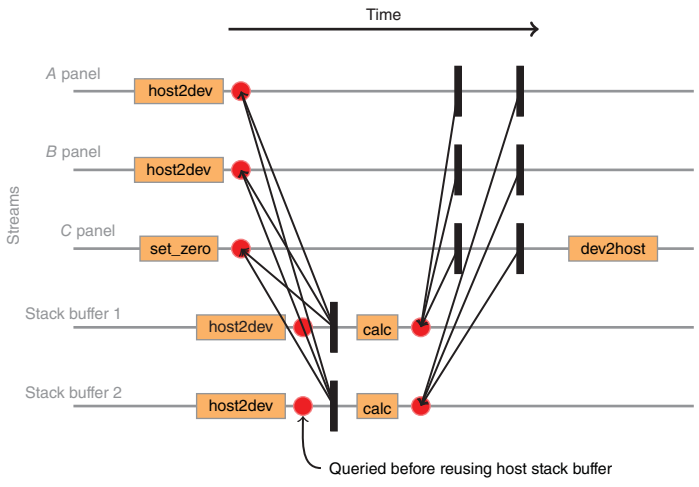


Figure 8.4 *Enabling concurrency and enforcing dependencies in DBCSR. Multiple streams are used to transfer data from the host to the device, and to process independent stacks. Dependencies between the streams, for example, a panel upload and stack calculations, and between host and device, for example, device buffer reuse, are enforced using events*

likely to change depending on the hardware and CUDA version. Unfortunately, mapping of otherwise independent streams to the same hardware queue can lead to unwanted serialization. Therefore, only a limited number of streams is created in DBCSR, specifically, two streams are exclusively for host-to-device transfers, one for odd Cannon ticks and one for even ticks, while a configurable but small (typically 2–4) number of streams is used for stack transfers and kernel launches. Finally, "priority streams" are a recent CUDA feature that introduces some way for the programmer to influence scheduling of kernels. In DBCSR this feature is used to load balance between host threads. In addition to generating stacks, occasionally a host thread will also process a stack. This happens when a host thread has no more free stack buffers available, that is, when the device is busy. In order to avoid that the device works on buffers of a thread that has finished its work already, and a busy thread looses time processing stacks, stack buffers come in two flavors: priority buffers and posterior buffers. A limited number of priority buffers is assigned to each thread, and mapped to a stream with high priority, while the posterior buffers are mapped to streams with lower priority. The effect of this is that the device will focus on doing the work for those threads that are actively generating stacks, that is, writing them to the priority buffers, while the posterior buffers are handled later. These buffers, as discussed below, are useful to overlap computation and communication during message passing or host to device transfers. Good performance requires that the number of priority buffers is tuned such that the device never idles if all threads are active and exclusively using priority buffers.

8.3.2 Double Buffered Cannon on Host and Device

In a sparse matrix multiplication algorithm, both data movement and floating point operations can contribute significantly to the total runtime. Maximum performance can only be achieved when the corresponding resources are utilized in parallel. To accomplish this, a double buffered scheme has been employed for both host and device. As shown in Figure 8.5, these two panel buffers are used in a complementary fashion, while one buffer is used for the computation, the other buffer is overwritten as part of a data transfer operation. Data transfer happens between MPI processes and between host

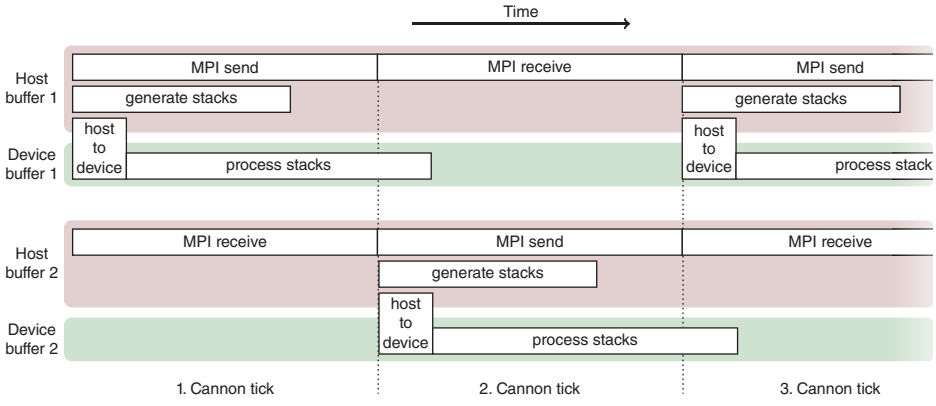


Figure 8.5 *Schematic representation of the double buffered Cannon algorithm, which illustrates how the use of two host and two device buffers for the A and B panels enables overlapping of message passing, host to device memory copies and computations. The ratio of the time needed for the important steps of the algorithm, depends on the hardware and on the science problem at hand*

and device, and thus double buffering is required for both operations. The host buffers are alternately used for MPI-send and MPI-receive. Once a panel has been received on the host it is copied to the corresponding device buffer, using the asynchronous *host-to-device* copy operation. At the same time, the CPU threads start to generate and fill stacks buffers. Stack buffers are transferred and processed by the device as soon as the host-to-device panel copy has finished. Typically, the CPU threads can generate the stacks faster than the GPU can process them, and a large number of stack buffer can be outstanding. These outstanding stack buffers can be processed by the device while the MPI transfer and the host to device copy of the next panel to the second buffer is taking place. Good performance requires that the number of posterior buffers is tuned such that device never idles during these transfers. A too large number of posterior buffers might lead to host threads waiting for the previous device panel buffer to finished. In the last Cannon tick, posterior buffers are not employed, as threads and device should finish roughly at the same time.

Fast device-to-host transfers require host-pinned memory. Since allocating host-pinned memory and CUDA device memory are slow operations, and memory usage is hard to predict in the case of varying sparsity patterns, memory-pools have been introduced that are persistent across sparse multiplications and only allowed to grow. In our application, the gain in performance outweighs the additional complexity and the fact that less memory is available for the rest of the application in between matrix multiplications.

Finally, whereas the MPI standard specifies non-blocking versions of send and receive (`isend`/`irecv`), actual implementations often perform the complete transfer in the corresponding `wait` statements. We have found that this is in particular the case for multi-megabyte messages, as is required for the panel transfers. To nevertheless overlap computation and communication, a "communication thread" has been introduced in the OpenMP parallel version of the DBCSR library. The master thread, which is responsible for all communication, is underloaded compared to the other threads, and will, given the barrier free nature of the implementation, enter early in a polling loop based on `test_any` to progress outstanding MPI communication. Tuning the load of the master thread, message passing can be effectively overlapped with computation performed by the other threads and the device.

8.4 Libcusmm: GPU Accelerated Small Matrix Multiplications

The core computational kernel in DBCSR is the computation of stacks of small matrix multiplications. The result block matrix $C^{u,v}$ is computed as the product of the block matrices $A^{u,w}$ and $B^{w,v}$ according to

$$C^{u,v}_{i,j} = C^{u,v}_{i,j} + \sum_{w,k} A^{u,w}_{i,k} B^{w,v}_{k,j}, \tag{8.11}$$

using superscripts to indicate the matrix block indices and subscripts to denote the matrix elements in each of the block matrices. The sum over w takes the sparsity pattern of A and B into account, that is, the product will be omitted whenever either $A^{u,w}$ or $B^{w,v}$ is absent, or their norms are small. Furthermore, w can only refer to those parts of A and B that are part of the panels of A and B that are local to the node for a given tick of Cannon's metronome. In a single stack, anywhere between one and a few tens of products will be present for a given block $C^{u,v}$. Note that this operation resembles the batched DGEMM operation in CUBLAS, but that this library expects all C matrices in a single batch to be different, and can thus not be used. In the following, the steps necessary to optimize these products on GPUs are described.

8.4.1 Small Matrix Multiplication Performance Model

At first sight, matrix multiplication seems dominated by floating point operations, while memory transfer is less important. This certainly is the case for large matrices, but not quite for the small matrices required in the current context. It is therefore useful to look at the arithmetic intensity, which we define to be the ratio of number of floating point operations versus number of bytes transferred between memory and processing units. In order to perform the matrix multiplication, A and B will need to be loaded from the device memory to the streaming multiprocessor (SM), while C might be assumed present on the SM (favorable limit of a large number of contributions from the summation over w), or might need to be loaded and stored as well. For the multiplication of an $m \times k$ by a $k \times n$ matrices, the intensity is thus between $\frac{2mnk}{8(mk+kn)}$ and $\frac{2mnk}{8(mk+kn+2mn)}$. In order to reach the favorable limit, the DBCSR library might sort the stacks, such that C matrix access occurs in order, prior to handing them to the GPU. Given a K20X GPU with 1.3TFlops peak double precision performance and 250 GB/s peak bandwidth, an arithmetic intensity of at least 5.2 is needed to achieve peak performance. With ECC turned on, a bandwidth of 180 GB/s is more realistically achievable for a kernel of this complexity, so an arithmetic intensity of at least 7.2 is needed to reach peak performance. Multiplications of matrices smaller than 60×60 are thus necessarily limited by the memory bandwidth, and this remains an important factor, even for significantly larger matrices. This clearly implies that the optimization should focus on reaching optimal memory bandwidth usage. For selected sizes of the small matrices encountered in CP2K applications, the arithmetic intensity, the reachable flop rate, and the actually achieved performance are shown in Figure 8.6.

8.4.2 Matrix-Product Algorithm Choice

The first step in implementing the small matrix products is to pick the most appropriate algorithm. Figure 8.7 shows two possible algorithms for computing the matrix product ($C = C + AB$). In the canonical form, the result elements in C are computed using the inner product of rows of A and columns of B, while an alternative algorithm is based on an outer product of columns of A and rows of B. These two algorithms result in the same number of floating point operations, but the latter option

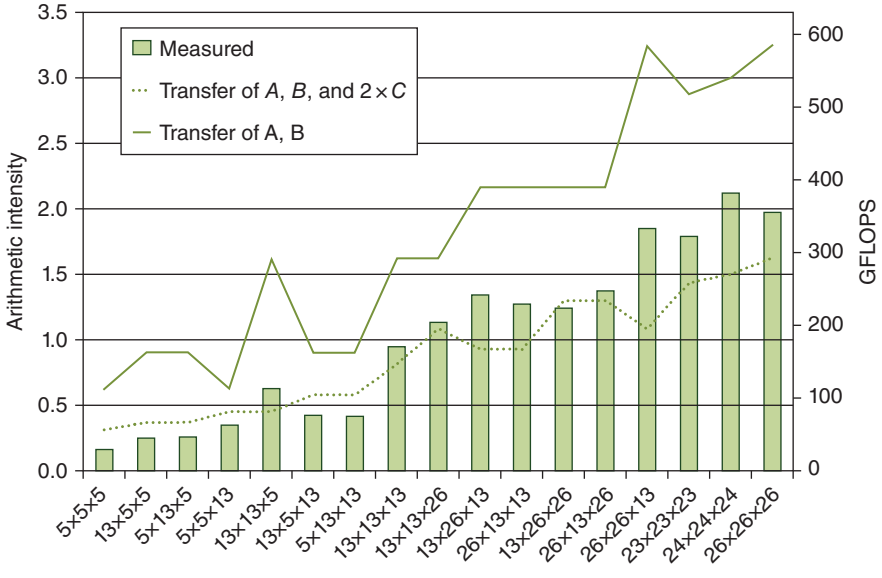


Figure 8.6 *Minimum (dotted line) and maximum (solid line) arithmetic intensity for different matrix sizes commonly employed in CP2K simulations, and the corresponding maximum possible flop rate. The performance as obtained from individual kernel launches in a mini-app is shown as bars*

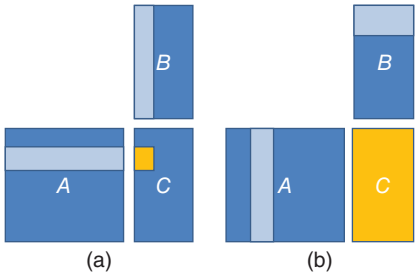


Figure 8.7 *Inner-product (a) and outer-product (b) form of matrix multiplication. The yellow areas in C indicate elements that can be computed independently by accessing the highlighted areas of A and B. (See insert for colour representation of this figure)*

exhibits significantly more parallelism in that it allows for computing an update for all elements of C using a single column of A and a single row of B. An additional benefit of using outer products is data locality, the outer product algorithm touches elements of A and B only once, while for the inner products, when computing one row of C, one row of A and the entire matrix B needs to be accessed. Based on the model developed in the previous section it is known that the kernel's performance for problem sizes of interest to CP2K will be limited by memory bandwidth. The outer products algorithm is therefore preferred.

8.4.3 GPU Implementation: Generic Algorithm

The next step in the design of a kernel for small matrix products is to consider data locality. Initially, the A, B, and C matrices, as well as the product descriptors, the so-called stacks, are all located in global memory on the GPU. Each entry in the stack describes one matrix–matrix product, thus

containing three pointers to the blocks in the *A*, *B*, and *C* panels. After the kernel has read a stack entry, it fetches the matrices *A* and *B*, and updates the *C* matrix with the product of *A* and *B*. The matrix sizes of interest correspond to typically 10–1000 elements per result matrix *C*, limiting the degree of parallelism to a similar order. An appropriate choice is therefore to process a matrix product using a single thread block. While this allows for efficient synchronization between the threads processing one product, it requires appropriate safe-guards to avoid data races between multiple updates of the same *C* matrix block. Multiple consecutive products updating the same result matrix *C* can be processed by the same thread block, requiring fewer reads and writes of *C* from global memory. In addition to reducing the number transfers of *C* between global memory and the SM, this also reduces the probability of collisions that happen when multiple thread blocks update the same *C* matrix block at the same time. On Kepler–Generation GPUs, atomic memory operations are efficient enough and are hence used to prevent data races. In this context, the overhead of using atomics instead of regular memory updates is on the order of 5%.

How a single thread block deals with the data is illustrated in Figure 8.8 and explained in the following. First, given that the elements of the result matrix *C* do not need to be shared between threads, the ideal location to store *C* is registers. In order to increase instruction level parallelism per thread, and given the large number of registers available per thread, a small tile (*T*) rather than a single result matrix element is processed per thread. The optimal choice of tile dimensions ($M \times N$) is determined via auto-tuning as described later. Next, the elements of matrix *A* and *B* need to be accessed by multiple threads, thus making them ideal candidates to be stored in shared memory. In order to avoid that shared memory utilization limits the number of concurrent thread-blocks (occupancy), a maximum of 3 kB per thread-block can be used. For larger matrix blocks *A* and *B*, it is thus desirable to read only parts into shared memory, we name these parts slabs. Using the outer product algorithm, these matrix slabs only need to be read once per product. The optimal width (w) of the slabs is also determined by auto-tuning. Given that matrices are stored in column major format, reading a slab of *A* still leads to perfectly coalesced memory access. However, reading only a slab of matrix *B* can lead to both significant memory access penalties and complex address computations. In order to avoid these

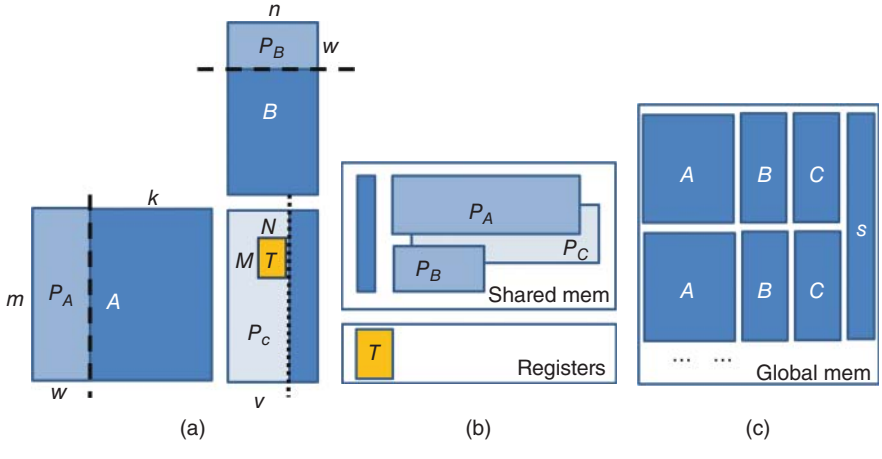


Figure 8.8 *(a) Parameterization of the $m \times n \times k$-matrix product $C = C + AB$. Each thread computes an $M \times N$ tile (T) of the result matrix C. In order to accommodate matrix sizes larger than the available shared memory, matrices are processed in slabs (P_A, P_B), with an input slab width w. In order to optimize the data output, the matrices (P_C) are written back using the output slab width v. (b) Close to the SM, registers are used to store the C matrix tile, while slabs of A, B, and C are stored in shared memory. (c) GPU memory stores all panel data, including the various blocks of A, B, C, and the stack buffers S. (See insert for colour representation of this figure)*

penalties it is therefore desirable to compute the product $C = A(B^T)^T$ instead, resulting in perfectly coalesced memory accesses with simple address computations for both A and B. Given that typically each B matrix is used many times per Cannon tick, the cost of transposing the full panel of B's once after the upload in a separate kernel is negligible compared to the time savings due to more efficient memory access and simplified address calculation. Finally, once the entire product is computed, and only when the next stack entry refers to a different C block, the results are added to the corresponding block in global memory. In order to ensure coalesced writes, an intermediate step is employed, in which slabs of C (of width v) are first put in shared memory, and only then added using an atomic compare-and-swap operation.

8.4.4 Auto-Tuning and Performance

The generic algorithm outlined above requires several parameters (M, N, w, and v), and finding an optimal set of values is not always intuitive. Fetching larger panels of A and B tends to improve performance, but at the same time will also increase the shared memory footprint and limit occupancy, thus potentially limiting the amount of latency hiding. A similar effect occurs for the number of result elements processed per thread: increasing this parameter improves the instruction level parallelism, but at the same time this limits the number of thread blocks resident on each SM. Additionally, some matrix sizes allow for significantly simplified versions of the general kernel and separate implementations were developed. In order to hide details of the tool chain, such as register allocation, that are unknown or subject to change, an autotuning framework based on a small standalone benchmark application is used to find optimal parameters and implementations for each given set of block dimensions m, n, k. It has been verified that the kernel performance in the small standalone benchmark application is very similar to the one observed in full CP2K simulations.

Figure 8.6 shows the performance obtained in the mini-app for relevant block sizes and optimal parameters. The performance is close to that estimated from the model based solely on memory bandwidth considerations. For very small matrices, the measured performance starts to deviate from the theoretically expected performance. We currently attribute this to the warp granularity of the execution on the SM, but have not further optimized for these sizes as we expect that small matrix sizes can just be handled on the CPU side if needed. Finally, for comparison, batched DGEMM in CUBLAS (version 5.0) for a 23×23×23 problem runs at 132 GFLOPS on an Nvidia K20X, while the current implementation in libcusmm achieves about 322 GFLOPS. For most of the small matrix sizes of interest, a speedup in the range of 2–4× has been measured. This demonstrates the quality of the generated kernels, and the appropriate choice of optimization techniques.

8.5 Benchmarks and Conclusions

In this final section, we illustrate the performance of the linear scaling GPU based implementation. In doing so, we attempt to cover synthetic benchmarks, current application style simulations, as well as very large scale simulations. Given the computational demands of these simulations, the focus is on parallel application of CP2K. The latter calculations have been performed on a recent hybrid architecture, a Cray XC30, which was installed in the fall of 2013 at CSCS, Switzerland. This machine is named Piz Daint, and is currently the leading European computer in the Top500 list. It features 5272 hybrid compute nodes, with one Intel Xeon E5-2670 processor (8 core, Sandy Bridge), and one Nvidia K20X per node. The nodes are connected with an Aries network based on a dragonfly topology.

As a first demonstration of performance, we focus on two synthetic benchmarks of the matrix–matrix multiplication of nearly dense matrices (occupied 50% or more), with favorable block sizes (23×23). A single node CPU–GPU comparison is shown in Figure 8.9. In this case, the

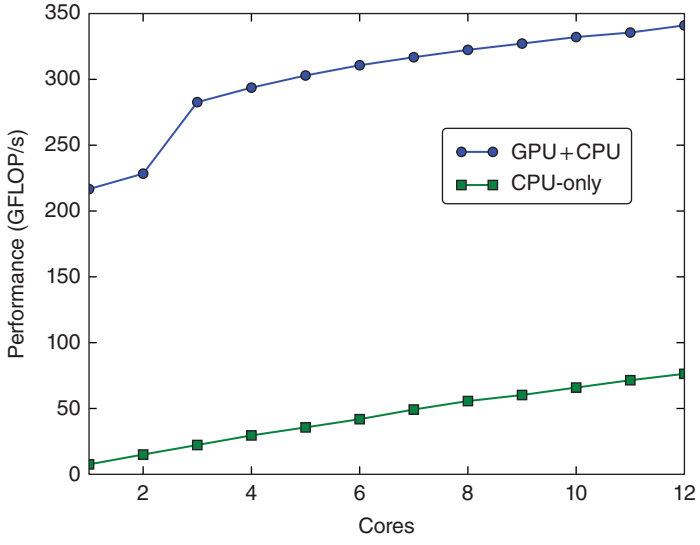


Figure 8.9 *Performance comparison of the multi-threaded DBCSR library based on* 23×23 *matrix blocks, and was not using the MPI capabilities. The benchmark was run on a dual Sandy Bridge (E5-2620, 2.0 GHz, 6 cores) machine, equipped with one Nvidia Tesla K20 card*

potential of the GPU is demonstrated, as it outperforms 12 Sandy Bridge cores by a significant factor. Furthermore, increasing the number of CPU cores, the hybrid implementation displays improving performance, showing that host-device sharing is effective. The CPU-only curve demonstrates good parallel efficiency of the OpenMP code. Taking this benchmark to the scale of 5184 hybrid nodes, a matrix of size $536,544 \times 536,544$, with 50% occupation, and 23×23 subblocks, can be multiplied in approximately 36 seconds with a sustained machine performance in excess of 2 PFLOPS (nearly 400 GFLOPS per node). Thus, exploiting the fact that the matrices are 50% occupied, already brings a speedup over a dense matrix multiplication. Indeed, assuming a dense parallel matrix multiplication to run at 6.2 PFLOPS (the Linpack number for Piz Daint), such a calculation would require 50 seconds. This performance illustrates the quality of the parallel implementation of the sparse matrix code.

More important is application level performance for realistic simulation setups. In order to assess this, we employ three benchmarks that are also part of the CP2K distribution, named amorph, H_2O, and TiO_2. These describe an amorphous organic hole conducting material, bulk liquid water, and titanium dioxide nanoparticles, respectively. Geometries are realistic, disordered, three-dimensional, and with periodic boundary conditions. Basis sets are of double zeta quality (DZVP-MOLOPT-SR-GTH) and include diffuse primitives, contraction based on molecular optimization makes them at the same time accurate and suitable for linear scaling calculations in the condensed phase [17]. Since these benchmarks are designed to run quickly on a relatively small number of compute nodes they have reduced SCF counts. Key quantities and results are provided in Table 8.1. First, for the given basis sets and thresholds, it shows that systems of approximately 10,000 atoms can be computed in minutes using 169 nodes (only 4% of the national supercomputer). This paves the way for scientific applications based on models of this size, including geometry optimization and molecular dynamics based relaxation. Second, this comparison at 169 nodes shows a speedup of 1.4–1.7 going from a traditional homogeneous node to a hybrid node. In these cases, the GPU is processing most of the FLOPS. More detailed analysis shows that for the amorph benchmark the small block sizes limit the speedup, while the H_2O testcase is already limited by MPI communication.

Table 8.1 *Key quantities of three linear scaling benchmarks that are distributed with CP2K*

	Amorph	H_2O	TiO_2
Number of atoms	13,846	20,736	9786
Number of basis functions	133,214	158,976	169,624
Block sizes	5, 13	23	13, 26
Number of SCF steps in benchmark	2	2	1
Filtering threshold	10^{-6}	10^{-6}	10^{-5}
Typical matrix occupation (%)	16	11	12
Run time on 169 × 2 SB (seconds)	372	275	446
Run time on 169 × 1 SB + 1 K20X (seconds)	272	187	263
Performance ratio on 169 nodes	1.4	1.5	1.7
GPU FLOP (%)	92	99	88

The run time is provided for two setups, one in which 2 Sandy Bridge (SB) CPUs are present per node, and a hybrid architecture in which 1 SB and 1 K20X GPU is present per node. Performance ratio compares the run time between these setups, GPU FLOP (%) gives the percentage of FLOPS that is executed on the GPU in the hybrid setup.

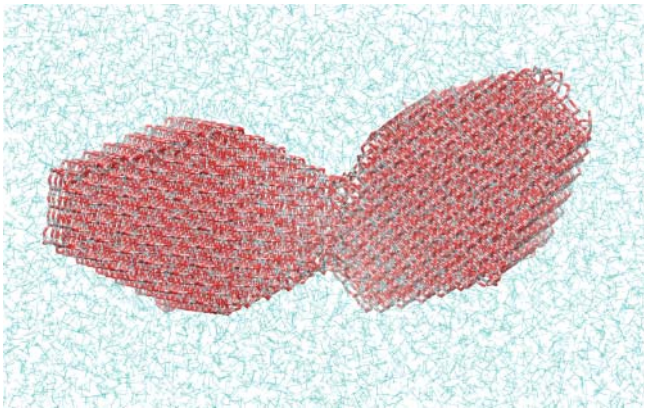


Figure 8.10 *Aggregated nanoparticles in explicit solution (77,538 atoms) can be run on the Piz Daint computer (5272 hybrid compute nodes) at approximately 122 seconds per SCF step*

The largest system computed so far on the hybrid system Piz Daint is shown in Figure 8.10. It consists of aggregated nanoparticles of TiO_2 in an explicit acetonitrile solvent, as found in dye sensitized solar cells, and consists of 77,538 atoms and 772,868 basis functions. For a filtering threshold of 10^{-6}, a matrix occupation of 4% is found. Running on 5184 nodes, a single SCF step takes approximately 122 seconds. Performance is roughly 30 GFLOPS per node, as the calculation is strongly dominated by MPI communication. The GPUs perform 99.4% of the FLOPS.

To conclude, we have shown that linear scaling SCF calculations with good quality on large three dimensional systems have become possible with good time to solution. As such, linear scaling approaches on large models have become one of the many tools that atomistic simulation offers to investigate an ever increasing range of physical systems. The progress can be attributed to an evolution and interplay between hardware, algorithms, and implementations. The GPU work presented here is a prime example. To harvest the raw power of GPUs, suitable algorithms had to be adopted and a very careful implementation was needed. In particular, the asynchronous nature of the device had to be taken into account at a sufficiently high level, and a library of highly optimized kernels had to be created. This required a detailed understanding of the GPU device and the application

programming interface. Finally, good performance has been demonstrated on one of the largest GPU based supercomputers worldwide. To further benefit from the GPU compute power, new message passing algorithms are being developed.

Acknowledgments

The authors acknowledge Urban Borštnik, Florian Schiffmann, Florian Thöle, Jinwoong Cha, Valéry Weber, Christiane Pousa Ribeiro, Iain Bethune (EPCC), Chris Mundy (PNNL), Nikolay Markovskiy (NVIDIA), Neil Stringfellow (CSCS), Gilles Fourestey (CSCS), Alfio Lazzaro (Cray), and Roberto Ansaloni (Cray) for their help with the implementation, discussions, and applications. J.V. acknowledges financial support by the European Union FP7 in the form of an ERC Starting Grant under contract no. 277910. Calculations were enabled by a grant of the Swiss National Supercomputer Centre (CSCS) under project IDs s238, s441, and h05. In preparation of this research, resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725, have been used. The research leading to these results has received funding from the Swiss University Conference through the High Performance and High Productivity Computing (HP2C) Programme.

References

1. Goedecker, S. (1999) Linear scaling electronic structure methods. *Rev. Mod. Phys.*, **71** (4), 1085–1123.
2. Bowler, D.R. and Miyazaki, T. (2012) O(N) methods in electronic structure calculations. *Rep. Prog. Phys.*, **75** (3), 036503.
3. The CP2K developers group. 2013. CP2K is freely available from: `http://www.cp2k.org/` (accessed 18 September 2015).
4. VandeVondele, J., Borstnik, U. and Hutter, J. (2012) Linear scaling self-consistent field calculations with millions of atoms in the condensed phase. *J. Chem. Theory Comput.*, **8** (10), 3565–3573.
5. Higham, N.J. (2008) *Functions of Matrices: Theory and Computation*, Society for Industrial and Applied Mathematics, Philadelphia, PA.
6. Beylkin, G., Coult, N. and Mohlenkamp, M. (1999) Fast spectral projection algorithms for density-matrix computations. *J. Comput. Phys.*, **152** (1), 32–54.
7. Nemeth, K. and Scuseria, G. (2000) Linear scaling density matrix search based on sign matrices. *J. Chem. Phys.*, **113** (15), 6035–6041.
8. Palser, A. and Manolopoulos, D. (1998) Canonical purification of the density matrix in electronic-structure theory. *Phys. Rev. B*, **58** (19), 12704–12711.
9. Niklasson, A.M.N., Tymczak, C.J. and Challacombe, M. (2003) Trace resetting density matrix purification in O(N) self-consistent-field theory. *J. Chem. Phys.*, **118** (19), 8611–8620.
10. Li, X., Nunes, R. and Vanderbilt, D. (1993) Density-matrix electronic-structure method with linear system-size scaling. *Phys. Rev. B*, **47** (16), 10891–10894.
11. Helgaker, T., Larsen, H., Olsen, J. and Jorgensen, P. (2000) Direct optimization of the AO density matrix in Hartree-Fock and Kohn-Sham theories. *Chem. Phys. Lett.*, **327** (5–6), 397–403.
12. Shao, Y., Saravanan, C., Head-Gordon, M. and White, C. (2003) Curvy steps for density matrix-based energy minimization: application to large-scale self-consistent-field calculations. *J. Chem. Phys.*, **118** (14), 6144–6151.
13. Salek, P., Host, S., Thogersen, L., Jorgensen, P., Manninen, P., Olsen, J. *et al.* (2007) Linear-scaling implementation of molecular electronic self-consistent field theory. *J. Chem. Phys.*, **126** (11), 114110.

14. Borstnik, U., VandeVondele, J., Weber, V. and Hutter, J. (2014) Sparse matrix multiplication: the distributed block-compressed sparse row library. *Parallel Comput.*, **40**, 47–58.
15. Cannon, L.E. (1969) *A Cellular Computer to Implement the Kalman Filter Algorithm*, Montana State University, Bozeman, MT.
16. Chatterjee, S., Lebeck, A.R., Patnala, P.K. and Thottethodi, M. (2002) Recursive array layouts and fast matrix multiplication. *IEEE Trans. Parallel Distrib. Syst.*, **13** (11), 1105–1123.
17. VandeVondele, J. and Hutter, J. (2007) Gaussian basis sets for accurate calculations on molecular systems in gas and condensed phases. *J. Chem. Phys.*, **127** (11), 114105.

9

Grid-Based Projector-Augmented Wave Method

Samuli Hakala[1], *Jussi Enkovaara*[1,2], *Ville Havu*[1], *Jun Yan*[3], *Lin Li*[3], *Chris O'Grady*[3] *and Risto M. Nieminen*[1]

[1]*Department of Applied Physics, Aalto University, Espoo, Finland*
[2]*CSC – IT Center for Science Ltd., Espoo, Finland*
[3]*SUNCAT Center for Interface Science and Catalysis, SLAC National Accelerator Laboratory, Menlo Park, CA, USA*

GPAW is a versatile open-source package for electronic structure simulations within density functional theory (DFT). GPAW implements the projector-augmented-wave (PAW) method utilizing uniform real-space grids, and we describe here how the resulting algorithms can be accelerated by using general-purpose graphics processing units (GPUs). In addition to standard DFT, the real-time form of time-dependent DFT (TD-DFT) is implemented to utilize GPUs. For DFT and TD-DFT calculations, we demonstrate speedups of up to 19 for simulations that use one GPU per regular CPU core. As a specific feature, we discuss also how the computationally demanding random phase approximation to the exchange-correlation energy can be accelerated by a factor of 40 using one GPU per CPU core.

9.1 Introduction

Density functional theory (DFT) [1, 2] is one of the most important theoretical tools for studying atomic-scale properties in various condensed-matter systems, and also one of the largest consumers of supercomputing resources. General-purpose graphics processing units (GPUs) are becoming a promising option over traditional CPUs in speeding up various computer simulations, including DFT-based methods. In this chapter, we discuss how GPUs are utilized in the open-source DFT software package GPAW [3, 4].

As was discussed in Chapter 3, several numerical approximations have to be made when solving the Kohn–Sham equations of DFT. The approximations can be related either to the treatment of

Electronic Structure Calculations on Graphics Processing Units:
From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

core electrons and the region close to atomic nuclei (all-electron vs. pseudopotential methods) or to the discretization of the equations. Each numerical approximation has its own advantages and disadvantages, and a large number of DFT software packages exist with varying numerical schemes. A nonexhaustive list is given in the references of this chapter [5–17]. Details in the context of GPU implementations can be found in this and other chapters of this book (Chapters 4–10 deal with DFT implementations). The GPAW package uses the projector-augmented wave (PAW) method [18] for treating the region near atomic nuclei; either uniform real-space grids, localized atomic orbitals or plane waves can be used for discretizing the equations. The GPU implementation uses the real-space grid mode, and as that is also the main working mode, we focus on real-space grid discretization in this chapter.

The underlying idea in the PAW method is similar to the pseudopotential approximation; that is, core electrons are considered frozen in their atomic configurations and valence electrons are represented with smooth functions (see also Chapter 7). However, the PAW method is formally an all-electron (AE) method retaining information about the whole nodal structure of the wave functions, and AE wave functions are available at any stage of the calculation. Compared to more conventional norm-conserving or ultrasoft pseudopotentials, the PAW method offers a more reliable description over the whole periodic table. Transferability problems with PAW potentials are normally less severe, and as the pseudo-wave functions in the PAW are often smoother, fewer degrees of freedom are needed for their representation.

Uniform real-space grids offer a relatively simple discretization for the Kohn–Sham equations within the PAW formalism. Accuracy of the discretization can be improved systematically by decreasing the spacing between the grid points, and with the typical grid spacings needed in the context of the PAW method, the discretization is efficient. With real-space grids it is possible to flexibly treat both periodic (bulk) and isolated (atoms, molecules, clusters) systems, as well as surfaces and wires where some of the dimensions are periodic and some finite. Real-space grids enable the use of efficient multigrid techniques, and they offer also good parallelization prospects, as information is typically needed only from a few neighboring grid points. These local aspects of parallelization are especially important when using multiple GPUs.

Standard DFT provides access only to the ground-state properties of the system. However, several important physical quantities such as excitation energies and optical spectra are related to the excited states of the system, which can be studied with time-dependent DFT (TD-DFT) [19]. GPAW has several different implementations of TD-DFT, of which the real-time integration of time-dependent Kohn–Sham equations can utilize GPUs.

The physical approximations in DFT are contained in the exchange-correlation (XC) functional. The simplest XC approximations are the local density approximation (LDA) and the various generalized-gradient approximations (GGAs). Despite their success in many physical systems, biomolecules on metallic surfaces, as an example, are described poorly by LDA or GGA. An improved description for these systems is given by the random phase approximation (RPA) to the correlation part of the XC energy. RPA is, however, computationally time consuming, and GPUs offer significant speedup for RPA calculations.

This chapter is organized as follows. First we give a general overview of the PAW method, as well as of the discretization with uniform real-space grids in Section 9.2. We also discuss one specific feature of the real-space algorithm, the multigrid method. The GPU implementation for the ground-state calculations and associated results are presented in Section 9.3. Section 9.4 discusses real-time propagation TD-DFT and its GPU implementation, and RPA together with the GPU implementation is presented in Section 9.5. Finally, we give a summary and outlook in Section 9.6.

9.2 General Overview

In this section we provide a short overview of the PAW method and discuss its implementation with uniform real-space grids. We also present an overview of one of the key algorithms in our real-space implementation, the multigrid method, which is used in the solution of the Poisson equation and in the preconditioning of the Kohn–Sham eigenproblem. More detailed descriptions of the underlying formalism and its implementation can be found in the original references of the PAW method [18] and the GPAW implementation [3, 4]. Atomic units ($\hbar = m = e = \frac{1}{4\pi\epsilon_0} = 1$) are used throughout the chapter.

9.2.1 Projector-Augmented Wave Method

The Kohn–Sham equations describe the core and valence electrons on equal footing. However, chemical environments affect mostly the valence electrons, and the core electrons remain in their atomic configuration. Thus, for many physical properties it is sufficient to solve the equations explicitly only for valence electrons. Because of the strong Coulomb interaction, the single-particle valence wave functions vary strongly near the atomic nuclei. The PAW method provides a rigorous formalism for working with smooth pseudo-valence wave functions, so that the resulting equations can be discretized more easily, for example, with plane waves (see Chapter 7) or uniform real-space grids (see also Chapter 10).

At the heart of the PAW formalism is a linear transformation between the AE wave function ψ_n and the smooth valence pseudo (PS) wave function $\tilde{\psi}_n$:

$$\psi_n(\mathbf{r}) = \hat{\mathcal{T}}\tilde{\psi}_n(\mathbf{r}), \tag{9.1}$$

where n is the electronic state index. The transformation operator $\hat{\mathcal{T}}$ is constructed in a manner such that the AE wave functions are orthogonal to the core wave functions $\phi_i^{a,\,\mathrm{core}}$, which are fixed to their reference shape in the isolated atom.

The transformation operator $\hat{\mathcal{T}}$ is defined in terms of atom-centered AE partial waves $\phi_i^a(\mathbf{r})$, the corresponding smooth partial waves $\tilde{\phi}_i^a(\mathbf{r})$, and projector functions $\tilde{p}_i^a(\mathbf{r})$ as

$$\hat{\mathcal{T}} = 1 + \sum_a \sum_i (|\phi_i^a\rangle - |\tilde{\phi}_i^a\rangle)\langle\tilde{p}_i^a|, \tag{9.2}$$

where atom a is at the position $\mathbf{R}^a$. The atom-centered AE partial waves and smooth PS partial waves are equal outside the atom-centered augmentation spheres of radii r_c^a:

$$\phi_i^a(\mathbf{r}) = \tilde{\phi}_i^a(\mathbf{r}), |\mathbf{r} - \mathbf{R}^a| > r_c^a. \tag{9.3}$$

The projector functions are localized inside the augmentation spheres and are orthogonal to the PS partial waves:

$$\langle\tilde{p}_{i_1}^a|\tilde{\phi}_{i_2}^a\rangle = \delta_{i_1 i_2}. \tag{9.4}$$

The projectors and partial waves are constructed from an AE calculation for a spherically symmetric atom, and the index i in Eq. (9.2) refers to the principal and angular momentum quantum numbers in an isolated atom. Given a complete set of atom-centered partial waves and projectors, the PAW transformation is exact. In practical calculations, two functions per angular momentum is typically enough.

The pseudo-electron density can be defined as

$$\tilde{n}(\mathbf{r}) = \sum_n f_n |\tilde{\psi}_n(\mathbf{r})|^2 + \sum_a \tilde{n}_c^a(\mathbf{r}), \tag{9.5}$$

where f_n are the occupation numbers between 0 and 2, and $\tilde{n}_c^a$ is a smooth PS core density equal to the AE core density n_c^a outside the augmentation sphere. Using the atomic density matrix $D^a_{i_1 i_2}$

$$D^a_{i_1 i_2} = \sum_n \langle \tilde{\psi}_n | \tilde{p}^a_{i_1} \rangle f_n \langle \tilde{p}^a_{i_2} | \tilde{\psi}_n \rangle. \tag{9.6}$$

it is possible to define one-center expansions of the AE and PS densities:

$$n^a(\mathbf{r}) = \sum_{i_1, i_2} D^a_{i_1 i_2} \phi^a_{i_1}(\mathbf{r}) \phi^a_{i_2}(\mathbf{r}) + n_c^a(\mathbf{r}) \tag{9.7}$$

and

$$\tilde{n}^a(\mathbf{r}) = \sum_{i_1, i_2} D^a_{i_1 i_2} \tilde{\phi}^a_{i_1}(\mathbf{r}) \tilde{\phi}^a_{i_2}(\mathbf{r}) + \tilde{n}_c^a(\mathbf{r}), \tag{9.8}$$

respectively. Finally, from $\tilde{n}$, n^a, and $\tilde{n}^a$, the AE density can be constructed in terms of a smooth part and atom-centered corrections:

$$n(\mathbf{r}) = \tilde{n}(\mathbf{r}) + \sum_a (n^a(\mathbf{r}) - \tilde{n}^a(\mathbf{r})). \tag{9.9}$$

Similar to the electron density, in the PAW formalism the expectation value of any local (or semilocal) operator can be obtained in terms of a smooth part and atom-centered corrections:

$$\langle \hat{O} \rangle = \tilde{O} + \sum_a \Delta O^a. \tag{9.10}$$

The Hamiltonian operator in the PAW formalism has the form

$$\hat{H} = -\frac{1}{2}\nabla^2 + \tilde{v} + \sum_a \sum_{i_1 i_2} |\tilde{p}^a_{i_1}\rangle \Delta H^a_{i_1 i_2} \langle \tilde{p}^a_{i_2}|, \tag{9.11}$$

where $\Delta H^a_{i_1 i_2}$ are the atom-centered PAW corrections to the Hamiltonian. The smooth effective potential has the form

$$\tilde{v} = \tilde{v}_{\text{coul}} + \tilde{v}_{\text{xc}} + \sum_a \bar{v}^a, \tag{9.12}$$

where the PAW-specific terms $\bar{v}^a$ compensate for the incompleteness of the partial waves and projectors. The Coulomb potential satisfies the Poisson equation

$$\nabla^2 \tilde{v}_{\text{coul}} = -4\pi\tilde{\rho}, \tag{9.13}$$

where the pseudo-charge density $\tilde{\rho}$ also contains the so-called compensation charges for electrostatic decoupling of different augmentation spheres [3, 20]. The XC potential $\tilde{v}_{\text{xc}}$ is the normal semilocal function evaluated with the pseudo-electron density $\tilde{n}$.

The PS wave functions are not orthonormal as such, but with respect to the overlap operator $\hat{S}$:

$$\langle \tilde{\psi}_n | \hat{S} | \tilde{\psi}_m \rangle = \delta_{nm}, \tag{9.14}$$

where

$$\hat{S} = \hat{\mathcal{T}}^{\dagger}\hat{\mathcal{T}} = 1 + \sum_{a}\sum_{i_1 i_2} |\tilde{p}^a_{i_1}\rangle \Delta S^a_{i_1 i_2} \langle \tilde{p}^a_{i_2}|, \tag{9.15}$$

$$\Delta S^a_{i_1 i_2} = \langle \phi^a_{i_1} | \phi^a_{i_2}\rangle - \langle \tilde{\phi}^a_{i_1} | \tilde{\phi}^a_{i_2}\rangle. \tag{9.16}$$

As a result, in the PAW formalism, the Kohn–Sham eigenvalues ϵ_n and the PS wave functions are obtained from a generalized eigenproblem

$$\hat{H}\tilde{\psi}_n = \epsilon_n \hat{S}\tilde{\psi}_n. \tag{9.17}$$

9.2.2 Uniform Real-Space Grids

The Poisson equation and the Kohn–Sham equations in the PAW formalism can be discretized conveniently with a uniform real-space grid. Physical quantities such as the wave functions, densities, and potentials are represented by their values at the grid points. The atom-centered partial waves and projector functions are represented on radial grids, and all the integrals involving only them are evaluated on this grid. For the operations that involve both atom-centered and extended functions, the localized functions are represented on the same uniform grid within the atomic spheres as the extended functions.

Both the Kohn–Sham Hamiltonian and the Poisson equation contain the Laplacian, which is evaluated with finite differences:

$$\nabla^2 f(\mathbf{r}) = \sum_{\alpha=1}^{3}\sum_{n=-N}^{N} b_\alpha c_n^N f(\mathbf{r} + n\mathbf{h}_\alpha) + \mathcal{O}(h^{2N}). \tag{9.18}$$

The grid spacing vectors are defined as $\mathbf{h}_\alpha = \mathbf{a}_\alpha / N_\alpha$; $b_\alpha = 1/h_\alpha^2$ and c_n^N are the Nth order finite difference coefficients for the second derivative expansion. The accuracy of the discretization is defined by the grid spacings h_α (typically a single value is used for all three Cartesian directions) and the order of the finite-difference stencil.

The three indices designating a point in the real-space grid $G_\alpha = 1, N_\alpha$ can be condensed into a single G index, so that, for example, the discretized PS wave functions are $\tilde{\psi}_n(\mathbf{r}) \approx \tilde{\psi}_{nG}$, and the Hamiltonian operator is

$$H_{GG'} = -\frac{1}{2}L_{GG'} + v_{\mathrm{eff,G}}\delta_{GG'} + \sum_{i_1 i_2} p^a_{i_1 G}\Delta H^a_{i_1 i_2} p^a_{i_2 G'}, \tag{9.19}$$

where $L_{GG'}$ is the finite-difference stencil for the Laplacian. The resulting matrix is sparse, and it is never stored explicitly. Instead, one evaluates the operation of Hamiltonian into wave functions, that is, matrix free matrix–vector products.

9.2.3 Multigrid Method

The multigrid method [21, 22] is a general method for solving partial differential equations by using a hierarchy of discretizations. The underlying idea is that many simple relaxation methods show different convergence rates for the long and short wavelength components of the error. Short wavelengths can be treated efficiently with fine discretization, while long wavelengths can be treated with coarser discretization; and by moving between discretization levels it is possible to converge all the components of the error efficiently.

Generally, multigrid algorithms contain three parts: a smoothing operation for removing error components relevant for the discretization level; a restriction operation for moving from fine discretization to a coarser one; and an interpolation operation for moving from coarse discretization to a finer one.

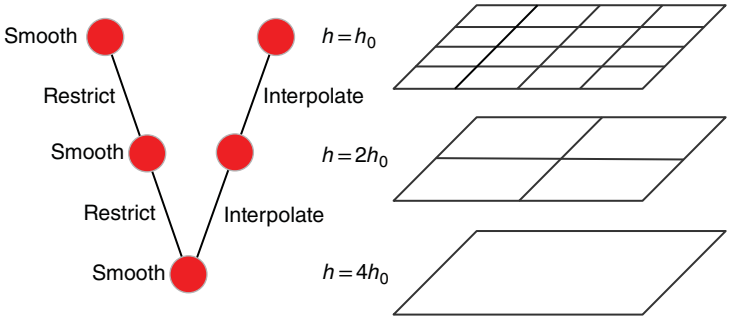


Figure 9.1 *Multigrid V-cycle with three levels*

As an example, the solution of the Poisson equation with the multigrid method proceeds as follows:

1. Perform a few iterations of the Jacobi method on the original real-space grid (smoothing).
2. Move to a 2× coarser grid (restrict).
3. Perform a few iterations of the Jacobi method on the coarser grid (smoothing).
4. Repeat Steps 2 and 3 until lowest resolution level is reached. In many multigrid algorithms the coarsest level is solved exactly, but in our case an approximate solution is sufficient.
5. Move back to the 2× finer grid (interpolate).
6. Perform a few iterations of the Jacobi method on the finer grid (smoothing).
7. Repeat Steps 5 and 6 until the original resolution level is reached.

This so-called multigrid V-cycle is illustrated in Figure 9.1.

In addition to the Poisson equation, GPAW uses the multigrid method in the preconditioning of the iterative eigensolver.

9.3 Using GPUs in Ground-State Calculations

GPAW is implemented using a combination of the Python and C-programming languages. Generally, high-level algorithms are implemented with Python, while the numerically intensive operations are implemented in C or utilize libraries. The goal of the GPU-accelerated implementation was to keep all the high-level algorithms identical and only make changes to low-level code [23]. We use the PyCUDA [24] toolkit to enable the use of GPU in the Python code and several custom CUDA kernels [25] to speed up the GPAW C-extensions. Most of the basic dense linear algebra operations are done with Nvidia's CUBLAS library. We have used GPUs to speed up most of the performance-critical parts of the self-consistent field (SCF) iteration. All our calculations use double-precision arithmetic.

Figure 9.2 shows a flowchart of a typical SCF loop in ground-state calculations. The most computationally expensive parts are construction of the Hamiltonian, subspace diagonalization, iterative updating of wave functions, and orthonormalization. The wave functions are stored in a four-dimensional array where each value corresponds to a point in the coarse 3D real space grid for a wave function of a particular electronic state. They consume most of the memory required for the calculation. The low bandwidth of the PCI-Express connection between the host and the device can be a significant bottleneck in the calculations. To combat that, we make an initial guess for the wave functions using the CPU and then transfer them to the GPU and only transfer them back after the SCF iteration has converged. Thus, all the computations that operate on the wave functions are entirely performed on the GPUs. The electron densities are stored in a dense 3D array. These are transferred between the GPU and the CPU depending on where they are needed at that point of the computation.

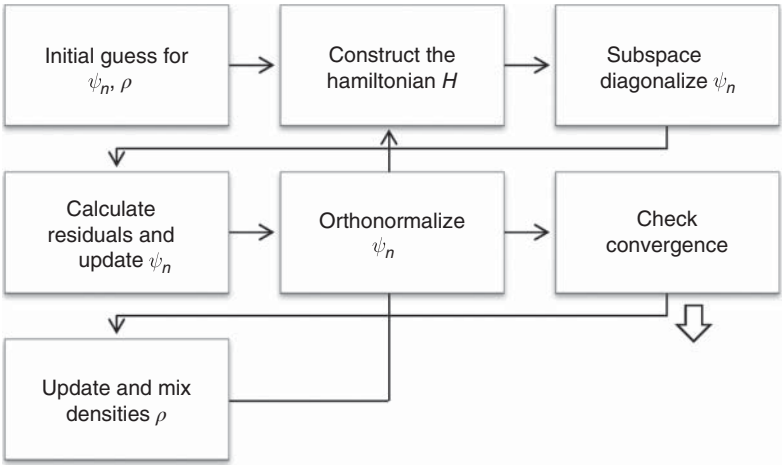


Figure 9.2 *Flowchart of the SCF loop*

For the Hamiltonian operator, the most time-consuming parts are the calculation of the Hartree and XC potentials. The Hartree potential is solved from the Poisson equation using a multigrid solver as described in Section 9.2.3. The basic operations are finite-difference stencils for the Laplace operator, and restriction and interpolation between coarser and finer grids. We have created custom CUDA kernels for all these operations. At the start of the Poisson solver, we transfer the grids containing the charge density and the initial guess for the potential to the device. We then iterate the solution on the GPU using CUDA kernels for the stencil operations, which are described in more detail later on.

The XC potential is calculated on the CPU using an external LibXC library [26]. It is a reusable library containing hundreds of different functional, and it is used by several different codes. We use a separate Python thread for XC calculations. This allows us to overlap the Poisson solver, which is done mainly on the GPU and the computation of XC potential on the CPU.

In the PAW method, one needs to often calculate integrals between atom-centered localized functions multiplied by functions spanning the whole coarse grid such as the projector-wave function integrals $\langle \tilde{p}_i^a | \tilde{\psi}_n \rangle$. In the GPU version, we have written custom CUDA kernels for the integration. The code is parallelized over a batch of grids, localized functions, and coarse grid points inside the augmentation sphere. For each thread located in the coarse grid inside the sphere, we use the secant method to search for the corresponding value of the localized function. We then multiply it by the function value at the grid point. The result is stored in shared memory. We then perform a parallel reduction in shared memory for each thread block and store the result in a temporary array. The same reduction kernel used for dot products is then called to complete the global reduction for each localized function.

The iterative updating of the eigenvectors (RMM-DIIS algorithm) is performed entirely on the GPUs. The basic operation is applying the Hamiltonian to the wave functions, which includes the finite-difference stencil for the Laplace operator. The multigrid preconditioner is very similar to the Poisson equation involving in addition to difference stencils also restriction and interpolation between coarser and finer grids. The PAW corrections to the Hamiltonian involve also projector–wave function integrals.

The computationally most intensive parts of subspace diagonalization and orthonormalization are matrix–matrix operations, which are performed using hybrid functions simultaneously on CPUs and GPUs, as will be described later on in more detail. In addition, the Hamiltonian operator and the

overlap operator are applied to the wave functions on the GPU. The Cholesky decomposition (in orthonormalization) and dense matrix diagonalization (in subspace diagonalization) are performed on the CPU using Scalapack.

9.3.1 Stencil Operations

Various stencil operations take up a large part of the execution time in the grid-based code, and we use CUDA kernels to accelerate these. The GPU versions of 3D finite difference, restriction, interpolation, and Jacobi relaxation kernels all use a similar approach and process the grid slice by slice [27]. Global memory read redundancy is reduced by performing the calculations from shared memory. Each *YZ*-slice of the grid is divided into 2D thread blocks. Each thread reads one grid point from the global memory to the shared memory. Also, data required for the stencil halos is added to shared memory. Each thread then calculates the stencil operator for one grid point. For the *YZ*-slice, data is read from the shared memory. Data required for the *X*-axis calculations is stored in registers for each thread. The working slice is then moved along the *X*-axis of the grid to completely process the grid. Our implementation automatically generates custom CUDA kernels for each order-k stencil from a single C source code base. All operations support real and complex grids and finite and periodic boundary conditions.

The stencils are applied either to the dense grids or to the coarse grid wave functions. Since the same operation is normally performed on all the wave functions, we have implemented batching versions of most of our kernels, which allow us to update a block of grids simultaneously on a GPU. Especially with multigrid methods and small grid sizes, this increases the output bandwidth considerably. When comparing single GPU to single CPU core, the speedups for stencil operations are typically between 10 and 40 depending on the grid size.

9.3.2 Hybrid Level 3 BLAS Functions

A few BLAS level 3 matrix–matrix computations consume a large part of the time in subspace diagonalization and orthonormalization. As subspace diagonalization and orthonormalization scale cubically with the number of atoms in system, they dominate the total computing time in large systems. Generally, the matrix–matrix operations are computationally bound, and they can be split into independent sub-problems which require very little communication between each other. Thus, we have developed hybrid routines that perform part of the computation in the CPU and part in the GPU.

A BLAS DGEMM (or ZGEMM in the complex case) routine performs a matrix–matrix multiplication $C = \alpha A \cdot B + \beta C$, where α and β are scalars and A, B, and C are matrices with dimensions of $m \times k$, $k \times n$, and $m \times n$. In GPAW, the dimensions of the matrices are usually number of wave functions times number of wave functions or number of wave functions times the number of grid points where the latter matrix is several times larger than the first one. We can now divide the A and C matrices into C_1, C_2, A_1, and A_2

$$\left[\frac{C_1}{C_2}\right] = \alpha \left[\frac{A_1}{A_2}\right] [B] + \beta \left[\frac{C_1}{C_2}\right] \tag{9.20}$$

and use CUBLAS GEMM on the GPU to calculate C_1, and simultaneously, BLAS GEMM on the CPU to calculate C_2. The matrices are initially stored on the device memory. We need to transfer the matrices A_2, B, and C_2 to host memory, perform the calculation, and then transfer the C_2 matrix back to GPU memory. On the GPU side, we use several CUDA streams to overlap the CUBLAS calculations and the matrix transfers between host and the device.

In order to distribute the workload effectively and to minimize the total computational time, we need to know the relative speed of GEMM (matrix multiplies) on the GPU and the CPU, as well as

the time required for data transfers across the PCI-Express bus. These can be measured using CUDA events on the GPU. Based on the measurements, we can calculate the average computational and transfer time per output unit. From these we can solve the optimal split value so that the GPU GEMM time equals the sum of the transfer times and the CPU GEMM time.

Each time the hybrid GEMM routine is called, we calculate new values for the average times per unit and use them to update previously calculated and stored values. Approximately similar matrix sizes get grouped together. A small benchmark matrix multiplication is run when the routine is called for the first time to get initial values. During the SCF iteration, the split ratio should converge to an optimal value. Alternatively to splitting A and C row-wise as in Eq. (9.20), it is also possible to split B and C matrices column-wise. In the hybrid GEMM routine, we choose either the row-wise or column-wise split, based on which we minimize the total transfer costs.

For the DSYR2K BLAS routine (ZHER2K in the complex case), we can perform the decomposition as follows:

$$\left[C\right] = \left[C_1\right] + \left[C_2\right] = \alpha \left[A_1 | \, A_2\right] \begin{bmatrix} B_1^H \\ \hline B_2^H \end{bmatrix} + \alpha \left[B_1 | \, B_2\right] \begin{bmatrix} A_1^H \\ \hline A_2^H \end{bmatrix} + \beta \left[C\right]. \tag{9.21}$$

Now we can calculate C_1 on the GPU:

$$C_1 = \alpha \left(A_1 B_1^H + B_1 A_1^H\right) + \beta C. \tag{9.22}$$

At the same time, we transfer A_2 and B_2 asynchronously to the host and then compute on the CPU:

$$C_2 = \alpha \left(A_2 B_2^H + B_2 A_2^H\right). \tag{9.23}$$

We then transfer the resulting matrix C_2 to the device and calculate C. As before, we can estimate and update the optimal decomposition parameters. The hybrid DSYRK and ZHERK functions can be thought of as simplified versions of this.

9.3.3 Parallelization for Multiple GPUs

In GPAW, the high-level parallelization of the code is done with the MPI (message passing interface). Multiple GPUs can be used by domain decomposition of the real-space grid or by parallelizing over **k**-points or spins. Domain decomposition for the finite-difference stencils, restriction, and interpolation as well as PAW integration operations involves communication with the nearest neighbor domains. A robust way that is compatible with all MPI libraries is to do the data transfers between the host and the device manually. This means that we move the data from the device memory to the main memory, then transfer the data to the destination node using MPI, and then move the data from the main memory to the device memory in the destination node. On an individual GPU card, the sends in boundary exchange are done in order, one boundary region at a time. This is to ensure the correctness of results. The receives can happen in any order, but they have to be done for the previous boundary regions before we can send data from the current boundary. The MPI transfers are always done asynchronously. The memory copies between host and device are done either synchronously or asynchronously depending on the size of the boundary regions. With asynchronous transfers, we can overlap sends and receives but the individual transfers have a larger initial overhead.

On large grids we overlap computation with communications. This means that we perform stencil calculations in the middle part of the grid, and at the same time exchange boundary regions with neighboring nodes. We also use batching to combine several small transfers into a few large ones. We also support using CUDA-aware MPI implementations where all the transfers are handled by the MPI library.

9.3.4 Results

We evaluated the performance and scalability of our code by comparing the ground-state DFT calculation times between the GPU and the CPU versions of the code using different simulation setups. For benchmarking purposes, only a fixed number of SCF iterations (usually 10) were performed for each system, and the average time for a single SCF iteration was used as the metric to compare the performance. For small systems, testing was performed with the Triton cluster at Aalto University, which has nine GPU nodes connected to an Infiniband network. Each node has two Intel Xeon X5650 processors and two GPU cards (Nvidia Tesla M2090).

Table 9.1 shows results of benchmarks obtained on the Triton cluster. We performed ground-state calculation with several different carbon nanotubes consisting of 120, 260, and 380 atoms. The smallest simulations with 120 atoms used a single GPU. To demonstrate the effect of the hybrid BLAS functions, we used either one CPU core or all six cores of the Xeon X5650 CPU. Using the threaded BLAS library on the CPU also speeds up other parts of the code that are calculated entirely on the CPU. To be able to fully assess the impact of GPU acceleration, we benchmarked the same simulations without GPU using one and six MPI processes (see Table 9.1(b)). For the 250-atom nanotube we used 4 GPUs and 4 or 24 CPU cores, and for the 380-atom nanotube we employed 8 GPUs and 8 or 48 CPU cores. In the GPU-accelerated version, the use of hybrid BLAS functions yields an additional performance improvement between 8% and 31% depending on the system size. Overall, the speedups between the GPU-accelerated version of the code and the CPU version are between 11.1 and 14.0 when using an equal number of CPU cores and GPUs, and from 3.3 to 4.0 when using all the available recources (CPU cores) during the CPU-only runs.

A larger test for weak scaling of the GPU-accelerated code was performed with the CURIE supercomputer based in France, which has a large hybrid GPU partition with 144 nodes connected to a Infiniband network. Each node has two Intel Xeon E5640 processors and two Nvidia Tesla M2090 GPU cards. Bulk silicon with periodic boundary conditions was selected as a test system. The number of atoms in the test system was increased concurrently with the MPI tasks. The size of the system varied from 8 MPI tasks, 383 atoms, and 1532 wave functions with grid size of $108 \times 108 \times 80$ to 256 tasks, 2047 atoms, and 8188 wave functions with grid size of $216 \times 216 \times 108$. The largest system

Table 9.1 *Performance for ground-state calculations of carbon nanotubes of different lengths (times in seconds, speedup denoted as S-up)*

Atoms/bands	120/480			260/1040			380/1520		
	(a) GPU accelerated ground state calculation								
CPU cores/GPUs	1/1	6/1	S-up	4/4	24/4	S-up	8/8	48/8	S-up
Orthonormalization	1.99	1.38	1.44	3.79	2.84	1.33	5.91	4.57	1.29
DSYRK	0.90	0.52	1.73	0.97	0.84	1.16	1.22	1.03	1.18
DGEMM	0.66	0.60	1.10	1.68	1.44	1.17	2.54	2.14	1.19
Subspace diag.	3.29	2.53	1.30	7.32	5.89	1.24	15.3	10.5	1.44
DSYR2K	1.64	1.08	1.52	1.78	1.53	1.16	2.21	1.93	1.15
DGEMM	0.60	0.54	1.10	1.52	1.29	1.18	2.31	1.91	1.21
Total (SCF)	10.3	9.03	1.14	18.1	16.7	1.08	32.7	25	1.31
	(b) CPU-only ground-state calculation								
CPU cores	1	6		4	24		8	48	
Total (SCF)	114	29.7	3.85	253	59.7	4.24	363	100	3.63
GPU speedup	11.1	3.29		14.0	3.57		11.1	4.00	

(a) GPU accelerated simulations with either one CPU core per GPU or using hybrid BLAS functions and all available CPU cores.
(b) CPU-only simulations, multicore speedup, and total speedup using GPU acceleration.

required ~ 1.3 TB of memory for the calculations. The performance, speedups, and scaling behavior of the CPU and GPU versions of the code are shown in Figure 9.3. The scalability and the performance of the multi-GPU code seems to be very good and consistent even on massive systems using 256 GPUs. The achieved total speedups using one GPU per CPU core varied from 15 to 19.

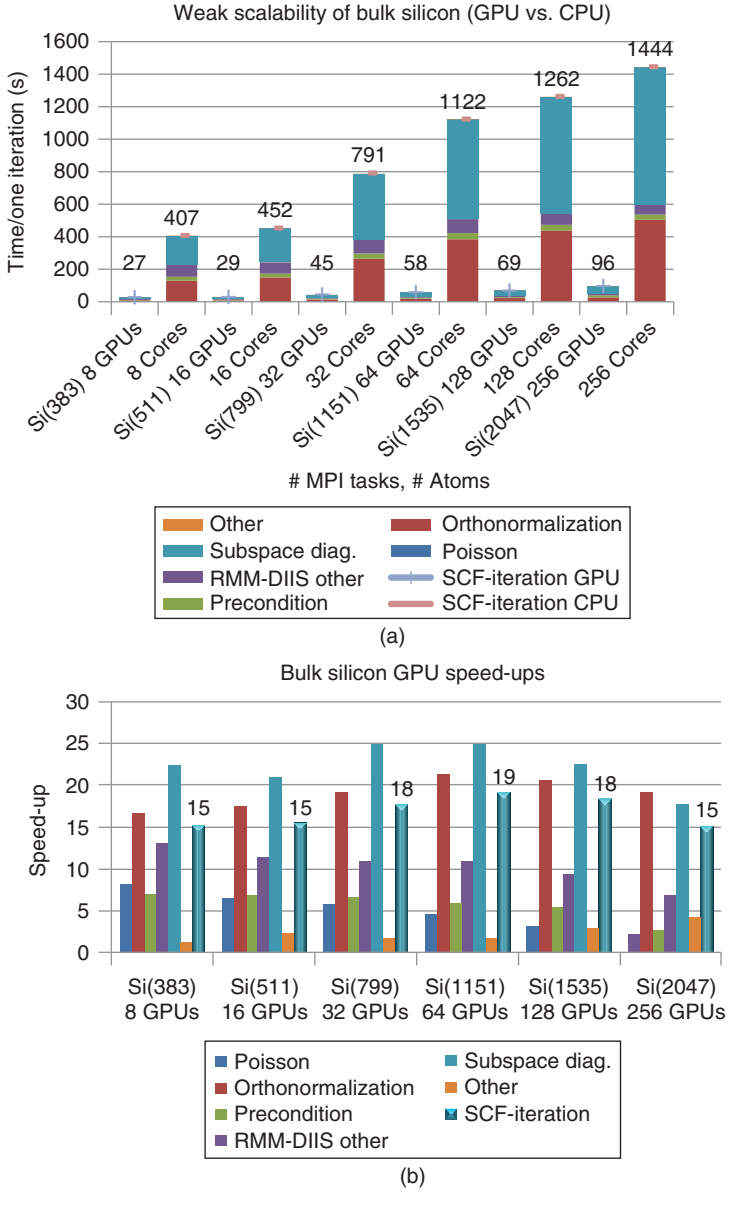


Figure 9.3 *(a) Weak scaling performance of the CPU and GPU versions of the program using bulk Si systems. (b) The achieved speedups with GPU acceleration. The GPU runs used one CPU core per GPU. (See insert for colour representation of this figure)*

9.4 Time-Dependent Density Functional Theory

Excited states and the effects of time-dependent potentials are generally beyond the scope of standard DFT. TD-DFT [28] is a popular method for investigating these properties. In the time propagation scheme, the time-dependent Kohn–Sham equations can be described as

$$i\hbar\frac{\partial}{\partial t}\psi_n(t) = H(t)\psi_n(t). \tag{9.24}$$

This can be transformed to the PAW formalism in the same way as the ground-state equations [29]. The time-depended AE wave functions $\psi_n(t)$ are replaced by smooth PS wave functions $\tilde{\psi}_n(t)$, and the time-dependent equations can then be written as

$$i\hbar\hat{S}\frac{\partial}{\partial t}\tilde{\psi}_n(t) = \hat{H}(t)\tilde{\psi}_n(t), \tag{9.25}$$

where the PAW overlap operator is $\hat{S} = \hat{\mathcal{T}}^\dagger\hat{\mathcal{T}}$ and the PAW time-dependent Hamiltonian is $\hat{H}(t) = \hat{\mathcal{T}}^\dagger H(t)\hat{\mathcal{T}}$. The transformation operator $\mathcal{T}$ is assumed to be time-independent.

GPAW uses a semi-implicit Crank–Nicolson (SICN) method with a predictor and a corrector step to propagate the wave functions. In the predictor step, the Hamiltonian is assumed to be constant during the time step, and the predicted wave functions $\tilde{\psi}_n^{\text{pred}}(t + \Delta t)$ are solved from the equation

$$\left(\hat{S} + \frac{\Delta t}{2\hbar}i\hat{H}(t)\right)\tilde{\psi}_n^{\text{pred}}(t + \Delta t) = \left(\hat{S} - \frac{\Delta t}{2\hbar}i\hat{H}(t)\right)\tilde{\psi}_n(t) + \mathcal{O}(\Delta t^2). \tag{9.26}$$

The predicted Hamiltonian $\hat{H}_{\text{pred}}(t + \Delta t))$ is then calculated using the predicted wave functions. We can approximate the Hamiltonian in the middle of the time step now as

$$\hat{H}\left(t + \frac{\Delta t}{2}\right) = \frac{1}{2}(\hat{H} + \hat{H}_{\text{pred}}(t + \Delta t)), \tag{9.27}$$

which is then used to obtain the propagated wave functions $\tilde{\psi}_n(t + \Delta t)$ in the corrector step:

$$\left(\hat{S} + \frac{\Delta t}{2\hbar}i\hat{H}\left(t + \frac{\Delta t}{2}\right)\right)\tilde{\psi}_n(t + \Delta t) = \left(\hat{S} - \frac{\Delta t}{2\hbar}i\hat{H}\left(t + \frac{\Delta t}{2}\right)\right)\tilde{\psi}_n(t) + \mathcal{O}(\Delta t^3). \tag{9.28}$$

The equations are discretized using uniform real-space grids in the same way as the ground-state solution. The predictor equation (9.26) and the corrector equation (9.28) involve complex symmetric matrices, and a conjugate gradient method is used to solve them.

A linear absorption spectrum can be calculated by applying a weak delta function pulse of a dipole field to the system and then letting the wave functions evolve freely. The optical spectrum is obtained from the Fourier transformation of the time evolution of the dipole moment vector.

9.4.1 GPU Implementation

The computationally most time-consuming part of the SICN algorithm in GPAW is the conjugate gradient solver for complex symmetric matrices that is used both in the predictor and in the corrector steps. The most important operation is the application of the Hamiltonian operator to the wave functions, which involves the finite difference Laplacian as well as projector wave function integrals. Another significant portion is the updating of the time-dependent Hamiltonian. A lot of the high level code is shared between the time propagator code and the ground-state code. In the GPU-accelerated version, all heavy computations are performed with GPUs.

To conserve memory and to speedup computations on the GPU, we use a batching version of the propagator code. The predictor and the corrector steps are calculated simultaneously for a block of wave functions. A copy of the wave functions is stored in the host memory at the start of the

Table 9.2 *Bulk Au time-propagation example*

Atoms/bands	48/268			96/536			144/804		
CPU cores/GPUs	1/0	1/1	S-up	4/0	4/4	S-up	8/0	8/8	S-up
Apply operators	30.4	2.83	10.7	31.9	2.79	11.4	35.0	3.51	9.97
Conjugate gradient	1190	106	11.2	1270	109	11.7	1380	136	10.1
Update operators	19.9	8.44	2.36	139	18.7	7.43	387	14.7	26.3
Propagation	1250	119	10.5	1450	132	11.0	1820	156	11.7

The times are averages for one propagation step in seconds, and the comparison is between equal numbers of GPUs and CPU cores.

propagator step and then transferred back block by block to the GPU when it is needed. This is done to avoid storing several copies of the wave function matrices in the limited GPU memory. Batching is also used in the conjugate gradient solver. The same CUDA kernels and libraries used in the DFT code are reused in the TD-DFT code.

The GPU TD-DFT code is parallelized in the same way as the ground-state code: that is, with domain decomposition over real-space grids. As the parallelization over electronic states is trivial in time-propagation TD-DFT, we have also implemented band parallelization.

9.4.2 Results

Table 9.2 shows an example time-propagation calculation performed on the Vuori cluster at CSC, Finland, consisting of Intel Xeon X5650 CPUs and Nvidia Tesla M2050 GPUs. Bulk gold systems are excited with a weak delta kick, and wave functions are propagated with an SICN propagator. The computation is performed with three different systems using 1–8 MPI tasks. Parallelization is done with a combination of domain decomposition and band parallelization. The smallest system has 48 gold atoms with 268 bands in the calculation, and the largest one has 144 atoms with 804 bands.

The computations are performed either using GPUs or CPUs. From the results we can see that the vast majority of time in both the CPU and GPU versions is spent in the conjugate gradient solver. Updating the operator also includes some functions (e.g., the XC potential), which are computed entirely on the CPU and which can consume a significant portion of time for small systems. Overall, the achieved speedups (using one GPU per CPU core) in the tested systems vary between 10.5 and 11.7. The scalability of the GPU-accelerated code is comparable to that of the original CPU code.

9.5 Random Phase Approximation for the Correlation Energy

As material simulations and predictions using DFT methods advance, DFT with semilocal density-based XC functionals face huge challenges in describing systems with long-range van der Waals interactions, strong correlation, localization, and so on. Climbing the so-called Jacob's ladder for XC functionals, one goes beyond density-based into orbital-based XC functionals. Exact exchange (EXX) includes the exchange interaction between orbitals and is self-interaction-free. However, EXX (or Hartree–Fock) alone has repulsion that is too strong and gives, for example, HOMO–LUMO values for molecules and band gaps for solids that are too large. Typical hybrid XC functionals take into account 20-30% of the EXX contribution, plus other semilocal XC contributions. However, the amount of the EXX included is empirical. Correlation using the RPA is orbital-based and fully nonlocal. As RPA is derived from the adiabatic connection fluctuation-dissipation theorem (ACFDT), EXX and RPA work naturally together, with 100% of EXX included and without any empirical parameters. EXX+RPA has been shown to systematically improve lattice constants [30],

atomization and cohesive energies [31], formation enthalpies [32], adsorption sites and energies [33–35], reaction barriers [36], structural phase transitions [37], and polymorphic energy ordering [38] for a wide range of systems that have ionic, covalent, or van der Waals interactions [39–42].

Similar to many other beyond-DFT orbital-based calculations, the RPA method is exceptionally computationally demanding. The challenges are twofold: First, it involves, per calculation, summations over large basis sets and hundreds to thousands of virtual orbitals, both of which are truncated above a certain energy. Second, the method scales as $\mathcal{O}(N^4)$, where N is the number of electrons. Methods that have been used to reduce the computational burden of RPA so far include reducing or eliminating the number of virtual orbitals by using approximations [43, 44]; more rigorous perturbation theory [45–47], or Green's functions; and range-separated RPA to achieve faster convergence [48]. Here, we address high computational burden by using GPUs without making any approximations or methodology change to the RPA method [49]. It is interesting, although beyond the scope of this article, to explore the combination of GPU and other improvements to the RPA method.

According to the ACDFT theory, the RPA correlation energy $E^{\mathrm{c}}_{\mathrm{rpa}}$, represented with a plane-wave basis set, is written as

$$E^{\mathrm{c}}_{\mathrm{rpa}} = \int_0^\infty \frac{d\omega}{2\pi} \int_{\mathrm{BZ}} d\mathbf{q} \mathrm{Tr}\{ \ln\ [1 - v_{\mathbf{G}}(\mathbf{q})\chi^0_{\mathbf{GG'}}(\mathbf{q}, i\omega)] + v_{\mathbf{G}}(\mathbf{q})\chi^0_{\mathbf{GG'}}(\mathbf{q}, i\omega)\}, \tag{9.29}$$

where $\mathbf{q}$ is a wave vector within the Brillouin zone (BZ), $\mathbf{G}$ is a reciprocal space lattice vector the size of which is defined by a cutoff energy E_{cut}, ω is the frequency, and v is the Coulomb interaction kernel. The noninteracting response function χ^0 is

$$\chi^0_{\mathbf{GG'}}(\mathbf{q}, i\omega) = \frac{2}{\Omega} \sum_{\mathbf{k}}^{\mathrm{BZ}} \sum_{nn'} \frac{f_{n\mathbf{k}} - f_{n'\mathbf{k+q}}}{i\omega + \epsilon_{n\mathbf{k}} - \epsilon_{n'\mathbf{k+q}}} n_{n\mathbf{k},n'\mathbf{k+q}}(\mathbf{G}) n^*_{n\mathbf{k},n'\mathbf{k+q}}(\mathbf{G'}), \tag{9.30}$$

where

$$n_{n\mathbf{k},n'\mathbf{k+q}}(\mathbf{G}) \equiv \langle \psi_{n\mathbf{k}} | e^{-i(\mathbf{q+G})\cdot\mathbf{r}} | \psi_{n'\mathbf{k+q}} \rangle \tag{9.31}$$

is the charge density matrix, and Ω is the volume of the unit cell. The occupations $f_{n\mathbf{k}}$, Kohn–Sham eigenvalues $\epsilon_{n\mathbf{k}}$, and eigenstates $\psi_{n\mathbf{k}}$ for band n at wave vector $\mathbf{k}$ are obtained from a ground-state DFT calculation, which can be performed in any basis supported by GPAW (i.e., real-space grids, plane waves, or localized orbitals).

9.5.1 GPU Implementation

In the actual numerical implementation, the frequency integration over ω in Eq. (9.29) is carried out using 16 Gauss–Legendre points following the procedure from Ref. [40]. The integration over the BZ is discretized using Monkhorst–Pack k-points, the size of which depends on whether one calculates metals or semiconductors and is on the order of 10^1–10^3. By exploiting the $\mathbf{q}$-mesh symmetry, the integration is reduced to a summation over the irreducible BZ (IBZ) : $\int_{\mathrm{BZ}} \rightarrow \sum_{\mathrm{IBZ}} w_{\mathbf{q}}$, where $w_{\mathbf{q}}$ is the weight for a specific $\mathbf{q}$ vector. The number of $\mathbf{G}$ vectors is on the order of 10^2–10^3, making the χ^0 matrix size difficult to store in the 2–3 GB of memory typically available on GPU cards at present. As a result, the $\mathbf{q}$ index is looped over, and for each $\mathbf{q}$ the $\chi^0(\omega, \mathbf{G}, \mathbf{G'})$ matrix is evaluated as (the pair of indices in the parentheses indicate the size of the matrix)

$$\chi^0(i\omega, \mathbf{G}, \mathbf{G'}) = \sum_{\mathbf{k},n,u \subset n'} A(u, i\omega) n(u, \mathbf{G}) n^*(u, \mathbf{G'}), \tag{9.32}$$

where u is a subset of index n', and $n(u, \mathbf{G})$ is a matrix, with each column representing a vector $n(\mathbf{G})$ in Eq. (9.31) at a particular n, n', and $\mathbf{k}$. In practice, the n' index is looped over the subset u, and for each electron–hole pair at $(n, \mathbf{k})$ and $(n', \mathbf{k} + \mathbf{q})$, the $n(\mathbf{G})$ vector is calculated according to Eq. (9.31), and

stored as a column in the matrix $n(u, \mathbf{G})$. When all the matrix elements of the $n(u, \mathbf{G})$ are calculated, a ZHERK routine in the BLAS or CUBLAS library is called and the result is added to the $\chi^0(i\omega, \mathbf{G}, \mathbf{G}')$ matrix. Such a process is repeated for all the $\mathbf{k}$, n, and u indices until the summations are complete. Similar to all the other physical quantities in the PAW method, the calculation of $n(\mathbf{G})$ consists of a smooth pseudo-part and an atom-centered PAW part. The details on how to evaluate $n(\mathbf{G})$ in the PAW method can be found in Refs [49, 50].

The calculation of the $\chi^0(i\omega, \mathbf{G}, \mathbf{G}')$ matrix is parallelized by dividing the summation onto the processors of each node using MPI. Since each processor computes its own set of $n(u, \mathbf{G})$, there is no communication during the calculations. Thus, MPI communications are only required at the beginning of the calculation, to setup and distribute indices, and so on, and at the end of the calculation, to sum up the χ^0 matrix on each processor. Linear scaling with both CPU and GPU count is thus expected. The χ^0 matrix is also stored in parallel if necessary.

The response function code has been entirely moved to the GPU (no "thunking"). The GPU port was done in three steps. In the first step, all the BLAS and FFTW routines were replaced with CUBLAS and batched CuFFT routines. In the second step, the finite difference stencil code on CPU, in the case of $\mathbf{q} \to 0$, was reformulated using CuFFT. In the third step, we wrote customized CUDA kernels for the rest of the code. This included obtaining and transforming wave functions using symmetry operations, mapping wave functions between the 3D FFT grid and reduced planewave grid defined by a cutoff radius, as well as the PAW corrections to the $n(\mathbf{G})$. The size of the code is roughly 6000 lines of python and 1000 lines of C/CUDA (many GPAW functions are reused and not counted here). The source code is available for download.[1]

9.5.2 Performance Analysis Techniques

To study the performance of the code, we have used the Nvidia "nvvp" profiling tool. As shown in Figure 9.4, the GPU is typically well utilized during RPA calculations, but there is a period where the GPU goes idle, waiting for the CPU. There are three labeled lines in the plot. "CPU kernel launches" shows various CUDA kernel launches, with the labeled brown boxes showing kernel launches that do not immediately return, indicating that the CPU is blocked waiting for the GPU. The "CPU routine markers" line was produced by inserting nvvp "range markers" in the code to indicate which CPU routines were running at which time. The "GPU utilization" line shows that the GPU is idle between 17.5 and 18.5 ms.

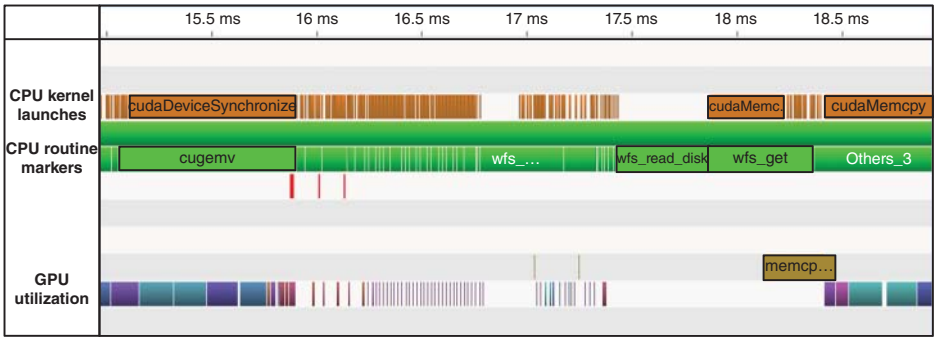


Figure 9.4 *Output of the Nvidia nvvp profiling tool for a portion of an RPA Li_2O calculation*

[1] `https://trac.fysik.dtu.dk/projects/gpaw/browser/branches/rpa-gpu-expt`.

During the idle period, the CPU is reading wave functions from disk (routine `wfs_read_disk` in the "CPU routine markers" line). Performance of the code could, in principle, be improved by prefetching these wave functions before launching the `cudaDeviceSynchronize` call, which blocks the CPU at ~15 ms. Such an overlap between CPU and GPU work would be possible because there are no dependencies between the new wave function data fetched and the results calculated from preceding wave function data.

9.5.3 Results

Figure 9.5 shows the speedup for a test system N_2/Ru(0001) surface with 14 atoms in the unit cell. These results were obtained running on a system with two 6-core Intel X5650 CPUs and eight Nvidia C2075 GPUs. The speedup for a few representative functions is shown as black lines, and the total speedup is shown in red. As the number of u increases, the speedup of most functions plateaus. Some of them, such as ZHERK and our customized kernel ("paw_P_ai, " "mapG, " and so on), transform themselves from being memory-bound to being compute/latency-bound: the number of kernel launches is reduced, reducing the kernel launch overhead, and the cudaMemcpy is executed with a larger amount of data per copy. For large N_u, the FFT operations, employed by calling the CuFFT library in a batched manner, gain the least speedup (12), compared to FFTW. This is not surprising since FFT operations are highly nonlocalized. The CUBLAS ZHERK routine gains a speedup of 27 compared to MKL BLAS. The maximum speedup of 36 for ZHERK is not achieved here because we did not choose a large enough system to stress the GPUs; instead we chose a system that was more scientifically interesting. Our own customized kernels "mapG" and "paw_P_ai" gain the most significant speedup by parallelizing over as many indices as possible using GPU threads. For example, in the

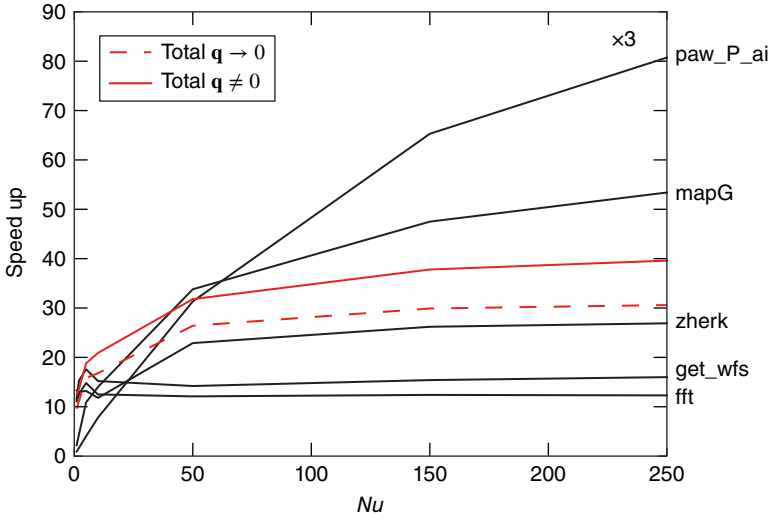


Figure 9.5 *Speedup (eight GPUs vs. eight CPU cores) as a function of the number of the u (Nu) for some representative functions such as get wave functions ("get_wfs"), mapping wave functions between the 3D FFT grid and reduced planewave grid ("mapG"), PAW corrections ("paw_P_ai"), batched CUFFT ("fft"), and CUBLAS ("zherk" routine). For a full list of the GPU ported functions, refer to Ref. [49]. The test system is a N_2/Ru(0001) surface, modeled with four layers of Ru in a $\sqrt{3} \times \sqrt{3}$ unit cell. The speedup (timing) information comes from a summation of 1 k-point (per core), 5 occupied, and 1486 unoccupied bands with an energy cutoff 150 eV. The total speedups ("Total") in the optical limit ($\mathbf{q} \to 0$) and other $\mathbf{q} \neq 0$ are also shown.*

Table 9.3 *Speedup (eight GPUs/eight CPU cores) for* $\mathbf{q} \neq 0$ *as well as time required for an entire RPA calculation with a response function cutoff of 150 eV for different simulations*

System	Phase	N_a	N_e	Spin	k-Points	Speedup	t_{gpu}
O_2	Gas	2	12	True	1	11.3×	41 seconds
Li_2O	Bulk	3	8	False	$4 \times 4 \times 4$	10.5×	63 seconds
MoO_3	Bulk	16	96	False	$4 \times 2 \times 4$	35.3×	1.0 hour
N_2/Ru(0001)	Surface	14	202	False	$4 \times 4 \times 1$	36.1×	1.4 hours
CO/Ni(111)	Surface	22	210	True	$4 \times 4 \times 1$	37.0×	5.5 hours

N_a denotes the number of atoms and N_e the number of electrons in the unit cell. Spin polarization is indicated in the column "Spin."

CPU code, the PAW corrections must loop over atoms, bands, and projector functions. In the GPU implementation, we parallelize all the atoms, bands, and projector function indices simultaneously using threads.

With a typical speedup of 40, RPA calculations can be performed in a timely manner using eight GPUs on the systems shown in Table 9.3. The speedup in the noninteracting response function makes it also applicable to other beyond-DFT methods such as GW and Bethe–Salpeter equation [51].

9.6 Summary and Outlook

The PAW method together with uniform real-space grids as implemented in the open-source software package GPAW has proven to be an accurate and scalable computational method for DFT-based calculations on traditional CPU-based supercomputers. As GPUs have been promising in speeding up DFT-based calculations, we have ported GPAW to utilize also GPUs.

The ground-state algorithm in GPAW contains several parts with non-negligible computation time including solution of the Poisson equation with the multigrid method, projector function–wave function integrals, application of the Hamiltonian operator to the wave functions, multigrid preconditioning and dense matrix–matrix products in subspace diagonalization, and orthonormalization of wave functions. All these parts can be executed on GPUs either with custom CUDA kernels or with the CUBLAS library. Multiple GPUs can be utilized with the MPI-based implementation. We have implemented also hybrid versions of special matrix–matrix operations, which can utilize concurrently both CPUs and GPUs. In single CPU core–GPU comparison, the individual computational kernels achieve speedups of a factor of 10–40, and the whole calculation can obtain speedups up to ∼20. Also, we have demonstrated that the multi-GPU ground-state algorithm scales efficiently at least to 256 GPUs. In typical hardware configurations, there are several CPU cores available per GPU, and in a CPU node–GPU node comparison the speedup for the full calculation is typically 3–4. This speedup is significant and justifies the use of GPUs when considering acquisition price versus performance or energy consumption versus performance.

GPAW implements also TD-DFT for studies of excited state properties. As the TD-DFT real-time propagation algorithm utilizes largely the same kernels as the ground-state algorithm, the GPU-based implementation is relatively straightforward based on the existing ground-state GPU implementation. The achievable speedup and scalability of TD-DFT calculations is similar to those of ground-state calculations.

For special GPAW features, it is possible to obtain very high speedups with GPUs. The RPA to the correlation energy is computationally very demanding, and the algorithm itself is well suited for GPUs. The RPA GPU implementation is also able to utilize multiple GPUs and achieves speedups of almost 40 in single CPU core–single GPU comparison.

Generally, DFT-based methods contain several time-consuming computational kernels whose optimization for GPUs is nontrivial. Thus, it is expected that there is room for further improvements in the GPU implementation of GPAW. In addition, hardware developments (e.g., caches in GPUs) as well as developments in the compilers are likely to improve the performance of GPU implementations, and make it also easier to implement new features utilizing GPUs. Even though GPUs might not become main stream for DFT-based simulations, there are clearly application cases where GPUs are highly competitive.

Acknowledgments

This work has been supported by the Academy of Finland (Project 110013 and the Center of Excellence program). This research used resources of CSC–IT Center for Science Ltd, Finland, and PRACE Research Infrastructure resource CURIE based in France at GENCI/CEA. Work on the RPA method was supported by the U.S. Department of Energy Chemical Sciences, Geosciences, and Biosciences Division under the Materials Genome Initiative: Predictive Theory of Transition Metal Oxide Catalysis grant.

References

1. Hohenberg, P. and Kohn, W. (1964) Inhomogeneous electron gas. *Phys. Rev.*, **136** (3B), B864–B871.
2. Kohn, W. and Sham, L.J. (1965) Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, **140** (4A), A1133–A1138.
3. Mortensen, J.J., Hansen, L.B. and Jacobsen, K.W. (2005) Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B*, **71**, 035109.
4. Enkovaara, J., Rostgaard, C., Mortensen, J.J., Chen, J., Dułak, M., Ferrighi, L. *et al*. (2010) Electronic structure calculations with GPAW: a real-space implementation of the projector augmented-wave method. *J. Phys. Condens. Matter*, **22** (25), 253202.
5. Payne, M.C., Teter, M.P., Allan, D.C., Arias, T.A. and Joannopoulos, J.D. (1992) Iterative minimization techniques for *ab initio* total-energy calculations: molecular dynamics and conjugate gradients. *Rev. Mod. Phys.*, **64** (4), 1045–1096.
6. Gonze, X., Beuken, J.M., Caracas, R., Detraux, F., Fuchs, M., Rignanese, G.M. *et al*. (2002) First-principle computation of material properties: the ABINIT software project. *Comput. Mater. Sci.*, **25**, 478.
7. Kresse, G. and Furthmüller, J. (1996) Efficiency of Ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. *Comput. Mater. Sci.*, **6**, 15–50.
8. Giannozzi, P., Baroni, S., Bonini, N., Calandra, M., Car, R., Cavazzoni, C. *et al*. (2009) QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. *J. Phys. Condens. Matter*, **21** (39), 395502 (19pp). Available from `http://www.quantum-espresso.org`.
9. Soler, J.M., Artacho, E., Gale, J.D., García, A., Junquera, J., Ordejón, P. *et al*. (2002) The SIESTA method for Ab initio order- N materials simulation. *J. Phys. Condens. Matter*, **14** (11), 2745.
10. Ahlrichs, R., Bär, M., Häser, M., Horn, H. and Kölmel, C. (1989) Electronic structure calculations on workstation computers: the program system Turbomole. *Chem. Phys. Lett.*, **162** (3), 165–169.
11. Chelikowsky, J.R., Troullier, N. and Saad, Y. (1994) Finite-difference-pseudopotential method: electronic structure calculations without a basis. *Phys. Rev. Lett.*, **72** (8), 1240–1243.

12. Briggs, E.L., Sullivan, D.J. and Bernholc, J. (1995) Large-scale electronic-structure calculations with multigrid acceleration. *Phys. Rev. B.*, **52** (8), R5471–R5474.
13. Marques, M.A.L., Castro, A., Bertsch, G.F. and Rubio, A. (2003) Octopus: a first-principles tool for excited electron-ion dynamics. *Comput. Phys. Commun.*, **151**, 60–78.
14. Blum, V., Gehrke, R., Hanke, F., Havu, P., Havu, V., Ren, X. *et al*. (2009) Ab initio molecular simulations with numeric atom-centered orbitals. *Comput. Phys. Commun.*, **180** (11), 2175–2196.
15. Tsuchida, E. and Tsukada, M. (1995) Electronic-structure calculations based on the finite-element method. *Phys. Rev. B*, **52**, 5573–5578.
16. Pask, J.E. and Sterne, P.A. (2005) Finite element methods in *ab initio* electronic structure calculations. *Modell. Simul. Mater. Sci. Eng.*, **13**, R71–R96.
17. Lehtovaara, L., Havu, V. and Puska, M. (2009) All-electron density functional theory and time-dependent density functional theory with high-order finite elements. *J. Chem. Phys.*, **131** (5), 054103.
18. Blöchl, P.E. (1994) Projector augmented-wave method. *Phys. Rev. B*, **50** (24), 17953–17979.
19. Runge, E. and Gross, E.K.U. (1984) Density-functional theory for time-dependent systems. *Phys. Rev. Lett.*, **52**, 997–1000.
20. Blöchl, P.E. (1994) Projector augmented-wave method. *Phys. Rev. B*, **50**, 17953–17979.
21. Brandt, A. (1977) Multi-level adaptive solutions to boundary-value problems. *Math. Comput.*, **31**, 333.
22. Briggs, W.L., Henson, V.E. and McCormick, S.F. (2000) *A Multigrid Tutorial*, 2nd edn, Society for Industrial and Applied Mathematics, Philadelphia, PA.
23. Hakala, S., Havu, V., Enkovaara, J. and Nieminen, R. (2013) Parallel electronic structure calculations using multiple graphics processing units (GPUs), in *Applied Parallel and Scientific Computing*, Lecture Notes in Computer Science, vol. **7782** (eds P. Manninen and P. Oster), Springer-Verlag, Berlin Heidelberg, pp. 63–76.
24. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P. and Fasih, A. (2012) PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation. *Parallel Comput.*, **38** (3), 157–174.
25. NVIDIA Corp. CUDA Parallel Computing Platform. `http://www.nvidia.com/object/cuda_home_new.html` (accessed 14 October 2013).
26. Libxc. `http://www.tddft.org/programs/octopus/wiki/index.php/Libxc` (accessed 15 September 2015).
27. Micikevicius, P. (2009) 3D finite difference computation on GPUs using CUDA, in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ACM, New York, pp. 79–84.
28. Yabana, K. and Bertsch, G.F. (1996) Time-dependent local-density approximation in real time. *Phys. Rev. B*, **54**, 4484–4487.
29. Walter, M., Häkkinen, H., Lehtovaara, L., Puska, M., Enkovaara, J., Rostgaard, C. *et al*. (2008) Time-dependent density-functional theory in the projector augmented-wave method. *J. Chem. Phys.*, **128** (24), 244101.
30. Harl, J., Schimka, L. and Kresse, G. (2010) Assessing the quality of the random phase approximation for lattice constants and atomization energies of solids. *Phys. Rev. B*, **81**, 115126.
31. Harl, J. and Kresse, G. (2008) Cohesive energy curves for noble gas solids calculated by adiabatic connection fluctuation-dissipation theory. *Phys. Rev. B*, **77**, 045136.
32. Yan, J., Hummelshøj, J.S. and Nørskov, J.K. (2013) Formation energies of group I and II metal oxides using random phase approximation. *Phys. Rev. B*, **87**, 075207.
33. Ren, X., Rinke, P. and Scheffler, M. (2009) Exploring the random phase approximation: application to CO adsorbed on Cu(111). *Phys. Rev. B*, **80**, 045402.

34. Schimka, L., Harl, J., Stroppa, A., Grüneis, A., Marsman, M., Mittendorfer, F. *et al*. (2010) Accurate surface and adsorption energies from many-body perturbation theory. *Nat. Mater.*, **9**, 741.
35. Paz-Borbón, L.O., Barcaro, G., Fortunelli, A. and Levchenko, S.V. (2012) Au_N clusters (N = 1–6) supported on MgO(100) surfaces: effect of exact exchange and dispersion interactions on adhesion energies. *Phys. Rev. B*, **85**, 155409.
36. Ren, X., Rinke, P., Joas, C. and Scheffler, M. (2012) Random-phase approximation and its applications in computational chemistry and materials science. *J. Mater. Sci.*, **47**, 7447.
37. Xiao, B., Sun, J., Ruzsinszky, A., Feng, J. and Perdew, J.P. (2012) Structural phase transitions in Si and SiO_2 crystals via the random phase approximation. *Phys. Rev. B*, **86**, 094109.
38. Peng, H. and Lany, S. (2013) Polymorphic energy ordering of MgO, ZnO, GaN, and MnO within the random phase approximation. *Phys. Rev. B*, **87**, 174113.
39. Harl, J. and Kresse, G. (2009) Accurate bulk properties from approximate many-body techniques. *Phys. Rev. Lett.*, **103**, 056401.
40. Olsen, T., Yan, J., Mortensen, J.J. and Thygesen, K.S. (2011) Dispersive and covalent interactions between graphene and metal surfaces from the random phase approximation. *Phys. Rev. Lett.*, **107**, 156401.
41. Dobson, J.F. and Gould, T. (2012) Calculation of dispersion energies. *J. Phys. Condens. Matter*, **24**, 073201.
42. Lu, D., Li, Y., Rocca, D. and Galli, G. (2009) *Ab initio* calculation of van der Waals bonded molecular crystals. *Phys. Rev. Lett.*, **102**, 206411.
43. Bruneval, F. and Gonze, X. (2008) Accurate *GW* self-energies in a plane-wave basis using only a few empty states: towards large systems. *Phys. Rev. B*, **78**, 085125.
44. Berger, J.A., Reining, L. and Sottile, F. (2010) *Ab initio* calculations of electronic excitations: collapsing spectral sums. *Phys. Rev. B*, **82**, 041103.
45. Umari, P., Stenuit, G. and Baroni, S. (2010) GW quasiparticle spectra from occupied states only. *Phys. Rev. B*, **81**, 115104.
46. Giustino, F., Cohen, M.L. and Louie, S.G. (2010) GW method with the self-consistent Sternheimer equation. *Phys. Rev. B*, **81**, 115105.
47. Rocca, D., Ping, Y., Gebauer, R. and Galli, G. (2012) Solution of the Bethe-Salpeter equation without empty electronic states: application to the absorption spectra of bulk systems. *Phys. Rev. B*, **85**, 045116.
48. Bruneval, F. (2012) Range-separated approach to the RPA correlation applied to the van der Waals bond and to diffusion of defects. *Phys. Rev. Lett.*, **108**, 256403.
49. Yan, J., Li, L. and O'Grady, C. (2013) Graphics processing unit acceleration of the random phase approximation in the projector augmented wave method. *Comput. Phys. Commun.*, **184**, 2728.
50. Yan, J., Mortensen, J.J., Jacobsen, K.W. and Thygesen, K.S. (2011) Linear density response function in the projector augmented wave method: applications to solids, surfaces, and interfaces. *Phys. Rev. B*, **83**, 245122.
51. Yan, J., Jacobsen, K.W. and Thygesen, K.S. (2012) Optical properties of bulk semiconductors and graphene/boron nitride: the Bethe-Salpeter equation with derivative discontinuity-corrected density functional energies. *Phys. Rev. B*, **86**, 045208.

10

Application of Graphics Processing Units to Accelerate Real-Space Density Functional Theory and Time-Dependent Density Functional Theory Calculations

Xavier Andrade and Alán Aspuru-Guzik
Department of Chemistry and Chemical Biology, Harvard University, Cambridge, MA, USA

Real-space density functional theory (DFT) is a powerful and flexible method for the numerical simulation of electronic systems within the Kohn–Sham approach. In real-space DFT the density, orbitals and other fields are represented on a grid and the differential operators are approximated by finite differences. With a proper implementation strategy, the real-space approach offers great opportunities for GPU processing due to naturally coalesced memory accesses, small and simple kernels, and extensive data parallelism. In this chapter, we examine the GPU implementation in real-space of two different types of calculations based on the Kohn–Sham approach, and that are commonly used to characterize electronic systems: the calculation of the ground-state using DFT, and electron dynamics using real-time time-dependent DFT (TDDFT). We provide details of the implementation of the different numerical procedures required for these calculations and how to make them suitable for the particular characteristics of current GPUs. The resulting numerical performance of the calculations is evaluated by absolute measurements, comparison with the optimized CPU version and other GPU implementations based on Gaussian basis sets.

Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

10.1 Introduction

As was discussed in detail in Chapter 2 designing a high-throughput processor like a graphics processing unit (GPU) requires engineers to make certain tradeoffs to stay within transistor and power budgets. These tradeoffs need to be taken into account when designing code to run on GPUs. As was discussed in Chapter 2, GPUs for example require a very large number of independent instructions to be executed in parallel in order to hide execution latency. Another important aspect of a GPU is that control units are shared among execution units and one therefore needs to be careful when designing code that shall execute different instructions in parallel, otherwise these will be performed in series.

These characteristics make the GPU architecture more suitable to certain types of numerical work than others. Given the complexity of the electronic structure problem and the diversity of methods used to tackle it by chemists and physicists, the use of GPUs to simulate electronic systems, as illustrated by this book and the following references, is an active area of research [1–19].

One of the most successful approaches for molecular electronic structure is the use of Gaussian type orbitals (GTOs) [20]. It is not surprising then, that the GTO approach was probably one of the first electronic structure methods to be implemented on GPUs [1, 21, 22]. However, there are some inherent characteristics of the methods that make the implementation of the GTO approach on GPUs complicated. The analytic evaluation of integrals, one of the most time consuming parts of this type of calculations, requires different formulations for different angular momentum components, making the code irregular. Moreover, the integrals that involve high-angular momentum components, required for heavy elements or high-accuracy calculations, are challenging to evaluate efficiently on the GPU [23, 24]. For example in the code Terachem [25, 26], discussed earlier in Chapter 4, the evaluation of integrals for d-type orbitals on the GPU requires the use of automatic code generation [24], and, at the time of writing, f-type orbitals are not supported.

A more straightforward approach for electronic structure is the real-space grid discretization. Probably, the first real-space implementation for polyatomic molecules was presented by Becke in 1989 [27], as a method for basis-set free DFT. His approach uses a set of radial grids centered around each atom. A simpler formulation based on uniform grids and pseudo-potentials was proposed by Chelikosky *et al*. in 1994 [28]. Since then, the real-space approach has become a popular alternative for DFT simulations and several implementations have been presented [29–41] (see also Chapter 9). A particular application of the real-space approach has been in association real-time electron dynamics calculations performed at the TDDFT theory level [42, 43], a combination known as the real-space real-time method.

What makes real-space grids attractive for electronic structure is the flexibility to model different types of electronic systems: molecular, crystalline [44], and low-dimensional model systems [45–47], the systematic control of the discretization error, and its potential for parallelization in distributed memory systems with thousands of processors [7, 48–50]. Recent works have shown that real-space grid methods have great potential for efficient execution on GPU architectures [6–8, 16, 51], with performance that rivals that of GTO calculations on GPUs (Chapter 4).

In this chapter, we discuss the real-space GPU approach we developed for DFT and TDDFT calculations based on the code OCTOPUS [33, 52] by reviewing our previous published work [6–8] and presenting some new advances. Our objective has been the development of efficient methods to perform electronic structure calculations on GPUs. This goes beyond rewriting and optimizing low-level routines for the GPU since choosing an appropriate design strategy for the entirety of the code can be fundamental for optimal GPU performance with complex scientific software. Our discussion is complementary to Chapter 9 that discusses the GPU implementation of real-space DFT with a focus on projector augmented methods and multiple GPUs. Our GPU implementation is based on OpenCL [53], a standard and portable framework for writing code for parallel processors, so it can run on GPUs, CPUs, and other processing devices, from multiple vendors.

In order to assess the efficiency of our implementation, we perform a series of tests involving ground-state and excited-state calculations using high-end GPUs from Nvidia and AMD, and a set of molecular systems of different sizes. We provide different indicators that illustrate the performance of our implementation: numerical throughput (number of floating point operations executed per unit of time), total calculation times, and comparisons with the CPU version of the code and a different GPU–DFT implementation. These results show that real-space DFT is an interesting and competitive approach for GPU-accelerated electronic structure calculations.

10.2 The Real-Space Representation

In the Kohn–Sham (KS) [54] formulation of DFT [55], the electronic density of an interacting electronic system, n, is generated by a set of single-particle orbitals, or states, ψ_k. For numerical simulations in DFT and TDDFT, the orbitals, the density, and other fields need to be represented as a finite set of numbers. The selection of the discretization scheme is probably the most important aspect in the numerical solution of the electronic structure problem.

In the real-space approach [27, 28], instead of a basis, fields are discretized on a grid and operators are approximated by finite-differences. This provides a simple and flexible scheme that is suitable to model both finite and periodic systems [44]. The electron–ion interaction is modeled by either the pseudo-potential approximation, or the projector-augmented wave method [36], that remove the problem of representing the hard Coulomb potential, allowing simple uniform grids to be used.

One of the main advantages of the real-space grid approach is that the discretization error can be controlled systematically by reducing the spacing and, in the case of finite systems, increasing the size of the simulation box. Of course, this increases the number of points and, proportionally, the time and memory cost of the calculation.

Many real-space implementations use parallelepipedic grids that are simple to implement, since the points positions can be easily mapped from the grid to a linear array. However, this has a disadvantage when simulating systems that are finite in one or more directions. In general, the error due to forcing the orbitals to zero outside of the box roughly depends on the smallest distance from any atom to the boundaries of the box. This implies that in a parallelepipedic grid, a large fraction of the points, typically the ones on the corners, are not significant to the quality of the discretization, and only contribute to make the calculation more expensive in memory and computing time. For example, in the simulation of an electronic system of spherical nature, a cubic grid will contain $6/\pi \approx 1.9$ times more points than a spherical grid with equivalent accuracy.

A more advanced alternative is to use grids with arbitrary shapes, so that the optimal number of grid points can be used for each system. For example, in the OCTOPUS code, the default approach is to use a grid composed of the union of spheres around each atom (as shown in Figure 10.1). The complication of this approach is that the mapping between the position of the point in space and in a linear array cannot be determined analytically and must be stored in tables. However, as we discuss in Section 10.5, having the freedom of choosing an arbitrary mapping can be used to improve the cache locality for some operations.

An alternative to further reduce the number of points is to use nonuniform grids that have higher densities close to the position of the atomic nuclei, where the orbitals and the density require higher resolution. Several schemes for such adaptive grids have been proposed [27, 56–58]. Nevertheless, most real-space implementations nowadays use uniform grids. As the computational cost per point is larger for a curvilinear mesh, in our experience the actual gains in performance are small and do not justify the additional complexity in the implementation [52]. However, this is a point that should be re-evaluated in the near future, as the "operations versus memory" trade-off offered by nonuniform meshes is attractive for GPUs, especially if the pseudo-potential or PAW approximation

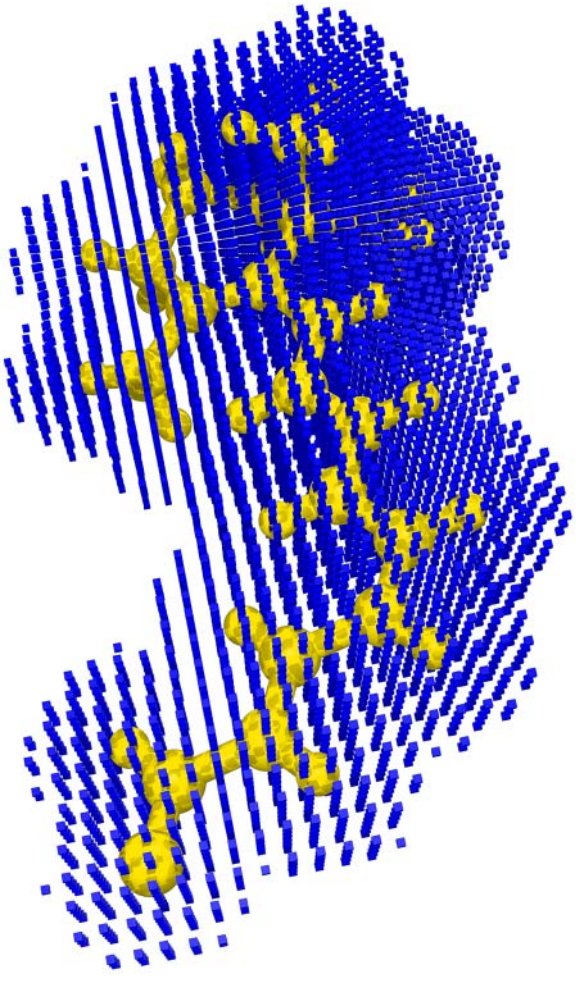


Figure 10.1 *Example of real-space grids adapted to the shape of a cis retinal molecule. The cubes mark the position of the grid points. For visualization purposes, we represent smaller and coarser grids than the ones used for actual calculations*

could be avoided altogether to obtain a simpler full-potential DFT implementation that can run more efficiently on GPUs.

10.3 Numerical Aspects of the Real-Space Approach

The basic procedure in DFT (see Chapter 3) is to calculate the ground state density of a system by solving the KS equations [54]

$$H[n]\psi_k(\boldsymbol{r}) = \epsilon_i\ \psi_k(\boldsymbol{r}) \tag{10.1a}$$

$$n(\boldsymbol{r}) = \sum_{k=1}^{N} \psi_k^*(\boldsymbol{r})\psi_k(\boldsymbol{r}), \tag{10.1b}$$

where the operator H is the KS effective single-particle Hamiltonian (atomic units are used throughout)

$$H[n] = -\frac{1}{2}\nabla^2 + v_{\text{ext}}(\boldsymbol{r}) + v_{\text{hxc}}[n](\boldsymbol{r}, t). \tag{10.2}$$

The external potential v_{ext} contains the nuclear potential and other external fields that may be present, v_{hxc} represents the electron–electron interaction and is usually divided into the Hartree term that contains the classical electrostatic interaction between electrons and the exchange and correlation (XC) potential.

To solve the KS equations, as it is standard in Hartree–Fock (HF) and DFT, a self-consistent field (SCF) iterative scheme is used [59–62]. Every SCF step we need to find the lower eigenvectors and eigenvalues of the KS Hamiltonian for a fixed electron density.

In real-space, the discretization of the KS Hamiltonian, Eq. (10.2), is achieved using a high-order finite differences representation [28]. As this results in a sparse operator representation, the diagonalization is performed using methods that do not require access to the elements of matrix representation of the Hamiltonian, but that instead apply the operator to the orbitals. Usually these eigensolvers are called iterative eigensolvers. We prefer the term sparse eigensolvers, as all eigensolvers, including the ones used for dense matrices, must be iterative as a corollary of the Abel–Ruffini theorem [63].

Several sparse eigensolvers have been proposed [64]. In this work, we use the efficient residual minimization–direct inversion in the iterative subspace (RMM-DIIS) eigensolver [65, 66]. Eigensolvers are usually preconditioned, for example, in real-space typical choices are a multigrid-based preconditioner[57] or a filter operator that removes high-frequency components [67].

The numerical operations required by most eigensolvers are the application of the KS Hamiltonian and two additional procedures that act over the whole set of orbitals: orthogonalization and subspace diagonalization [66]. Given a set of orbitals, the orthogonalization procedure performs a linear transformation that generates a new orthogonal set. Similarly, subspace diagonalization is an effective method to remove contamination between orbitals. It calculates the representation of the KS Hamiltoninan in the subspace spanned by a set of orbitals, and generates a new set where the subspace Hamiltonian is diagonal.

After the diagonalization step a new set of orbitals and a new electron density are obtained; this density is mixed with the densities from previous steps to generate a new guess density. From the new guess density, the corresponding KS effective potential needs to be calculated. Numerically, the most expensive part of this step is obtaining the Hartree potential, v_{H}, that requires the solution of the Poisson equation. The approximated XC potential, v_{xc}, also needs to be recalculated each SCF step. This potential is usually approximated by a local or semi-local expression that is evaluated directly on the grid, in fact most DFT codes do this evaluation on a real-space grid.

TDDFT [68] is the extension of DFT to describe excitations and time-dependent processes. The basic formulation of TDDFT is given by the time-dependent KS (TDKS) equation

$$i\frac{\partial}{\partial t}\psi_k(\boldsymbol{r}, t) = H(t)[n]\ \psi_k(\boldsymbol{r}, t) \tag{10.3a}$$

$$n(\boldsymbol{r}, t) = \sum_{k=1}^{N} \psi_k^*(\boldsymbol{r}, t)\psi_k(\boldsymbol{r}, t). \tag{10.3b}$$

In this case the effective KS Hamiltonian is time-dependent. This dependency comes from the external potential, that can now involve some time-dependent fields, and from the Hartree and XC potentials that depend on the time-dependent density. In the exact TDDFT formulation, the XC potential depends on the density from all previous times and on the initial conditions. However, in practice these *memory effects* are neglected and the XC potential is only considered as a function of the electronic density at the current time, this is known as the adiabatic approximation [69].

The direct solution of the TDKS leads to the method of real-time TDDFT. This is a flexible approach to model the response of an electronic system, molecular [42] or crystalline [43], to different kinds of perturbations. It is useful, for example, to calculate optical absorption spectra [42, 70], non-linear optical response [71, 72], circular dichroism [73, 74], van der Waals coefficients [75], Raman intensities [76, 77], and photoemission [78]. It is also the basis for the Ehrenfest-TDDFT molecular dynamics approach [33, 79–84].

We now detail the procedure to perform a real-time time-propagation in real-space TDDFT. Typically, the initial conditions are the electronic ground-state as obtained from a DFT calculation. The KS orbitals are then propagated under the action of a time-dependent Hamiltonian. In the case of linear optical response calculations, the external field has the form $E(\boldsymbol{r}, t) = \boldsymbol{\kappa}\delta(t)$ that can be applied as a phase to the initial KS orbitals [42, 70]. For crystalline systems, a macroscopic electric field cannot be included through the scalar potential and needs to be represented by a time-dependent vector potential [43].

There are many approaches to perform the propagation of the KS orbitals and the electron density in time [85]. In this work, we use a predictor–corrector approach to account for the nonlinearity induced by the density dependence on Eq. (10.3a), combined with an approximate enforced time-reversal symmetry (AETRS) propagator [85]. The AETRS propagator, as several other propagation methods, requires the calculation of the exponential of the KS Hamiltonian. For the time-propagation results presented in this chapter, we use a fourth order Taylor expansion of the exponential. This exponential is never calculated explicitly, but applied over orbitals. Hence, as in the case of the eigensolver, the main operation required for the time propagation is the application of the KS Hamiltonian. In this case, there is no need to orthonormalize the orbitals, since orthonormality is preserved during the time evolution. Additionally, as in the case of a ground-state calculation, each time step we need to recalculate the Hartree and XC potentials.

In the real-time framework, the observables are time-dependent quantities. For example, the induced electric field is obtained from the electric dipole moment that can be readily calculated from the electronic density. For the calculation of circular dichroism the quantity of interest is the time-dependent magnetization, that is calculated as the expectation value of the angular momentum operator [73, 74]. Frequency resolved quantities are obtained from the Fourier coefficients of the time-resolved quantities. They are traditionally obtained through a windowed Fourier transform, however, more advanced method based on the compressed sensing approach have been shown to require considerably shorter, and computationally cheaper, time propagations [86, 87].

10.4 General GPU Optimization Strategy

In this section, we discuss the general scheme that we have developed to efficiently solve the real-space DFT and TDDFT equations on GPUs. This strategy was designed taking into account the strengths and weaknesses of the current generation of GPUs, but is also effective for CPUs with vectorial floating point units.

A way to fulfill the GPU requirement of multiple independent operations is to expose data-parallelism to the low-level routines. For example, if the same operation needs to be performed over multiple data objects, the routines should receive as an argument a group of those objects, instead of operating over one object per call. For the DFT case, our GPU optimization strategy is based on the concept of blocks of KS orbitals [6]. Instead of acting over a single KS orbital at a time, performance critical routines receive a group of orbitals as argument. By operating simultaneously over several orbitals, the amount of parallelism exposed to the processor is increased considerably. We illustrate this idea in Figure 10.2. This approach has been adopted by other real-space GPU implementations [16].

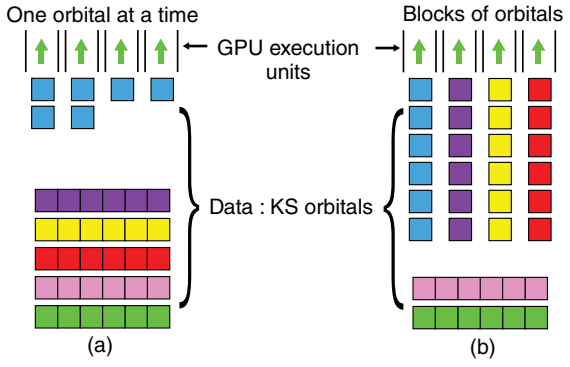


Figure 10.2 *Scheme illustrating the blocks of orbitals strategy for DFT on GPUs. (a) Operating on a single orbital might not provide enough parallelism for the GPU to perform efficiently. (b) By operating simultaneously over several orbitals there is a larger degree of data parallelism and there is less divergence among GPU threads. Reproduced from Ref. [8]*

The blocks-of-orbitals strategy has an additional advantage: in a GPU, threads are divided in groups of 32 (Nvidia) or 64 (AMD), called warps or wavefronts; for efficient execution all threads in a warp must execute exactly the same instruction sequence, otherwise execution will be performed in serial fashion. Since for most operations in DFT or TDDFT the same instructions have to be performed for each orbital, we can assign operations corresponding to different orbitals to different threads in a warp. This ensures that the execution within each warp is regular, without divergences in the instruction sequence. In a CPU, the vectorial floating point units play a similar role as warps, with the difference that groups are typically much smaller (2, 4, or 8 elements).

A possible drawback of the block-of-orbitals approach is that memory access issues might appear, as working with larger amount of data can saturate caches and reduce their ability to speed-up memory access. This is especially true for CPUs, which rely more on caches, than GPUs. Larger blocks can also increase the amount of memory required for temporary data. Consequently, using blocks that are too large can be detrimental for performance.

In our implementation the number of orbitals in a block, or block-size, is variable and controlled at execution time. Ideally the block-size should be an integer multiple of the warp-size. This might not be possible if not enough orbitals are available, in such a case the block-size should be a divisor of the warp size. Following these considerations we restrict our block-size to be a small power of 2[1].

The way blocks of orbitals are stored in memory is also fundamental for optimal performance. A natural scheme would be to store the coefficients for each orbital contiguously in memory, so that each orbital in a block can be easily accessed as an individual object. However, memory access is usually more efficient when threads access adjacent memory locations as loads or stores go to the same cache-lines. Since in our approach consecutive threads are assigned to different orbitals, we order blocks by the orbital index first and then by the discretized $\boldsymbol{r}$-index, ensuring that adjacent threads will access adjacent memory locations.

10.5 Kohn–Sham Hamiltonian

For both DFT and real-time TDDFT in real-space, the application of the KS Hamiltonian, Eq. (10.2), over trial orbitals is the fundamental operation. As such, it was our first target for efficient GPU

[1] This has the additional advantage that the integer multiplication by the block-size, required for array address calculations, can be done using the cheaper bit-shift instructions.

execution. This is also the case for other methods that are beyond the scope of this article, like on-the-fly molecular dynamics [80, 88], and some linear response approaches [89, 90].

As a matrix, the real-space KS Hamiltonian operator is sparse, with a number of coefficients that is proportional to the number of grid points. While the matrix could be stored in a sparse form, it is not convenient to do so. It is more efficient to use it in operator form, with different terms that are applied independently: the kinetic energy operator, and the local and nonlocal potentials.

In real-space the kinetic-energy operator is approximated by a high-order finite-differences expansion of the Laplacian operator [28]. Numerically, this is a stencil calculation, where the value at each point is calculated as a linear combination of the neighboring-point values. In the simulations presented in this chapter a fourth-order approximation was used, this results in a stencil of size 25.

Since stencil calculations are common in scientific and engineering applications, their optimization on CPU and GPU architectures has received considerable interest [91–97]. By applying the operator over several orbitals at once in our approach, we avoid some of the performance issues commonly associated with stencil calculations, in particular with respect to vectorization [96].

Memory access is usually the limiting factor for the performance of the finite-difference operators [92], since for each point we need to iterate over the stencil loading values that are only used for one multiplication and one addition. One issue is that, as the values of the neighbors are scattered, memory access is not regular. This is solved by using blocks of orbitals: since the Laplacian is calculated over a group of orbitals at a time, for each point of the stencil we load several values, one per orbital in the block, that are contiguous in memory. This makes memory access more regular and hence more efficient for both GPUs and CPUs.

Still, a potential problem with memory access persists. As each input value of the stencil has to be loaded several times, ideally input values should be loaded from main memory once and kept in cache for subsequent uses. Unfortunately, as the stencil operation has poor memory locality, this is not always the case.

We devised an approach to improve cache utilization by controlling how the three-dimensional grid is mapped to a linear array, that is, how grid points are ordered in memory. The standard approach is to use a row-major or column-major order which leads to some neighboring points in the grid being allocated in distant memory locations (Figure 10.3a). We have tested two different approaches that permits close spatial regions to be stored closer in memory, improving memory locality for the Laplacian operator. The first one is to enumerate the grid points based on a sequence of small tiles, or *bricks* in three dimensions [7], as shown in the example of Figure 10.3b. The problem with this approach is that the optimal size of the bricks depends on the cache size, and the shape and size of the grid, which change for each molecule. Since it is not practical to optimize these parameters for each calculation, we need to use a fixed set that does not always yield the best possible performance. To avoid this problem, we have developed a second approach based on using a Hilbert space-filling curve [98–100], as shown in Figure 10.3c.

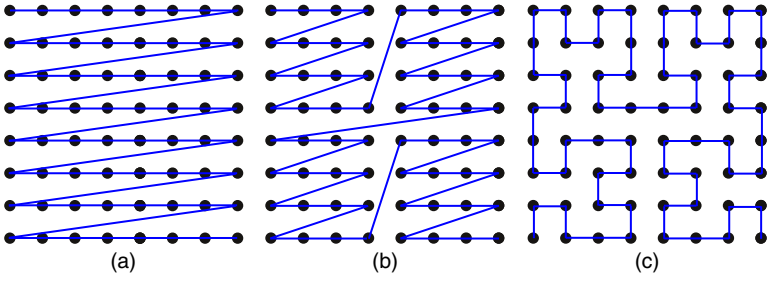


Figure 10.3 *Examples of different grid orders in 2D: (a) standard order (b) grid ordered by small parallelepipedic subgrids or* bricks*, and (c) order given by a Hilbert space-filling curve*

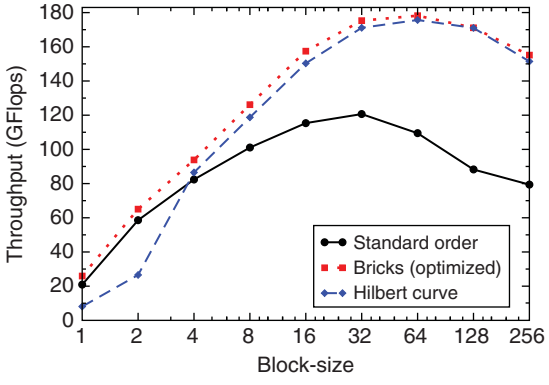


Figure 10.4 *Effect of the optimization of the grid mapping for data locality in the numerical throughput of the Laplacian operator as a function of the size of the orbitals block. Spherical grid with ∼ 500k points. Computations with a AMD Radeon 7970 GPU*

In Figure 10.4, we compare the throughput of the Laplacian operator on an AMD GPU, as a function of the block-size, for the different strategies to order the grid: the standard grid order, the *bricks* strategy with parameters optimized for each calculation and the Hilbert space-filling curve. The are several conclusions we can draw from the figure. The first one is that even for the standard grid order, there is a significant improvement in performance associated with the use of blocks of orbitals: the throughput for a block-size of 32 is around 6× the one for block-size 1, which represents working with one orbital at a time. When we consider the *bricks* strategy, an additional 50% increase in performance is obtained, for a total gain of 9× with respect to the basic implementation. The position of the maximum throughput is also shifted from blocksize 32 to 64, this is expected, since a better cache locality decreases the impact of working with larger data sets.

Finally, the Hilbert grid-order achieves a performance that is close to that of the bricks approach, but without requiring optimized parameters. The difference in performance, in particular for small block size, can be understood when considering how neighboring points are addressed. For the application of the Laplacian in an arbitrarily shaped grid, a table with the location of neighboring points is required. In the case of standard- and brick-ordered grids, this table can be easily compressed since the relative distance of the neighbors does not usually change between points [101]. For the Hilbert-ordered grid, we do not have a strategy to compress this table and a full table is required. For small blocks, the access to this table dominates ad constitutes a large performance overhead that hits the performance. As the block of orbitals becomes larger, the relative cost of loading the table of neighbors decreases.

The second term in the Hamiltonian is the scalar potential that includes the contributions from the external potential, including the ionic pseudo-potentials, the Hartree, exchange, and correlation potentials. This is applied in two parts: the local one and the non-local one that comes from the pseudo-potentials [102, 103].

The application of the local potential has very little arithmetic intensity and is heavily limited by memory access. By using blocks of orbitals a larger number of simultaneous operations can hide the memory access latency, and the values of the potential are reused, reducing the number of memory accesses. See Ref. [8] for details.

The nonlocal part of the potential appears as each angular momentum component of an orbital needs to see a different pseudo-potential. In practice, we calculate

$$V_{\rm nl}\psi_k(\boldsymbol{r}) = \sum_A \sum_{lm} \gamma^A_{lm}(\boldsymbol{r}-\boldsymbol{R}) \int_{r'<r_c} \mathrm{d}\boldsymbol{r}'\ \gamma^A_{lm}(\boldsymbol{r}'-\boldsymbol{R}_A)\psi_k(\boldsymbol{r}'), \tag{10.4}$$

where γ_{lm}^{A} corresponds to the pseudo-potential projectors for atom A, and l and m are the angular momentum components that go from 0 to a certain $l_{\max}$, usually 3, and from $-l$ to l, respectively. The projector functions are localized over a sphere, such that $\gamma_{lm}^{A}(\boldsymbol{r}) = 0$ for $|\boldsymbol{r}| > r_{\mathrm{c}}$.

In our implementation, Eq. (10.4) is calculated in two parts that are parallelized differently on the GPU. The first part is to calculate the integrals over $\boldsymbol{r}'$ and store the results. This calculation is parallelized for a block of orbitals, angular-momentum components and all atoms, with each GPU-thread calculating an integral.

The second part of the application of the nonlocal potential is to multiply the stored integrals by the radial functions and sum over angular-momentum components. In this case, the calculation can be parallelized over orbitals, and, if the pseudo-potential spheres associated with each atom do not overlap, it can also be parallelized over the $\boldsymbol{r}$-index and atoms. For most systems these spheres do not overlap, but if they do, race conditions would appear as several threads would try to update the same point. In order to perform the calculations in parallel, essential for performance on the GPU, we divide the atoms in groups that have nonoverlapping spheres as shown in Figure 10.5. Then, we parallelize over all atoms in each group.

In Figure 10.6, we plot the throughput obtained by the non-local potential implementation for a β-cyclodextrin molecule. The Nvidia card shows good performance, 46 GFlops, only when large blocks of orbitals are used. The AMD card has a similar behavior, but the performance is much lower, with a maximum of 11 GFlops. This is a clear example of how our approach is an effective way of increasing the performance that can be obtained from the GPU. As this is a complex routine, and our current implementation is very basic, we suspect that a more sophisticated and optimized version could significantly increase the numerical performance of this part of the application of the KS Hamiltonian, in particular for the AMD GPU.

The total performance of the application of the KS Hamiltonian is shown in Figure 10.7. This combines the finite-difference Laplacian, as well as the local and nonlocal parts of the potential.

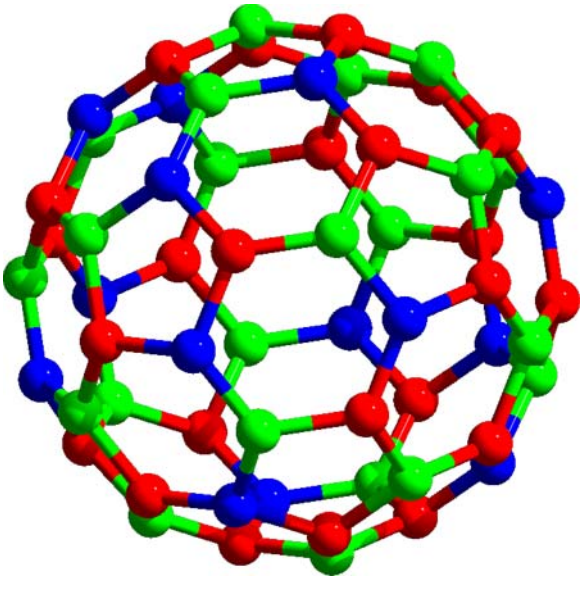


Figure 10.5 *Division of the atoms of a C_{60} molecule in groups (represented by different colors) whose pseudo-potential spheres do not overlap. Reproduced from Ref. [8]*

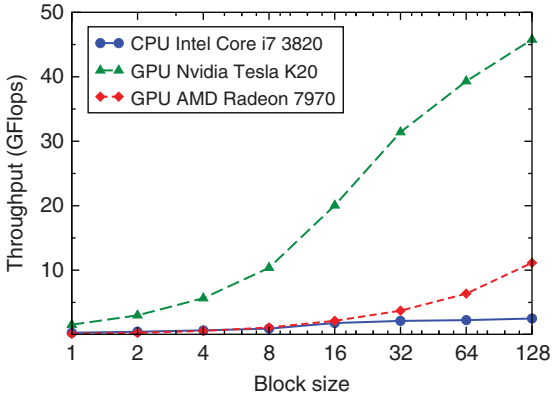


Figure 10.6 *Numerical throughput of the application of the pseudo-potentials nonlocal part as a function of the size of the block of orbitals (block-size). Calculation for β-cyclodextrin with 256 orbitals and 260k grid points for one CPU and two GPUs. Reproduced from Ref. [8]*

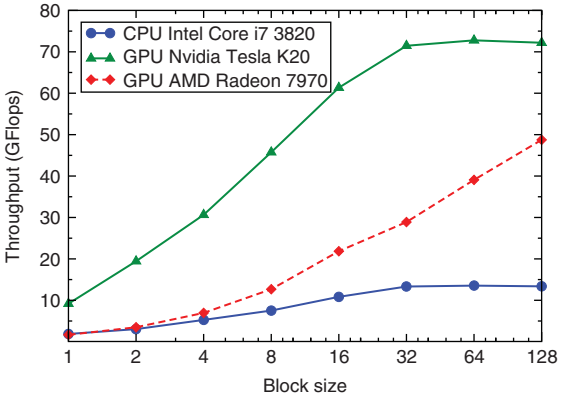


Figure 10.7 *Numerical throughput of the application of the Kohn–Sham Hamiltonian as a function of the size of the block of orbitals (block-size). Calculation for β-cyclodextrin with 256 orbitals and 260k grid points for one CPU and two GPUs*

10.6 Orthogonalization and Subspace Diagonalization

The orthogonalization and subspace diagonalization procedures are required by eigensolvers to ensure that the eigenvectors are orthogonal and to remove any possible cross-contamination between them. These operations mix different orbitals, so they cannot be directly implemented using blocks of orbitals. In our approach, we copy the orbitals to an array where all the coefficients corresponding to different orbitals are contiguous in memory [8]. To avoid allocating a full copy of all the orbitals, we perform the operation for a set of points at a time. Effectively, we are switching from a block-of-orbitals representations to a block-of-points approach.

The orthogonalization and subspace diagonalization can be efficiently implemented in terms of dense linear algebra operations [8, 66]. Additionally, the subspace diagonalization requires the application of the KS Hamiltonian. For CPUs BLAS and LAPACK provide an efficient and portable set of linear algebra routines. For GPUs, we use the OpenCL BLAS implementation provided by AMD as

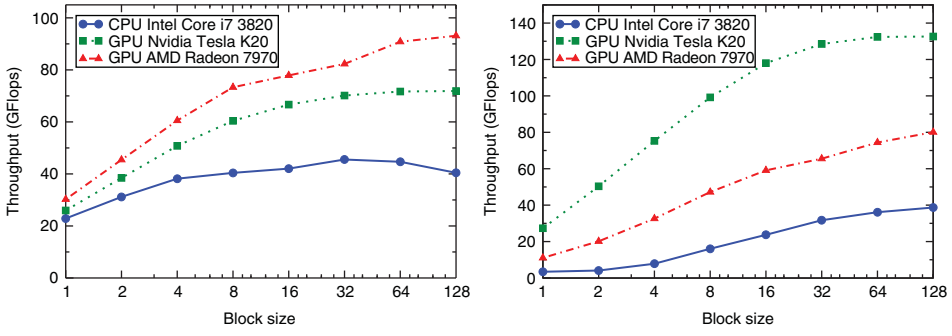


Figure 10.8 *Numerical throughput of (a) the orthogonalization procedure and (b) the subspace diagonalization as a function of the size of the block of orbitals (block-size) for different processors. Calculation for β-cyclodextrin with 256 orbitals and 260k grid points*

part of the Accelerated Parallel Processing Math Libraries (APPML), while the routines provided by Lapack, the Cholesky decomposition [104], and the dense-matrix diagonalization, are performed on the CPU. This does not affect the performance in our current implementation, but to study larger systems it might be necessary to use the GPU accelerated Lapack routines provided by the Magma library [105].

In Figure 10.8a, we show the performance obtained for our implementation of the orthogonalization procedure. The GPU speed-up is not very large with respect to the CPU. As this procedure is based on linear algebra operations, we attribute the poor speed-up to differences in the linear algebra libraries. Figure 10.8b shows the performance obtained for the subspace diagonalization. In this case the GPU speed-up is larger than for the orthogonalization case, probably because this routine is based on our implementation of the KS Hamiltonian, and on matrix–matrix multiplications, that in general are simpler to optimize and parallelize than other linear algebra operations.

10.7 Exponentiation

To solve the TDKS equations we use the AETRS method. In this approach, the orbitals are propagated according to

$$\psi_k(\boldsymbol{r}, t + \Delta t) = e^{-i\Delta t\ H(t+\Delta t)} e^{-i\Delta t\ H(t)} \psi_k(\boldsymbol{r}, t), \tag{10.5}$$

where the KS Hamiltonian at time $t + \Delta t$ is obtained from an interpolation from previous times [71]. Each iteration of this propagator requires the application of two exponentials of the Hamiltonian over the orbitals. In this work, we calculate this exponential using a fourth-order truncated Taylor expansion. This approximation only requires the application of the KS Hamiltonian and some basic operations like multiplication by a scalar and addition of orbitals.

While in real-time TDDFT the orbitals are complex, most operations, including the KS Hamiltonian, do not mix the real and imaginary part. This means that a block of complex orbitals can be simply considered as a block of real orbitals of twice the size. Just a few operations need to be aware of the complex nature of the orbitals and operate by pairs on the orbitals of a block.

In Figure 10.9, we plot the performance obtained for our implementation of the exponential of the Hamiltonian. Since in this case the orbitals are complex, the smallest possible block-size is 2 and the largest one is 512 (twice the number of orbitals used in the calculation). Both GPU models achieve more than 60 GFlops of processing throughput, which represents a speed-up of more than 5× with respect to the CPU.

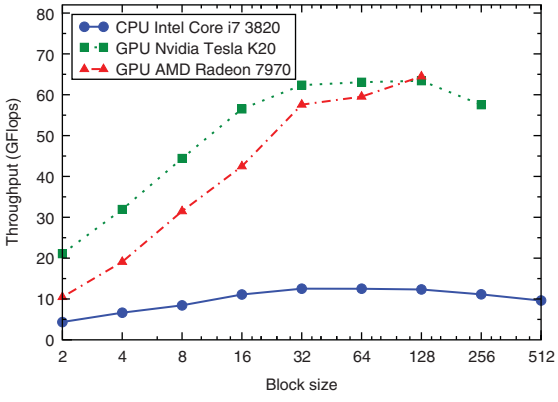


Figure 10.9 *Numerical throughput of the fourth-order Taylor approximation to the exponential operator as a function of the size of the block of orbitals (block-size). Calculation for β-cyclodextrin with 256 orbitals and 260k grid points for one CPU and two GPUs*

10.8 The Hartree Potential

For ground-state and real-time calculations, another operation that we execute on the GPU is the calculation of the Hartree potential by solving the Poisson equation

$$\nabla^2 v_{\mathrm{H}}(\boldsymbol{r}) = -4\pi\, n(\boldsymbol{r}). \tag{10.6}$$

This is an equation that appears in many physical contexts. For example, in electronic structure it is used in the calculation of approximations to the exchange term [106–108], in the calculation of integrals that appear in Hartree–Fock or Casida theories [109], or to impose electrostatic boundary conditions [110–115].

Many methods have been proposed to solve the Poisson equation in linear or quasi-linear time [116–121]. In our GPU implementation, we use an approach based on fast Fourier transforms (FFTs), as it is quite efficient and simple to implement. While FFTs impose periodic boundary conditions, by enlarging the FFT grid and using a modified interaction kernel we can find the solution for the free boundary problem [122], which is the relevant one for finite electronic systems.

The FFT solution of the Poisson equation is based on applying the Coulomb interaction kernel in Fourier space, where it is local. For each solution, a forward and a backward FFT are required, these are performed using the clAMDFft library. For CPUs we use the multi-threaded FFTW library [123]. Since there is a single Poisson equation to solve in the Kohn–Sham approach, independently of the number of electrons, we cannot use the block approach in this case. However, we plan to develop a GPU accelerated blocked Poisson solver that could be applied to problems like Casida linear-response TDDFT, or Hartee–Fock and hybrid functional calculations [124].

In Figure 10.10, we show the performance of our GPU based Poisson solver for different system sizes. For the AMD card, the GPU version outperforms the CPU version, in some cases by a factor of 4×. For the Nvidia GPU the performance is smaller and less consistent, possibly because the library has not been explicitly optimized for this GPU. The step structure seen on the plots is caused by the fact that FFTs cannot be performed efficiently over grids of any size: the grid dimension in each direction must be a product of certain values, or radices, that are determined by the implementation. If a grid dimension is not valid, the size of the grid has to be increased. Since the CPU implementation is more mature and supports more radices, the steps are smaller than the GPU implementation.

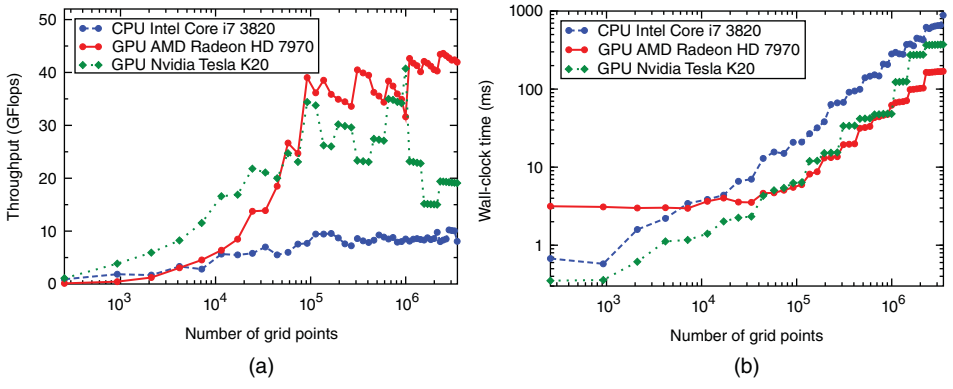


Figure 10.10 *Comparison of (a) the throughput and (b) calculation time achieved by our FFT Poisson solver as a function of the number of grid points for one CPU and two GPUs. The data is originally on main memory, so the time required to copy the input data to the GPU and copy back the result is included. The number of points corresponds to the spherical grid used by* OCTOPUS*, the FFT grid has a larger number of points. Following Ref. [123], the operation count for the FFTs is assumed to be* $5N\log_2 N$

10.9 Other Operations

There are several simpler operations that also need to be performed on the GPU, beyond the ones we have discussed in previous sections. These operations include basic operations between orbitals, like copies, linear combinations, and dot products. All of these procedures are implemented on the GPU using the block-of-orbitals approach. In fact, we have found that it is necessary to pay attention to the parallelization and optimization of most of the operations executed on the GPU, as a single routine that is not properly optimized can affect the numerical performance of the entire code considerably.

In our current implementation, there are two procedures that are still done by the CPU, as they would require a considerable effort to implement on the GPU. This will the focus of future work.

The first operation is the evaluation of the XC energy and potential. This is a local operation that is straightforward to evaluate in parallel and should perform well on the GPU. The problem is that there are a large number of XC approximations, each one involving complex formulas [125] that would need to be implemented on the GPU. For the ground-state the evaluation of the XC term does not greatly influence the computational time when done on the CPU. However, for the time propagation each time step involves less computational work than for the SCF and the cost of the CPU evaluation of the XC potential affects the GPU performance. For this reason, the time-propagation results presented in the chapter use the Teter 93 local density approximation (LDA) [126]. This parametrization is based on Padé approximation for both the exchange and correlation energy so it is cheap to evaluate.

The second procedure that is executed on the CPU is that of the evaluation of the atomic orbitals and pseudo-potentials. These are required to construct the atomic potentials on the grid and for the initialization of SCF by a linear combination of the atomic orbitals. The reason is that we use a spline interpolation to transfer the orbitals to the grid, which depends on the GSL library [127] that is not available on the GPU. For the results presented here, this fact does not influence performance, but it is important for Ehrenfest molecular dynamics runs, as the atomic potential needs to be reevaluated each time step.

10.10 Numerical Performance

In this section, we evaluate the numerical performance of our implementation for ground-state and real-time calculations. For this analysis we use several measurements: the throughput, the wall-clock calculation time, the speed-up with respect to the CPU implementation, and the comparison with a second GPU-based DFT approach. We present these different parameters since they are required to paint a clear picture of the performance of a GPU implementation.

Unfortunately, it is commonplace in articles presenting GPU implementations to restrict the performance analysis to the speed-up with respect to a CPU implementation of the same problem. Moreover these comparisons sometimes present very large speed-ups that are not realistic if we consider that the maximum speed-up, the peak-performance ratio between the GPU and the CPU, is approximately 10×. If performance is limited by the memory bandwidth, then the maximum speed-up is reduced to 6×. The issue with these comparisons is usually the CPU code that is taken as reference. A proper speed-up calculation should use similar optimization strategies on the CPU and the GPU, and in both cases it should be parallelized to use all the execution units available on each processor, unfortunately this is not always the case. For example, a typical approach that can misrepresent performance gains is to calculate the speed-up of a multi-GPU implementation with respect to an equivalent number of *CPU cores*. Given that the price and power consumption of a single CPU core is typically not directly comparable with that of a GPU, and that modern CPUs can contain up to 16 cores, this is clearly not a representative measure of the GPU speed-up.

We first analyze how effective the block of orbitals approach is to increase the performance of a full GPU calculation. In Figure 10.11 we plot, for the β-cyclodextrin molecule, the numerical throughput for the ground-state and real-time calculations as a function of the block-size. We can see that in both cases there is an important gain in performance associated with using a block of orbitals with respect to working with a single orbital at a time (the block-size 1 case). For example, for the ground-state calculation the speed-up for the CPU is approximately 3.4×, while for the AMD and Nvidia GPUs the speed-up is 15× and 7.7×, respectively. Since our implementation is not optimized for the block-size 1 case there is an overhead associated to the blocked implementation. On the CPU, we can compare with our older nonblocked implementation, this results in a still significant speed-up of 2.5×.

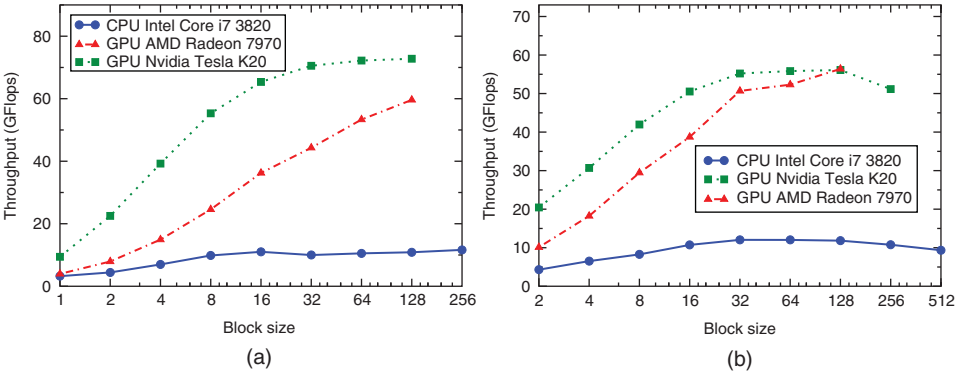


Figure 10.11 *Numerical throughput of our CPU and GPU implementations as a function of the size of the block of orbitals (block-size). (a) Self-consistency cycle in a ground-state DFT calculation. (b) Real-time TDDFT propagation. β-cyclodextrin molecule with 256 orbitals and 260k grid points*

We now focus our attention on the performance of our GPU implementation for DFT and TDDFT calculations on molecules of different sizes. For this we use the set of 40 molecules listed in Ref. [8]. In Figures 10.12 and 10.13, we show, for the molecules in our set, the performance measured as throughput and total computational time as a function of the number of electrons. For ground-state DFT we plot the total computational time while for the real-time TDDFT calculation the time required to propagate an interval of 1 attosecond. As expected, the computational time tends to increase with the number of electrons, but there is a strong variation from system to system. This variation is mainly explained by the physical size of each molecule, that determines the size of the grid that is used in the simulation. For ground-state calculation the number of self-consistency iterations also changes from one system to the other, affecting the total calculation time.

As the size of the system increases, the GPU becomes more efficient, with a maximum throughput of 90 GFlops for DFT and 56 GFlops for TDDFT. This effect can also be seen in Figure 10.14, where we plot the speed-up with respect to the CPU for both types of calculations. If we compare the DFT and real-time TDDFT calculations, we can see that the efficiency of the ground-state calculations depends more on the size of the systems, a possible explanation for this effect is that real-time

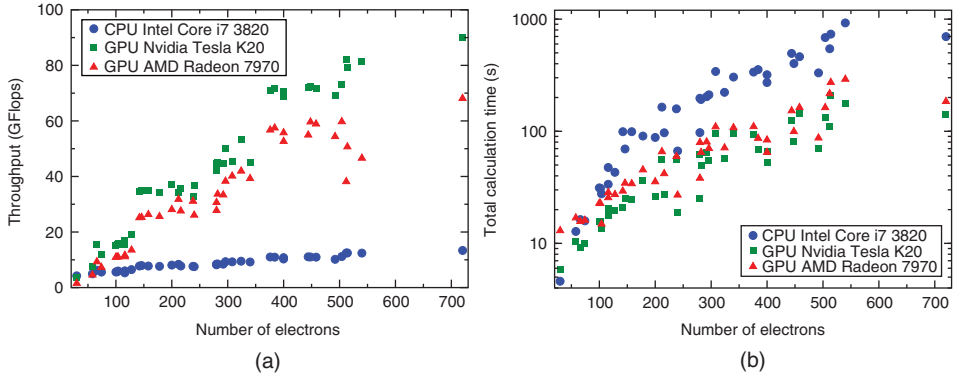


Figure 10.12 *Performance of our CPU and GPU implementations for a set of 40 molecules of different sizes. (a) Numerical throughput of the self-consistency cycle. (b) Total execution time for a single-point energy calculation. The set of molecules is taken from Ref. [8]*

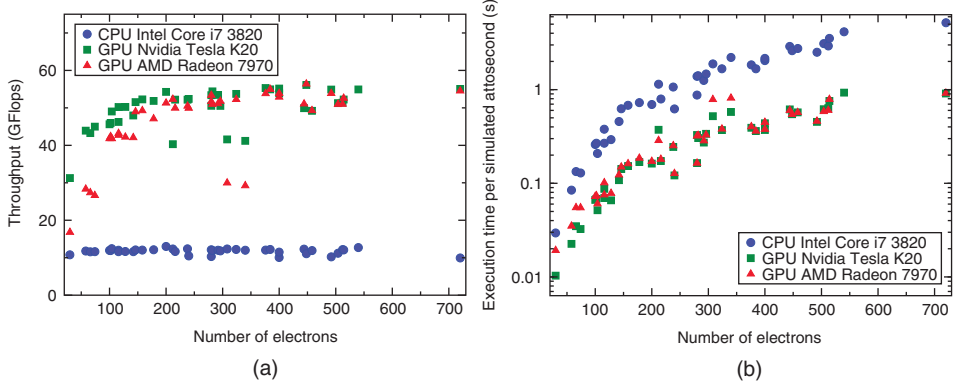


Figure 10.13 *Performance of our CPU and GPU real-time TDDFT implementations for a set of 40 molecules of different sizes. (a) Numerical throughput of the real-time propagation. (b) Computational time required to propagate 1 attosecond. The set of molecules is taken from Ref. [8]*

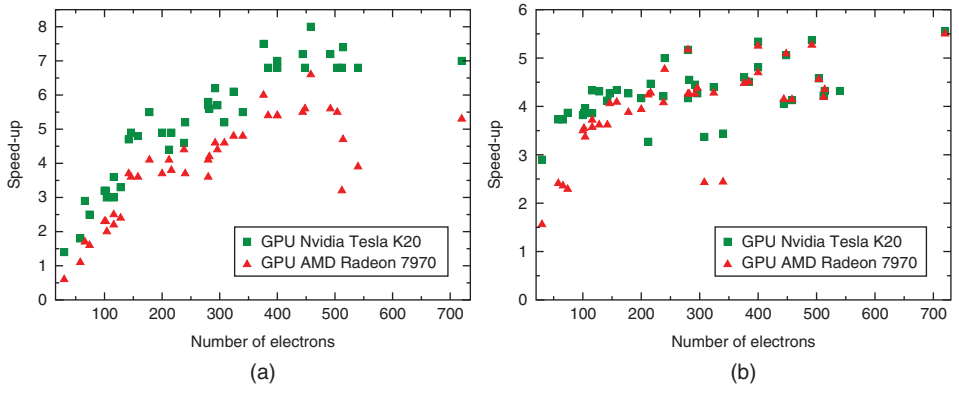


Figure 10.14 *Speed-up of the GPU calculation with the respect to the CPU for different molecules as a function of the number of valence electrons. (a) Speed-up for the time spent in the SCF-cycle of a ground-state DFT calculation (without considering initializations). (b) Speed-up for real-time TDDFT. Intel Core i7 3820 using 8 threads*

calculations use complex orbitals, that for our approach allow blocks of orbitals that are twice as large as in the ground-state calculations. On the other hand, the ground-state calculation reaches a higher throughput and speed-up than the real-time calculations: the maximum GPU/CPU speed-up is 8.0× for DFT versus 5.6× for TDDFT for the Nvidia GPU and 6.6× against 5.5× for AMD. We think this is explained by subspace diagonalization, that is only used for the ground-state calculations that, as shown in Figure 10.8, can reach a throughput in excess of 100 GFlops on the Nvidia card.

To conclude our performance evaluation, we show the comparison we made in Ref. [8] between our ground-state DFT implementation and the TERACHEM code [21, 22, 25, 26, 128]. TERACHEM (Chapter 4) uses GTOs as a basis for the expansion of the molecular orbitals: the traditional approach used in quantum chemistry. Since OCTOPUS and TERACHEM use very different simulation techniques, we took great care in selecting simulation parameters that produce a similar level of discretization error. The timings for both codes are compared in Figure 10.15, we show the comparison between absolute times and also the relative performance between the two DFT implementations. We can see

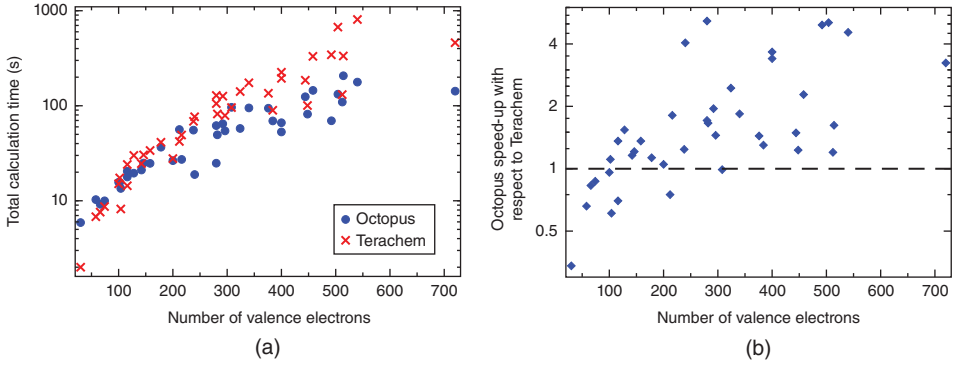


Figure 10.15 *Numerical performance comparison between our GPU implementation (OCTOPUS) and the TERACHEM code. (a) Comparison of the total calculation time as a function of the number of valence electrons. (b) Speed-up of our implementation with respect to TERACHEM (run time of TERACHEM divided by the run time of OCTOPUS). The calculations are single-point energy evaluations performed on a set of 40 molecules, running on a Nvidia Tesla K20 GPU. Reproduced from Ref. [8]*

that TERACHEM tends to be faster for smaller systems, while OCTOPUS has an advantage for systems with more than 100 electrons. It is difficult to generalize these results due to the different simulation approaches and their different strengths and weaknesses. For example, our current implementation will certainly be much slower than TERACHEM for hybrid HF-DFT XC approximations [124] due to the cost of applying an exact-exchange operator in real-space. However, we can conclude that for pure DFT calculations the real-space method can compete with the Gaussian approach, and can outperform it for some systems.

10.11 Conclusions

In this chapter we have discussed the implementation of real-space DFT and TDDFT on GPUs. The development of simulation strategies for GPUs involves much more than rewriting code in GPU language. Real-space DFT is not the exception and what we have presented is a scheme designed to perform DFT calculations efficiently on massively parallel processors. The approach is based on using blocks of KS orbitals as the basic data object. This provides the GPU with the data parallelism required to perform efficiently, which would be harder to achieve by working on single orbitals at a time. Many of these techniques are applicable to other DFT discretization approaches, especially those based on a sparse representation like plane-waves (see Chapter 7) or wavelets [129, 130] (see Chapter 6).

We presented results for several benchmark simulations in order to give a full picture of the performance of our GPU and CPU implementations. We achieve a considerable throughput and speed-up with respect to the CPU version of the code. More importantly, by comparing with a GPU-accelerated implementation of DFT based on Gaussian basis sets, we find that calculation times are similar, with our code being faster for several of the systems that were tested. The advantage of our approach is that we only require a small number of simple kernels (functions that are executed on the GPU) that can be efficiently optimized for regular execution and memory access.

The results show that the real-space formulation provides a good framework for the implementation of DFT on GPUs, making real-space DFT an interesting alternative for electronic structure calculations that offers good performance, systematic control of the discretization, and the flexibility to study many classes of systems, including both periodic and finite systems. Moreover real-space DFT is suitable for large scale parallelization in distributed memory systems with tens of thousands of processors [7, 49, 50].

There are, however, several issues that need to be addressed in order to make the real-space approach a competitive framework for all types of electric structure methods. One of the main limitations is the poor performance of the real-space implementations of the exact-exchange operator used by hybrid XC functionals. Many wave-function methods [62] are also prohibitively expensive in real-space, but recently proposed real-space stochastic methods [131–133] might provide an interesting way forward.

10.12 Computational Methods

Our numerical implementation is included in the OCTOPUS code [7, 33, 52] and it is publicly available under the GPL free-software license [134]. The calculations were performed with the development version (*octopus superciliosus*, svn revision 11531). GPU support is also available in the 4.1 release of OCTOPUS.

Since OCTOPUS is written in Fortran 95, we wrote a wrapper library to call OpenCL from that language. This library is called FORTRANCL and it is available as a standalone package under a free-software license [135].

All calculations were performed using the default pseudo-potentials of OCTOPUS, filtered to remove high-frequency components [136]. The grid for all simulation is a union of spheres of radius 5.5 Bohr around each atom with a uniform spacing of 0.41 Bohr. For ground-state calculations we used the BLYP XC functional [137–139] and for real-time TDDFT we used the LDA in the Teter 93 parametrization [126].

The GTO calculations were done with TERACHEM (version v1.5K) with the *6-311g** basis and `dftgrid = 1`. All other simulation parameters were kept in its default values.

The system used for the tests has an Intel Core i7 3820 CPU, which has four cores running at 3.6 GHz that can execute two threads each. The CPU has a quad-channel memory subsystem with 16 GiB of RAM running at 1600 MHz. The GPUs are an AMD Radeon HD 7970 with 3 GiB of RAM and Nvidia Tesla K20c with 5 GiB (ECC is disabled, as the other processors do not support ECC). Both GPUs are connected to a PCIe 16× slot, the AMD card supports the PCIe 3 protocol while the Nvidia card is limited to PCIe 2. OCTOPUS was compiled with the GNU compiler (gcc and gfortran, version 4.7.2) with AVX vectorization enabled. For finite-difference operations, CPU vectorization is implemented explicitly using compiler directives. We use the Intel MKL (version 10.3.6) implementation of BLAS and LAPACK that is optimized for AVX. We use the OpenCL implementation from the respective GPU vendor: the AMD OpenCL version is 1084.4 (VM) and the Nvidia one is 310.32 (OpenCL is not used for the CPU calculations). All tests are executed with 8 OpenMP threads.

Total and partial execution times were measured using the `gettimeofday` call. The throughput is defined as the number of floating point additions and multiplications per unit of time. The number of operations for each procedure is counted by inspection of the code. The operation count for FFTs is assumed to be $5N\log_2 N$[123]. For TERACHEM the total execution time is obtained from the program output.

Acknowledgments

We would like to acknowledge Nvidia for support via the Harvard CUDA Center of Excellence, and both Nvidia and Advanced Micro Devices (AMD) for providing the GPUs used in this work. This work was supported by the Defense Threat Reduction Agency under Contract No. HDTRA1-10-1-0046. We thank the generous support of the FAS Science Division Research Computing Group at Harvard University.

References

1. Yasuda, K. (2008) Accelerating density functional calculations with graphics processing unit. *J. Chem. Theory Comput.*, **4**, 1230–1236.
2. Vogt, L., Olivares-Amaya, R., Kermes, S., Shao, Y., Amador-Bedolla, C. and Aspuru-Guzik, A. (2008) Accelerating resolution-of-the-identity second-order Moller-Plesset quantum chemistry calculations with graphical processing units. *J. Phys. Chem. A*, **112**, 2049–2057.
3. Genovese, L., Ospici, M., Deutsch, T., Méhaut, J.-F., Neelov, A. and Goedecker, S. (2009) Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *J. Chem. Phys.*, **131**, 034103 (8 pages).

4. Watson, M., Olivares-Amaya, R., Edgar, R.G. and Aspuru-Guzik, A. (2010) Accelerating correlated quantum chemistry calculations using graphical processing units. *Comput. Sci. Eng.*, **12**, 40–51.
5. Tomono, H., Aoki, M., Iitaka, T. and Tsumuraya, K. (2010) GPU based acceleration of first principles calculation. *J. Phys. Conf. Ser.*, **215**, 012121.
6. Andrade, X. and Genovese, L. (2012) Harnessing the power of graphic processing units, in *Fundamentals of Time-Dependent Density Functional Theory*, Lecture Notes in Physics, vol. **837** (eds M.A. Marques, N.T. Maitra, F.M. Nogueira, E. Gross and A. Rubio), Springer-Verlag, Berlin, Heidelberg, pp. 401–413.
7. Andrade, X., Alberdi-Rodriguez, J., Strubbe, D.A., Oliveira, M.J., Nogueira, F., Castro, A., Muguerza, J., Arruabarrena, A., Louie, S.G., Aspuru-Guzik, A., Rubio, A. and Marques, M.A.L. (2012) Time-dependent density-functional theory in massively parallel computer architectures: the octopus project. *J. Phys. Condens. Matter*, **24**, 233202.
8. Andrade, X. and Aspuru-Guzik, A. (2013) Real-space density functional theory on graphical processing units: computational approach and comparison to Gaussian basis set methods. *J. Chem. Theory Comput.*, **9**, 4360–4373.
9. Maintz, S., Eck, B. and Dronskowski, R. (2011) Speeding up plane-wave electronic-structure calculations using graphics-processing units. *Comput. Phys. Commun.*, **182**, 1421–1427.
10. DePrince, A.E. and Hammond, J.R. (2011) Coupled cluster theory on graphics processing units I. The coupled cluster doubles method. *J. Chem. Theory Comput.*, **7**, 1287–1295.
11. Asadchev, A. and Gordon, M.S. (2012) New multithreaded hybrid CPU/GPU approach to Hartree–Fock. *J. Chem. Theory Comput.*, **8**, 4166–4176.
12. Spiga, F. and Girotto, I. (2012) phiGEMM: a CPU-GPU library for porting quantum ESPRESSO on hybrid systems. Parallel, Distributed and Network-Based Processing (PDP), 20th Euromicro International Conference, pp. 368–375.
13. Maia, J.D.C., Urquiza Carvalho, G.A., Mangueira, C.P., Santana, S.R., Cabral, L.A.F. and Rocha, G.B. (2012) GPU linear algebra libraries and GPGPU programming for accelerating MOPAC semiempirical quantum chemistry calculations. *J. Chem. Theory Comput.*, **8**, 3072–3081.
14. Hacene, M., Anciaux-Sedrakian, A., Rozanska, X., Klahr, D., Guignon, T. and Fleurat-Lessard, P. (2012) Accelerating VASP electronic structure calculations using graphic processing units. *J. Comput. Chem.*, **33**, 2581–2589.
15. Esler, K., Kim, J., Ceperley, D.M. and Shulenburger, L. (2012) Accelerating quantum Monte Carlo simulations of real materials on GPU clusters. *Comput. Sci. Eng.*, **14**, 40–51.
16. Hakala, S., Havu, V., Enkovaara, J. and Nieminen, R. (2013) Parallel electronic structure calculations using multiple graphics processing units (GPUs), in *Applied Parallel and Scientific Computing*, Lecture Notes in Computer Science, vol. **7782** (eds P. Manninen and P. Oster), Springer-Verlag, Berlin, Heidelberg, pp. 63–76.
17. Jia, W., Cao, Z., Wang, L., Fu, J., Chi, X., Gao, W. and Wang, L.-W. (2013) The analysis of a plane wave pseudopotential density functional theory code on a GPU machine. *Comput. Phys. Commun.*, **184**, 9–18.
18. Jia, W., Fu, J., Cao, Z., Wang, L., Chi, X., Gao, W. and Wang, L.-W. (2013) Fast plane wave density functional theory molecular dynamics calculations on multi-GPU machines. *J. Comput. Phys.*, **251**, 102–115.
19. Hutter, J., Iannuzzi, M., Schiffmann, F. and VandeVondele, J. (2013) CP2K: atomistic simulations of condensed matter systems. *Wiley Interdiscip. Rev. Comput. Mol. Sci.*, **4** (1), 15–25.
20. Hehre, W.J., Radom, L., von Rague Schleyer, P. and Pople, J. (1986) *Ab Initio Molecular Orbital Theory*, Wiley-Interscience, John Wiley & Sons, Inc.
21. Ufimtsev, I. and Martínez, T. (2008) Graphical processing units for quantum chemistry. *Comput. Sci. Eng.*, **10**, 26–34.

22. Ufimtsev, I.S. and Martínez, T.J. (2008) Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation. *J. Chem. Theory Comput.*, **4**, 222–231.
23. Asadchev, A., Allada, V., Felder, J., Bode, B.M., Gordon, M.S. and Windus, T.L. (2010) Uncontracted Rys quadrature implementation of up to G functions on graphical processing units. *J. Chem. Theory Comput.*, **6**, 696–704.
24. Titov, A.V., Ufimtsev, I.S., Luehr, N. and Martínez, T.J. (2013) Generating efficient quantum chemistry codes for novel architectures. *J. Chem. Theory Comput.*, **9**, 213–221.
25. Ufimtsev, I.S. and Martínez, T.J. (2009) Quantum chemistry on graphical processing units. 2. Direct self-consistent-field implementation. *J. Chem. Theory Comput.*, **5**, 1004–1015.
26. Ufimtsev, I.S. and Martínez, T.J. (2009) Quantum chemistry on graphical processing units. 3. Analytical energy gradients, geometry optimization, and first principles molecular dynamics. *J. Chem. Theory Comput.*, **5**, 2619–2628.
27. Becke, A.D. (1989) Basis-set-free density-functional quantum chemistry. *Int. J. Quantum Chem.*, **36**, 599–609.
28. Chelikowsky, J.R., Troullier, N. and Saad, Y. (1994) Finite-difference-pseudopotential method: electronic structure calculations without a basis. *Phys. Rev. Lett.*, **72**, 1240–1243.
29. Briggs, E.L., Sullivan, D.J. and Bernholc, J. (1995) Large-scale electronic-structure calculations with multigrid acceleration. *Phys. Rev. B*, **52**, R5471–R5474.
30. Fattebert, J.-L. and Bernholc, J. (2000) Towards grid-based $O(N)$ density-functional theory methods: optimized nonorthogonal orbitals and multigrid acceleration. *Phys. Rev. B*, **62**, 1713–1722.
31. Fattebert, J.-L. and Nardelli, M.B. (2003) Finite difference methods for Ab initio electronic structure and quantum transport calculations of nanostructures, in *Special Volume, Computational Chemistry*, Handbook of Numerical Analysis, vol. **10** (ed. C.L. Bris), Elsevier, pp. 571–612.
32. Beck, T.L. (2000) Real-space mesh techniques in density-functional theory. *Rev. Mod. Phys.*, **72**, 1041–1080.
33. Marques, M.A., Castro, A., Bertsch, G.F. and Rubio, A. (2003) Octopus: a first-principles tool for excited electron–ion dynamics. *Comput. Phys. Commun.*, **151**, 60–78.
34. Torsti, T., Heiskanen, M., Puska, M.J. and Nieminen, R.M. (2003) MIKA: multigrid-based program package for electronic structure calculations. *Int. J. Quantum Chem.*, **91**, 171–176.
35. Hirose, K. (2005) *First-Principles Calculations in Real-Space Formalism: Electronic Configurations and Transport Properties of Nanostructures*, Imperial College Press.
36. Mortensen, J.J., Hansen, L.B. and Jacobsen, K.W. (2005) Real-space grid implementation of the projector augmented wave method. *Phys. Rev. B*, **71**, 035109.
37. Kronik, L., Makmal, A., Tiago, M.L., Alemany, M.M.G., Jain, M., Huang, X., Saad, Y. and Chelikowsky, J.R. (2006) PARSEC—the pseudopotential algorithm for real-space electronic structure calculations: recent advances and novel applications to nano-structures. *Phys. Status Solidi B*, **243**, 1063–1079.
38. Yabana, K., Nakatsukasa, T., Iwata, J.-I. and Bertsch, G.F. (2006) Real-time, real-space implementation of the linear response time-dependent density-functional theory. *Phys. Status Solidi B*, **243**, 1121–1138.
39. Hernández, E.R., Janecek, S., Kaczmarski, M. and Krotscheck, E. (2007) Evolution-operator method for density functional theory. *Phys. Rev. B*, **75**, 075108.
40. Iwata, J.-I., Takahashi, D., Oshiyama, A., Boku, T., Shiraishi, K., Okada, S. and Yabana, K. (2010) A massively-parallel electronic-structure calculation based on real-space density functional theory. *J. Comput. Phys.*, **229**, 2339–2363.
41. Losilla, S.A. and Sundholm, D. (2012) A divide and conquer real-space approach for all-electron molecular electrostatic potentials and interaction energies. *J. Chem. Phys.*, **136**, 214104.

42. Yabana, K. and Bertsch, G.F. (1996) Time-dependent local-density approximation in real time. *Phys. Rev. B*, **54**, 4484–4487.
43. Bertsch, G.F., Iwata, J.-I., Rubio, A. and Yabana, K. (2000) Real-space, real-time method for the dielectric function. *Phys. Rev. B*, **62**, 7998–8002.
44. Natan, A., Benjamini, A., Naveh, D., Kronik, L., Tiago, M.L., Beckman, S.P. and Chelikowsky, J.R. (2008) Real-space pseudopotential method for first principles calculations of general periodic and partially periodic systems. *Phys. Rev. B*, **78**, 075109.
45. Pittalis, S., Räsänen, E., Helbig, N. and Gross, E.K.U. (2007) Exchange-energy functionals for finite two-dimensional systems. *Phys. Rev. B*, **76**, 235314.
46. Räsänen, E., Castro, A., Werschnik, J., Rubio, A. and Gross, E.K.U. (2007) Optimal control of quantum rings by Terahertz laser pulses. *Phys. Rev. Lett.*, **98**, 157404.
47. Helbig, N., Fuks, J.I., Casula, M., Verstraete, M.J., Marques, M.A.L., Tokatly, I.V. and Rubio, A. (2011) Density functional theory beyond the linear regime: validating an adiabatic local density approximation. *Phys. Rev. A*, **83**, 032503.
48. Bernholc, J., Hodak, M. and Lu, W. (2008) Recent developments and applications of the real-space multigrid method. *J. Phys. Condens. Matter*, **20**, 294205.
49. Enkovaara, J., Rostgaard, C., Mortensen, J.J., Chen, J., Dulak, M., Ferrighi, L., Gavnholt, J., Glinsvad, C., Haikola, V., Hansen, H.A., Kristoffersen, H.H., Kuisma, M., Larsen, A.H., Lehtovaara, L., Ljungberg, M., Lopez-Acevedo, O., Moses, P.G., Ojanen, J., Olsen, T., Petzold, V., Romero, N.A., Stausholm-Møxller, J., Strange, M., Tritsaris, G.A., Vanin, M., Walter, M., Hammer, B., Häkkinen, H., Madsen, G.K.H., Nieminen, R.M., Nørskov, J.K., Puska, M., Rantala, T.T., Schiøtz, J., Thygesen, K.S. and Jacobsen, K.W. (2010) Electronic structure calculations with GPAW: a real-space implementation of the projector augmented-wave method. *J. Phys. Condens. Matter*, **22**, 253202.
50. Hasegawa, Y., Iwata, J.-I., Tsuji, M., Takahashi, D., Oshiyama, A., Minami, K., Boku, T., Shoji, F., Uno, A., Kurokawa, M., Inoue, H., Miyoshi, I. and Yokokawa, M. (2011) First-principles calculations of electron states of a silicon nanowire with 100,000 atoms on the K computer. Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, New York, SC '11, pp. 1:1–1:11.
51. Losilla Fernández, S., Watson, M.A., Aspuru-Guzik, A. and Sundholm, D. (2014) GPGPU-accelerated real-space methods for molecular electronic structure calculations, under review.
52. Castro, A., Appel, H., Oliveira, M., Rozzi, C.A., Andrade, X., Lorenzen, F., Marques, M.A.L., Gross, E.K.U. and Rubio, A. (2006) octopus: a tool for the application of time-dependent density functional theory. *Phys. Status Solidi B*, **243**, 2465–2488.
53. Munshi, A. (2009) *The OpenCL Specification*, Khronos group, Philadelphia, PA.
54. Kohn, W. and Sham, L.J. (1965) Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, **140**, A1133–A1138.
55. Hohenberg, P. and Kohn, W. (1964) Inhomogeneous electron gas. *Phys. Rev.*, **136**, B864–B871.
56. Gygi, F. and Galli, G. (1995) Real-space adaptive-coordinate electronic-structure calculations. *Phys. Rev. B*, **52**, R2229–R2232.
57. Briggs, E.L., Sullivan, D.J. and Bernholc, J. (1996) Real-space multigrid-based approach to large-scale electronic structure calculations. *Phys. Rev. B*, **54**, 14362–14375.
58. Modine, N.A., Zumbach, G. and Kaxiras, E. (1997) Adaptive-coordinate real-space electronic-structure calculations for atoms, molecules, and solids. *Phys. Rev. B*, **55**, 10289–10301.

59. Broyden, C.G. (1965) A class of methods for solving nonlinear simultaneous equations. *Math. Comput.*, **19**, 577–593.
60. Srivastava, G.P. (1984) Broyden's method for self-consistent field convergence acceleration. *J. Phys. A: Math. Gen.*, **17**, L317.
61. Pulay, P. (1980) Convergence acceleration of iterative sequences. The case of SCF iteration. *Chem. Phys. Lett.*, **73**, 393–398.
62. Szabo, A. and Ostlund, N. (1996) *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory*, Dover Books on Chemistry Series, Dover Publications.
63. Trefethen, L.N. and Bau, D. (1997) *Numerical Linear Algebra*, Society for Industrial and Applied Mathematics.
64. Saad, Y. (2011) *Numerical Methods for Large Eigenvalue Problems: Revised Edition*, Classics in Applied Mathematics, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA.
65. Wood, D.M. and Zunger, A. (1985) A new method for diagonalising large matrices. *J. Phys. A: Math. Gen.*, **18**, 1343.
66. Kresse, G. and Furthmüller, J. (1996) Efficient iterative schemes for Ab initio total-energy calculations using a plane-wave basis set. *Phys. Rev. B*, **54**, 11169–11186.
67. Saad, Y., Stathopoulos, A., Chelikowsky, J., Wu, K. and Öğüt, S. (1996) Solution of large eigenvalue problems in electronic structure calculations. *BIT Numer. Math.*, **36**, 563–578.
68. Runge, E. and Gross, E.K.U. (1984) Density-functional theory for time-dependent systems. *Phys. Rev. Lett.*, **52**, 997–1000.
69. Maitra, N.T. (2012) Memory: history, initial-state dependence, and double-excitations, in *Fundamentals of Time-Dependent Density Functional Theory*, Lecture Notes in Physics, vol. **837** (eds M.A. Marques, N.T. Maitra, F.M. Nogueira, E. Gross and A. Rubio), Springer-Verlag, Berlin, Heidelberg, pp. 167–184.
70. Castro, A., Marques, M.A.L., Alonso, J.A. and Rubio, A. (2004) Optical properties of nanostructures from time-dependent density functional theory. *J. Comput. Theor. Nanosci.*, **1**, 231–255.
71. Castro, A., Marques, M.A.L., Alonso, J.A., Bertsch, G.F. and Rubio, A. (2004) Excited states dynamics in time-dependent density functional theory. *Eur. Phys. J. D*, **28**, 211–218.
72. Takimoto, Y., Vila, F.D. and Rehr, J.J. (2007) Real-time time-dependent density functional theory approach for frequency-dependent nonlinear optical response in photonic molecules. *J. Chem. Phys.*, **127**, 154114.
73. Yabana, K. and Bertsch, G.F. (1999) Application of the time-dependent local density approximation to optical activity. *Phys. Rev. A*, **60**, 1271–1279.
74. Varsano, D., Espinosa Leal, L.A., Andrade, X., Marques, M.A.L., di Felice, R. and Rubio, A. (2009) Towards a gauge invariant method for molecular chiroptical properties in TDDFT. *Phys. Chem. Chem. Phys.*, **11**, 4481–4489.
75. Marques, M.A.L., Castro, A., Malloci, G., Mulas, G. and Botti, S. (2007) Efficient calculation of van der Waals dispersion coefficients with time-dependent density functional theory in real time: application to polycyclic aromatic hydrocarbons. *J. Chem. Phys.*, **127**, 014107.
76. Aggarwal, R., Farrar, L., Saikin, S., Andrade, X., Aspuru-Guzik, A. and Polla, D. (2012) Measurement of the absolute Raman cross section of the optical phonons in type Ia natural diamond. *Solid State Commun.*, **152**, 204–209.
77. Thomas, M., Latorre, F. and Marquetand, P. (2013) Resonance Raman spectra of ortho-nitrophenol calculated by real-time time-dependent density functional theory. *J. Chem. Phys.*, **138**, 044101.

78. De Giovannini, U., Varsano, D., Marques, M.A.L., Appel, H., Gross, E.K.U. and Rubio, A. (2012) *Ab initio* angle- and energy-resolved photoelectron spectroscopy with time-dependent density-functional theory. *Phys. Rev. A*, **85**, 062515.
79. Meng, S. and Kaxiras, E. (2008) Real-time, local basis-set implementation of time-dependent density functional theory for excited state dynamics simulations. *J. Chem. Phys.*, **129**, 054110.
80. Alonso, J.L., Andrade, X., Echenique, P., Falceto, F., Prada-Gracia, D. and Rubio, A. (2008) Efficient formalism for large-scale *Ab Initio* molecular dynamics based on time-dependent density functional theory. *Phys. Rev. Lett.*, **101**, 096403.
81. Andrade, X., Castro, A., Zueco, D., Alonso, J.L., Echenique, P., Falceto, F. and Rubio, A. (2009) Modified Ehrenfest formalism for efficient large-scale Ab initio molecular dynamics. *J. Chem. Theory Comput.*, **5**, 728–742.
82. Avendaño Franco, G., Piraux, B., Grüning, M. and Gonze, X. (2012) Time-dependent density functional theory study of charge transfer in collisions. *Theor. Chem. Acc.*, **131**, 1–10.
83. Akimov, A.V. and Prezhdo, O.V. (2013) The PYXAID program for non-adiabatic molecular dynamics in condensed matter systems. *J. Chem. Theory Comput.*, **9**, 4959–4972.
84. Akimov, A.V. and Prezhdo, O.V. (2014) Advanced capabilities of the PYXAID program: integration schemes, decoherence effects, multi-excitonic states, and field-matter interaction. *J. Chem. Theory. Comput.*, **10**, 789–804.
85. Castro, A., Marques, M.A.L. and Rubio, A. (2004) Propagators for the time-dependent Kohn-Sham equations. *J. Chem. Phys.*, **121**, 3425–3433.
86. Andrade, X., Sanders, J.N. and Aspuru-Guzik, A. (2012) Application of compressed sensing to the simulation of atomic systems. *Proc. Natl. Acad. Sci. U. S. A.*, **109**, 13928–13933.
87. Markovich, T., Blau, S.M., Parkhill, J., Kreisbeck, C., Sanders, J.N., Andrade, X. and Aspuru-Guzik, A. (2013) More accurate and efficient bath spectral densities from super-resolution, *arXiv preprint arXiv:1307.4407*
88. Tuckerman, M.E. and Parrinello, M. (1994) Integrating the Car–Parrinello equations. I. Basic integration techniques. *J. Chem. Phys.*, **101**, 1302–1315.
89. Baroni, S., de Gironcoli, S., Dal Corso, A. and Giannozzi, P. (2001) Phonons and related crystal properties from density-functional perturbation theory. *Rev. Mod. Phys.*, **73**, 515–562.
90. Andrade, X., Botti, S., Marques, M.A.L. and Rubio, A. (2007) Time-dependent density functional theory scheme for efficient calculations of dynamic (hyper)polarizabilities. *J. Chem. Phys.*, **126**, 184106.
91. Peng, L., Seymour, R., Nomura, K.-I., Kalia, R.K., Nakano, A., Vashishta, P., Loddoch, A., Netzband, M., Volz, W. and Wong, C. (2009) High-order stencil computations on multicore clusters. IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009, pp. 1–11.
92. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J. and Yelick, K. (2009) Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.*, **51**, 129–159.
93. Dursun, H., Nomura, K.-I., Peng, L., Seymour, R., Wang, W., Kalia, R., Nakano, A. and Vashishta, P. (2009) A multilevel parallelization framework for high-order stencil computations, in *Euro-Par 2009 Parallel Processing*, Lecture Notes in Computer Science, vol. **5704** (eds H. Sips, D. Epema and H.-X. Lin), Springer-Verlag, Berlin, Heidelberg, pp. 642–653.
94. Treibig, J., Wellein, G. and Hager, G. (2011) Efficient multicore-aware parallelization strategies for iterative stencil computations. *J. Comput. Sci.*, **2**, 130–137.
95. de la Cruz, R. and Araya-Polo, M. (2011) Towards a multi-level cache performance model for 3D stencil computation. *Procedia Comput. Sci.*, **4**, 2146–2155.

96. Henretty, T., Stock, K., Pouchet, L.-N., Franchetti, F., Ramanujam, J. and Sadayappan, P. (2011) Data layout transformation for stencil computations on short-vector SIMD architectures, in *Compiler Construction*, Lecture Notes in Computer Science, vol. **6601** (ed. J. Knoop), Springer-Verlag, Berlin, Heidelberg, pp. 225–245.
97. Holewinski, J., Pouchet, L.-N. and Sadayappan, P. High-performance code generation for stencil computations on GPU architectures. Proceedings of the 26th ACM international conference on Supercomputing, ACM, New York, ICS '12, pp. 311–320.
98. Peano, G. (1890) Sur une courbe, qui remplit toute une aire plane. *Math. Ann.*, **36**, 157–160.
99. Hilbert, D. (1891) Über die stetige Abbildung einer Linie auf ein Flächenstück. *Math. Ann.*, **36**, 459–460.
100. Skilling, J. (2004) Programming the Hilbert curve. AIP Conference Proceedings, vol. 707, pp. 381–387.
101. Andrade, X. (2010) Linear and non-linear response phenomena of molecular systems within time-dependent density functional theory. PhD thesis. University of the Basque Country, UPV/EHU.
102. Kleinman, L. and Bylander, D.M. (1982) Efficacious form for model pseudopotentials. *Phys. Rev. Lett.*, **48**, 1425–1428.
103. Troullier, N. and Martins, J.L. (1991) Efficient pseudopotentials for plane-wave calculations. *Phys. Rev. B*, **43**, 1993–2006.
104. Benoit, C. (1924) Note Sur Une Méthode de Résolution des Équations Normales Provenant de L'Application de la Méthode des Moindres Carrés a un Systéme D'equations Linéaires en Nombre Inférieur a Celui des Inconnues.—Application de la Méthode a la R ésolution D'un Systéme Defini D'equations Linéaires. *Bull. Geod.*, **2**, 67–77.
105. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P. and Tomov, S. (2009) Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *J. Phys. Conf. Ser.*, **180**, 012037.
106. Perdew, J.P. and Zunger, A. (1981) Self-interaction correction to density-functional approximations for many-electron systems. *Phys. Rev. B*, **23**, 5048–5079.
107. Umezawa, N. (2006) Explicit density-functional exchange potential with correct asymptotic behavior. *Phys. Rev. A*, **74**, 032505.
108. Andrade, X. and Aspuru-Guzik, A. (2011) Prediction of the derivative discontinuity in density functional theory from an electrostatic description of the exchange and correlation potential. *Phys. Rev. Lett.*, **107**, 183002.
109. Shang, H., Li, Z. and Yang, J. (2010) Implementation of exact exchange with numerical atomic orbitals. *J. Phys. Chem. A*, **114**, 1039–1043.
110. Tan, I.-H., Snider, G.L., Chang, L.D. and Hu, E.L. (1990) A self-consistent solution of Schrödinger–Poisson equations using a nonuniform mesh. *J. Appl. Phys.*, **68**, 4071–4076.
111. Luscombe, J.H., Bouchard, A.M. and Luban, M. (1992) Electron confinement in quantum nanostructures: self-consistent Poisson-Schrödinger theory. *Phys. Rev. B*, **46**, 10262–10268.
112. Klamt, A. and Schuurmann, G. (1993) COSMO: a new approach to dielectric screening in solvents with explicit expressions for the screening energy and its gradient. *J. Chem. Soc., Perkin Trans. 2*, 799–805.
113. Tomasi, J. and Persico, M. (1994) Molecular interactions in solution: an overview of methods based on continuous distributions of the solvent. *Chem. Rev.*, **94**, 2027–2094.
114. Olivares-Amaya, R., Stopa, M., Andrade, X., Watson, M.A. and Aspuru-Guzik, A. (2011) Anion stabilization in electrostatic environments. *J. Phys. Chem. Lett.*, **2**, 682–688.
115. Watson, M.A., Rappoport, D., Lee, E.M.Y., Olivares-Amaya, R. and Aspuru-Guzik, A. (2012) Electronic structure calculations in arbitrary electrostatic environments. *J. Chem. Phys.*, **136**, 024101 (14 pages).

116. Greengard, L.F. and Rokhlin, V. (1997) A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta Numer.*, **6**, 229.
117. Kutteh, R., Apra, E. and Nichols, J. (1995) A generalized fast multipole approach for Hartree-Fock and density functional computations. *Chem. Phys. Lett.*, **238**, 173–179.
118. Briggs, W.L. (1987) *A Multigrid Tutorial*, John Wiley & Sons, Inc., New York.
119. Beck, T.L. (1997) Real-space multigrid solution of electrostatics problems and the Kohn–Sham equations. *Int. J. Quantum Chem.*, **65**, 477–486.
120. Cerioni, A., Genovese, L., Mirone, A. and Sole, V.A. (2012) Efficient and accurate solver of the three-dimensional screened and unscreened Poisson's equation with generic boundary conditions. *J. Chem. Phys.*, **137**, 134108 (9 pages).
121. García-Risueño, P., Alberdi-Rodriguez, J., Oliveira, M.J.T., Andrade, X., Pippig, M., Muguerza, J., Arruabarrena, A. and Rubio, A. (2013) A survey of the parallel performance and accuracy of poisson solvers for electronic structure calculations. *J. Comput. Chem.*, **35** (6), 427–444.
122. Rozzi, C.A., Varsano, D., Marini, A., Gross, E.K.U. and Rubio, A. (2006) Exact Coulomb cutoff technique for supercell calculations. *Phys. Rev. B*, **73**, 205119.
123. Frigo, M. and Johnson, S.G. (2005) The design and implementation of FFTW3. *Proc. IEEE*, 2005, **93**, 216–231, Special issue on "Program Generation, Optimization, and Platform Adaptation".
124. Becke, A.D. (1993) A new mixing of Hartree–Fock and local density-functional theories. *J. Chem. Phys.*, **98**, 1372–1377.
125. Marques, M.A., Oliveira, M.J. and Burnus, T. (2012) LIBXC: a library of exchange and correlation functionals for density functional theory. *Comput. Phys. Commun.*, **183**, 2272–2281.
126. Goedecker, S., Teter, M. and Hutter, J. (1996) Separable dual-space Gaussian pseudopotentials. *Phys. Rev. B*, **54**, 1703–1710.
127. Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Booth, M. and Rossi, F. (2003) *GNU Scientific Library: Reference Manual*, Network Theory Ltd.
128. Luehr, N., Ufimtsev, I.S. and Martínez, T.J. (2011) Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs). *J. Chem. Theory Comput.*, **7**, 949–954.
129. Harrison, R.J., Fann, G.I., Yanai, T., Gan, Z. and Beylkin, G. (2004) Multiresolution quantum chemistry: basic theory and initial applications. *J. Chem. Phys.*, **121**, 11587–11598.
130. Genovese, L., Neelov, A., Goedecker, S., Deutsch, T., Ghasemi, S.A., Willand, A., Caliste, D., Zilberberg, O., Rayson, M., Bergman, A. and Schneider, R. (2008) Daubechies wavelets as a basis set for density functional pseudopotential calculations. *J. Chem. Phys.*, **129**, 014109 (14 pages).
131. Baer, R., Neuhauser, D. and Rabani, E. (2013) Self-averaging stochastic Kohn-Sham density-functional theory. *Phys. Rev. Lett.*, **111**, 106402.
132. Neuhauser, D., Rabani, E. and Baer, R. (2013) Expeditious stochastic approach for MP2 energies in large electronic systems. *J. Chem. Theory Comput.*, **9**, 24–27.
133. Neuhauser, D., Rabani, E. and Baer, R. (2013) Expeditious stochastic calculation of random-phase approximation energies for thousands of electrons in three dimensions. *J. Phys. Chem. Lett.*, **4**, 1172–1176.
134. The Octopus source code can be obtained from `http://tddft.org/programs/octopus/` (accessed 25 September 2015).
135. Andrade, X. (2011) FortranCL: a Fortran/OpenCL interface, `http://fortrancl.googlecode.com` (accessed 21 September 2015).
136. Tafipolsky, M. and Schmid, R. (2006) A general and efficient pseudopotential Fourier filtering scheme for real space methods using mask functions. *J. Chem. Phys.*, **124**, 174102 (9 pages).

137. Becke, A.D. (1988) Density-functional exchange-energy approximation with correct asymptotic behavior. *Phys. Rev. A*, **38**, 3098–3100.
138. Lee, C., Yang, W. and Parr, R.G. (1988) Development of the Colle-Salvetti correlation-energy formula into a functional of the electron density. *Phys. Rev. B*, **37**, 785–789.
139. Miehlich, B., Savin, A., Stoll, H. and Preuss, H. (1989) Results obtained with the correlation energy density functionals of Becke and Lee, Yang and Parr. *Chem. Phys. Lett.*, **157**, 200–206.

11

Semiempirical Quantum Chemistry

Xin Wu, Axel Koslowski and Walter Thiel
Max-Planck-Institut für Kohlenforschung, Mülheim an der Ruhr, Germany

In this chapter, we demonstrate how graphics processing units (GPUs) can be used to accelerate large-scale semiempirical quantum-chemical calculations on hybrid CPU–GPU platforms. We examine the computational bottlenecks using a series of calculations on eight proteins with up to 3558 atoms and outline how relevant operations are parallelized and ported to GPUs, making use of multiple devices where possible. Significant speedups are achieved that enable simulations on large systems with thousands of atoms. As an example we present results for geometry optimizations of three representative proteins with α-helix, β-sheet, and random coil structures using several common semiempirical Hamiltonians.

11.1 Introduction

Semiempirical quantum chemical methods are cost-effective tools for chemists to study the structure, stability, and spectroscopy of molecules as well as chemical reactions [1] (see also Chapter 3). They are based on the Hartree–Fock method commonly used in *ab initio* molecular orbital (MO) theory [2]. The different semiempirical models simplify the Hartree–Fock procedure by introducing distinct approximations to the Hamiltonian, neglecting many integrals to speed up computations by several orders of magnitude [3]. The remaining integrals are modeled using empirical functions with adjustable parameters that are calibrated against a large number of accurate experimental or high-level theoretical reference data to make semiempirical methods as reliable and general as possible. These features make semiempirical models well suited to many research areas in chemistry, and enabled a large number of semiempirical applications already in the 1970s and 1980s. Since the 1990s, density functional theory (DFT) has become the major workhorse in computational chemistry [4]. However, considering that semiempirical methods are ∼1000× faster than standard DFT approaches [5], they are still valuable computational tools nowadays, for example, for screening large numbers of drug

Electronic Structure Calculations on Graphics Processing Units:
From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

candidates [6], for calculations on proteins [7], for long-time scale ground-state molecular dynamics simulations [8], and for nonadiabatic excited-state dynamics of large chromophores [9].

The development of computational chemistry is intimately tied to the evolution of computer technology. The first computational chemistry programs developed since the 1950s had been exclusively written for sequential execution on a single central processing unit (CPU) [10]. With the widespread advent of parallel computing in the 1990s, many quantum chemical codes were parallelized to take advantage of the new architectures, including semiempirical programs [11]. The most recent wave of hardware-driven code development was triggered by the rise of graphics processing units (GPUs). A GPU is a specially designed integrated circuit with powerful, but fixed-function pipelines for faster image rendering and video games (see also Chapter 1). Until 2006, implementing algorithms for general numeric calculations on a GPU was tediously difficult because the problem had to be cast into graphics operations by resorting to a specific (graphics) API (application programming interface). Programming purely computational tasks on a GPU was considerably simplified by the introduction of the CUDA (compute unified device architecture) and OpenCL (open computing language) frameworks (see Chapter 2). In this chapter, we will focus exclusively on the CUDA framework, which allows developers to employ the C programming language, with CUDA-specific extensions, to use a CUDA-capable GPU as coprocessor of the CPU for computations [12]. As of 2012, the raw hardware peak performance and memory bandwidth of a many-core GPU had significantly outpaced a multicore CPU (see Figure 1.5 in Chapter 1). For example, the maximum floating-point performance and theoretical memory bandwidth of an Intel Xeon E5-4650 CPU (eight cores with a base clock of 2.7 GHz and a maximum boost clock of 3.3 GHz with the Intel Turbo Boost Technology, four-channel DDR-1600) are 0.17–0.21 TFlop/s (floating-point operations per second) and 51.2 GB/s, respectively. By contrast, the flagship Tesla K20x by Nvidia (2688 CUDA cores at 732 MHz) has a peak of 1.31 TFlop/s for double-precision arithmetic and a memory bandwidth of 250 GB/s with ECC (error-correcting code) off. Hence many groups decided to develop GPU-accelerated programs [13, 14] to take advantage of this promising device for quantum Monte Carlo computations [15, 16], evaluation of two-electron integrals [17–22], DFT calculations [23–30], high-level correlated *ab initio* methods [31–38], and semiempirical quantum chemistry [39, 40]. The other chapters of this book contain an excellent overview of many of these porting efforts.

In this chapter, we begin with a brief review of semiempirical quantum chemistry, referring readers interested in the detailed formalism and the numerical results to available books [41–43] and reviews [5, 11, 44–50]. We then examine the computational bottlenecks by performing systematic calculations on a set of eight proteins with up to 3558 atoms and 8727 basis functions. Thereafter, we outline how the hotspots identified in this manner are ported to a GPU (making use of multiple devices where possible), and how the remaining code is parallelized using CPUs only using the symmetric multiprocessing (SMP) capabilities via OpenMP. Next, we analyze the overall performance of our code on the hybrid CPU–GPU platform and compare it with the CPU-only case. Finally, as an illustrative application, we use our CPU–GPU hybrid program to optimize the geometries of three small proteins, each consisting predominantly of one type of secondary structure, namely α-helix, β-strand, and random coil, employing six different semiempirical methods.

11.2 Overview of Semiempirical Methods

Nonrelativistic quantum chemistry aims at finding sufficiently accurate but approximate solutions to the Schrödinger equation. In the early days of quantum chemistry, the zero-differential-overlap (ZDO) approximation [51, 52] was introduced to deal with "the nightmare of the integrals" [10], that is, the difficulty of evaluating the large number of three- and four-center integrals in *ab initio* methods. As a consequence, the integral problem could be tackled at different levels of approximation. Currently, the most accurate semiempirical methods are based on the NDDO (neglect of diatomic

differential overlap) model [3], which retains all one- and two-center two-electron repulsion integrals in the Fock matrix. The first successful and widely adopted NDDO-based parameterization was the MNDO (modified neglect of diatomic overlap) method [53–55]. The MNDO model also serves as the basis for later parameterizations that have been widely applied, including AM1 (Austin Model 1) [56], PM*x* (parametric methods, $x = 3$, 5, 6, and 7) [57–60], as well as PDDG/MNDO and PDDG/PM3 (MNDO and PM3 augmented with pairwise distance directed Gaussian functions) [61].

Conceptual deficiencies in the established MNDO-type methods include the lack of representation of Pauli exchange repulsion in the Fock matrix. One possible remedy is to introduce orthogonalization corrections into the Fock matrix to account for Pauli exchange repulsion. This can be done through truncated and parameterized series expansions in terms of overlap, which provide corrections to the one-electron core Hamiltonian. These corrections are applied to the one-center matrix elements in OM1 (orthogonalization model 1) [62] and to all one- and two-center matrix elements in OM2 [63] and OM3 [64]. Benchmark calculations demonstrate that the OM*x* methods, especially OM2 and OM3, are superior to AM1 and PM3 for both ground-state and excited-state molecular properties [65–67]. The computational cost of OM*x* calculations is roughly the same as that for MNDO-type calculations [39], especially when using suitable cutoffs to neglect the exponentially decreasing three-center orthogonalization corrections to matrix elements involving distant atoms.

11.3 Computational Bottlenecks

In this chapter, the OM3 method is taken as an example to illustrate the general strategy of optimizing a semiempirical quantum chemical program on a hybrid CPU–GPU platform. We have selected a set of eight proteins that are denoted as P_x (x being the number of residues) and listed in Table 11.1, for the purpose of profiling OM3 calculations in a systematic manner [68–75]. Timings for the OM*x* methods are also representative for MNDO-type methods, because the most time-consuming parts of the calculations are the same in both cases. Consequently, similar wall clock times are obtained: for example, one SCF (self-consistent field) iteration in MNDO, AM1, PM3, OM1, OM2, and OM3 calculations on a cluster of 1000 water molecules takes 80, 84, 89, 73, 87, and 83 seconds, respectively, on a single Intel Xeon X5670 CPU core [39]. Hence, it is sufficient to consider only OM3 in the following.

The OM3 calculations on our test proteins were performed on a server with two Intel Xeon X5690 CPUs (six cores at 3.46 GHz per chip), 48 GiB host memory (24 GiB of triple-channel DDR-1333 per chip) with a total theoretical bandwidth[1] of 64 GB/s, and two Nvidia Tesla M2090 GPUs (512 CUDA cores at 1.3 GHz per device) with 5.25 GiB ECC memory and a bandwidth of 155 GB/s per device. Intel Turbo Boost Technology (which may automatically increase the CPU frequency above the base clock in accordance with the workload in order to exhaust the allowed thermal envelope of the CPU) was intentionally turned off to ensure consistent timings. Three criteria were adopted for SCF convergence in our single-point energy calculations: (i) a variation of the electronic energy in successive

Table 11.1 *Proteins in the test set for the OM3 calculations*

Notation	P_{020}	P_{063}	P_{086}	P_{100}	P_{125}	P_{156}	P_{166}	P_{221}
PDB ID	1BTQ	1K50	2HXX	3K6F	1ACF	2A4V	4A02	3AQO
N_a	307	1097	1495	1842	2004	2969	3415	3558
N_f	754	2699	3655	4446	4920	7157	8173	8727

N_a and N_f denote the number of atoms and basis functions, respectively.

[1] If one CPU needs to access memory connected to the other CPU, the theoretical bandwidth is lower.

SCF iterations of at most 1.0×10^{-6} eV, (ii) a maximum change of the density matrix elements of 1.0×10^{-6}, and (iii) a maximum entry in the error matrix of 1.0×10^{-6} in the DIIS (direct inversion of iterative subspace) extrapolation [76]. To speed up the calculations, the full diagonalization was automatically replaced in the SCF procedure by fast pseudo-diagonalization [77] whenever possible.

The code development was conducted on a CVS version of the MNDO99 package [78]. The Intel composer XE 13.1 and Nvidia CUDA Toolkit 5.0 were used for compiling the Fortran subroutines of the CPU code and the CUDA kernels for the GPU, respectively. The final executable was dynamically linked against Intel Math Kernel Library (MKL) 11.0, CUBLAS from the Nvidia Toolkit, and MAGMA version 1.3.0 [79]. The latter includes a subset of LAPACK routines ported to the GPU; it has been modified locally to conform to the ILP64 (64-bit integers, long integers, and pointers) data model, which is needed to access arrays with 2^{32} or more elements.[2] Before the inclusion of dynamic memory allocation in the Fortran standard, the early versions of the MNDO program emulated dynamic memory by passing sections of a fixed-size array in the unnamed COMMON block as arguments to subroutines. The current version of the MNDO code uses essentially the same mechanism, but with a dynamically allocated array instead of the fixed-size array. For larger proteins, the indices of this array may exceed the 32-bit integer range—this is why 64-bit integers are needed.

The computing setup for the OM3 benchmark calculations is denoted as $C_{[xC-yG]}$, where the subscripts x and y are numbers of CPU cores and GPU devices in use, respectively. The wall clock time of an OM3 calculation on $C_{[1C]}$ is the reference for calculations with the other compute configurations and the basis for assessing the corresponding speedups. Timings for $C_{[1G]}$ and $C_{[2G]}$ refer to subroutines executed exclusively on one GPU or two GPUs, respectively, including the associated and generally negligible CPU–GPU communication. All floating-point operations were done in double precision, both on the CPUs and GPUs, and therefore the numerical results produced on all hardware setups are essentially the same. Deviations in the computed heat of formation (total energy) were occasionally encountered, but remained below 1.0×10^{-5} kcal/mol. Such tiny discrepancies can be attributed to the different order in which the floating-point operations are performed on the CPU and GPU architectures. Since many operations are performed in parallel, the execution order may not even be fixed, that is, there might be small deviations between different runs of the same calculation on the same computing setup. The execution order matters because limited-precision floating-point arithmetics is not associative.

The general form of a two-electron repulsion integral (ERI) in *ab initio* and DFT methods is

$$(\mu\nu|\lambda\sigma) = \int_1\int_2 \frac{\mu(1)\ \nu(1)\ \lambda(2)\ \sigma(2)}{r_{12}}\ dV_1\ dV_2,$$

where the Greek letters represent basis functions or atomic orbitals (AOs). The complexity of the two-electron integral evaluation formally scales as $\mathcal{O}(N_f^4)$ for N_f basis functions, but the actual scaling may be more favorable due to the application of screening techniques [80]. The currently applied semiempirical methods make use of the NDDO approximation [3] for ERI evaluation:

$$(\mu_A\nu_B|\lambda_C\sigma_D) = \delta_{AB}\ \delta_{CD}\ (\mu_A\nu_B|\lambda_C\sigma_D),$$

where atomic centers are denoted by capital letters and δ_{AB} (or δ_{CD}) will vanish unless A and B (or C and D) are the same atom. This rather drastic approximation reduces the formal scaling of the ERI computation in semiempirical methods to $\mathcal{O}(N_f^2)$ and makes it possible to simulate complex systems with thousands of atoms. The solution of the secular equations

$$\sum_\nu (F_{\mu\nu} - \delta_{\mu\nu}\epsilon_i)C_{\nu i} = 0 \tag{11.1}$$

scales as $\mathcal{O}(N_f^3)$ and thus becomes the primary computational task in semiempirical methods. ϵ_i is the energy of the ith MO. Because the Fock matrix elements $F_{\mu\nu}$ depend on the elements $C_{\nu i}$ of the

[2] Starting with version 1.4, MAGMA supports both 32-bit and 64-bit integers out of the box.

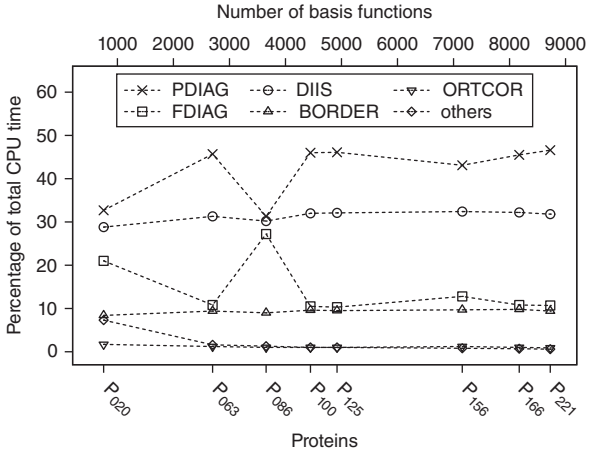


Figure 11.1 *Profiles of the OM3 calculations for the test proteins for the $C_{[1C]}$ computing setup*

eigenvectors, Eq. (11.1) has to be solved by an iterative SCF procedure that requires $\mathcal{O}(N_f^3)$ dense linear algebraic operations.

Figure 11.1 depicts the profiles of OM3 calculations for the $C_{[1C]}$ setup. The pseudo-diagonalization procedure (PDIAG) is roughly twice as fast as a full diagonalization (FDIAG), and it is thus preferable to replace FDIAG by PDIAG as often as possible. Applying the default criteria of the MNDO code for the choice between FDIAG and PDIAG, it is normally sufficient to call FDIAG in four of the SCF iterations (i.e., the first three and the last one) during single-point energy evaluation and to call PDIAG in the other SCF iterations (typically 25).[3] Hence most OM3 calculations are dominated by PDIAG with 42.1% of the wall clock time on average. FDIAG and PDIAG complement each other; they collectively contribute ∼55% of the total CPU time and are thus the first two bottlenecks.

DIIS is the third hotspot that consumes ∼30% of the computation time (see Figure 11.1). Although the DIIS extrapolation may be omitted for small systems (with less than 100 atoms), it is in our experience imperative to apply DIIS to reliably converge the SCF procedure for larger molecules such as proteins. We will thus also investigate the option of leveraging multiple GPUs for the DIIS treatment (see Section 11.4).

The last two bottlenecks are the calculation of the density matrix (also called the bond-order matrix, subroutine BORDER) and the orthogonalization corrections (subroutine ORTCOR in the case of OM3). We spent considerable effort on both routines to achieve optimum performance with the MNDO99 program [78], especially for ORTCOR, where we obtained a huge speedup by formulating all operations as standard matrix–matrix multiplications. After code optimization, BORDER and ORTCOR take 9.4% and 1.1% of the wall clock time on average, respectively, on the $C_{[1C]}$ setup.

Other computational tasks in an OM3 calculation include integral evaluation, formation of the Fock matrix, and initial density matrix generation, which all scale as $\mathcal{O}(N_f^2)$. Cumulatively, they require 7% of the CPU time in a serial calculation for a small protein such as P_{020} with 307 atoms and 754 orbitals, but this portion quickly diminishes with increasing system size, to ∼0.5% for the largest proteins in our test set, which are the main targets of our code development. Therefore, these other tasks are not considered to be real bottlenecks, and the corresponding subroutines are thus only subjected to an OpenMP parallelization to take advantage of multiple CPU cores.

[3] An exception is P_{086} with 11 calls to FDIAG.

In summary, we have identified five subroutines (FDIAG, PDIAG, DIIS, BORDER, and ORTCOR) as computational bottlenecks by systematic analysis of OM3 calculations on a set of proteins. We describe the optimization of these hotspots on a hybrid CPU–GPU platform in the following.

11.4 Profile-Guided Optimization for the Hybrid Platform

11.4.1 Full Diagonalization, Density Matrix, and DIIS

The GPU-accelerated full diagonalization, density matrix construction, and DIIS extrapolation are jointly described here because they heavily rely on the standard routines in the BLAS (basic linear algebra subprograms) and LAPACK (linear algebra package) libraries.

Equation (11.1) is an eigenvalue problem that can be solved by diagonalizing the Fock matrix $\mathbf{F}$, which yields the ith MO energy ϵ_i and the coefficient vector $\mathbf{c}_i$:

$$\mathbf{F}\mathbf{c}_i = \epsilon_i \mathbf{c}_i.$$

This task can be carried out by the LAPACK function DSYEVD, which computes all eigenvalues and eigenvectors of a real symmetric matrix using the divide-and-conquer algorithm. DSYEVD of the Intel MKL library makes use of all processor cores on a CPU-only platform, whereas the DSYEVD implementation in MAGMA is a hybrid that utilizes both multicore CPUs and GPU(s)[4] for the diagonalization [81]. In Figure 11.2, the speedups of FDIAG are plotted as obtained in the OM3 calculations on the proteins in our test set. The scalability on CPU-only setups is evidently rather poor: for instance, the best speedups are observed in the calculations on P_{063}, which are 4.3 on $C_{[6C]}$ and 5.4 on $C_{[12C]}$. Hence, the symmetric parallel processors are highly underutilized in the FDIAG subroutine, and the efficiency[5] is merely 0.72 and 0.45, respectively. This becomes even worse for larger systems: for example, the speedup of FDIAG for P_{221} on $C_{[6C]}$ is 3.3 and barely increases to 3.8

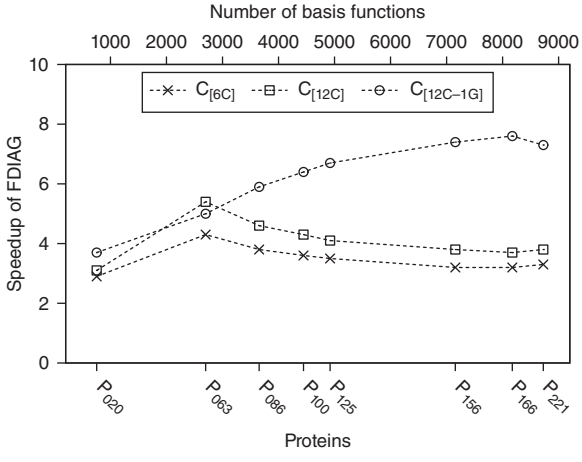


Figure 11.2 *Speedups of the FDIAG subroutine in the OM3 calculations on the multi-CPU $C_{[6C]}$, $C_{[12C]}$, and hybrid CPU–GPU $C_{[12C-1G]}$ computing setups over the serial configuration*

[4] The hybrid DSYEVD function in MAGMA version 1.3 does not support multiple GPUs. This feature is available starting with MAGMA version 1.4.

[5] Processor efficiency is defined as the speedup divided by the number of parallel processing units.

on $C_{[12C]}$, with corresponding efficiencies of 0.55 and 0.32, respectively. On the contrary, the speedup of the hybrid FDIAG subroutine is constantly rising until P_{166} (up to more than 8000 basis functions). Moreover, it is always superior to its CPU-only counterpart with the exception of P_{063} on the $C_{[12C]}$ setup. For the larger calculations, the hybrid FDIAG subroutine tends to be more than 7× faster than the serial version and at least 2× faster than the parallel CPU-only version.

The primary computational task (>99% of the CPU time) in BORDER is a matrix–matrix multiplication, $\mathbf{P} = 2\mathbf{C}_o\mathbf{C}_o^T$, where **P** is the density matrix and $\mathbf{C}_o$ is the coefficient matrix of the occupied MOs. A general DGEMM routine could be used to perform this task. Because **P** is symmetric, and only the lower triangle is stored as a linear array in the MNDO99 package, we employ a more specific function, namely DSYRK, which only calculates the lower part of a symmetric matrix and thus avoids unnecessary floating-point operations. The CPU-only DSYRK routine has no difficulty to fully load all processors, and the performance scales almost ideally with respect to the number of CPU cores (see Figure 11.3). For example, the speedups for P_{166} are 5.8 on $C_{[6C]}$ and 9.9 on $C_{[12C]}$. At present, no multi-GPU-enabled version of DSYRK is available in either CUBLAS or MAGMA. On the other hand, DSYRK on a single GPU may be more than 20× faster than a single-threaded CPU routine. Thus, we will stick to DSYRK in our development, hoping that multi-GPU support will be added by the vendors in the future.

The DIIS procedure is composed of several different kinds of algebraic operations, in which the calculation of the error matrix ($\mathbf{\Delta} = \mathbf{FP} - \mathbf{PF}$) usually consumes more than 98% of the CPU time [39]. Because the product of **F** and **P** is a general matrix, the standard DGEMM function is chosen for the DIIS subroutine. The number of floating-point operations and memory accesses in DGEMM scale as $\mathcal{O}(N^3)$ and $\mathcal{O}(N^2)$ (N being the matrix dimension), respectively. This implies that the number of compute operations per memory access is proportional to N in DGEMM. Thus DGEMM is a compute-bound routine that should be well suited to parallelization. The observed speedups on the CPU-only setups are ∼5.5 on $C_{[6C]}$ and ∼10.0 on $C_{[12C]}$. Moreover, a call to DIIS accelerated by a single GPU ($C_{[1G]}$) can be up to 20× faster than for the baseline setup $C_{[1C]}$. However, the speedup for $C_{[1G]}$ turns out not to be monotonous with increasing system size: it is highest for P_{125} with ∼20 and then drops again for the next-larger protein P_{156} to ∼18.

In order to make the best use of our dual-GPU equipped hardware, we designed a block matrix scheme for the matrix–matrix multiplication aimed at multiple GPU devices based on the standard DGEMM routine (X. Wu, A. Koslowski, W. Thiel, unpublished results). There are of course more

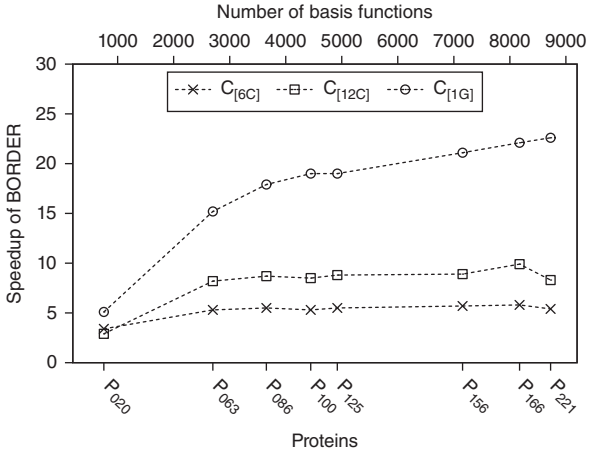


Figure 11.3 *Speedups of the BORDER subroutine in the OM3 calculations on the multi-CPU $C_{[6C]}$, $C_{[12C]}$, and GPU-only $C_{[1G]}$ computing setups over the serial configuration*

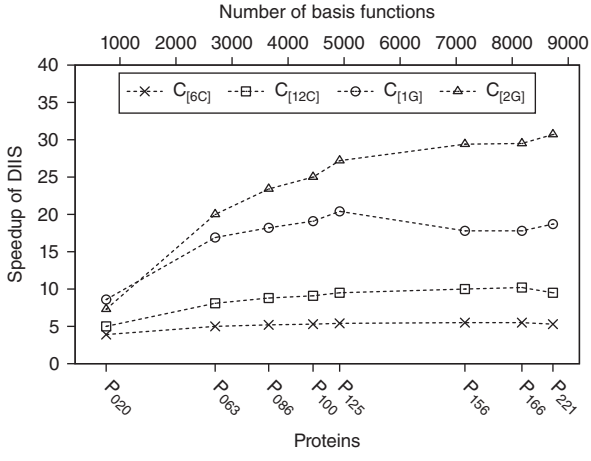


Figure 11.4 *Speedups of the DIIS subroutine in the OM3 calculations on the multi-CPU $C_{[6C]}$, $C_{[12C]}$, and GPU-only $C_{[1G]}$ and $C_{[2G]}$ computing setups over the serial configuration*

sophisticated multi-GPU DGEMM implementations reported in the literature [82, 83], but the performance of our homemade multi-GPU DGEMM is virtually doubled on two GPUs ($C_{[2G]}$ compared to $C_{[1G]}$) with a peak around 0.7 TFlop/s.

The overall speedup for the DIIS procedure with the multi-GPU DGEMM routine on the $C_{[2G]}$ setup is plotted in Figure 11.4. We find a monotonous increase in performance up to a factor of 30 compared with the $C_{[1C]}$ setup. The use of two GPU devices ($C_{[2G]}$) results in a 1.6-fold speedup over the setup with one single GPU ($C_{[1G]}$).

11.4.2 Pseudo-diagonalization

As mentioned in the previous section, pseudo-diagonalization will be ∼2× faster than the conventional diagonalization in a given SCF iteration. Thus PDIAG is used instead of FDIAG whenever possible. However, an efficient implementation of PDIAG on multiple GPUs can be challenging. Here, we first analyze the computations involved in pseudo-diagonalization, and then report the individual and overall speedups that have been achieved.

The details of pseudo-diagonalization have been described in the original paper [77]. From a computational point of view, it is basically comprised of two tasks. First, the Fock matrix is transformed from the AO basis to the MO basis by a triple matrix multiplication (FMO):

$$\mathbf{F}_{\mathrm{MO}} = \mathbf{C}_{\mathrm{o}}^{\mathrm{T}}\mathbf{F}\mathbf{C}_{\mathrm{v}},$$

where $\mathbf{C}_{\mathrm{o}}$ and $\mathbf{C}_{\mathrm{v}}$ denote the matrices of the occupied and virtual MO vectors, respectively. Then noniterative Jacobi-like 2×2 rotations (JACOBI) between pairs of occupied ($\mathbf{c}_{\mathrm{o}}$) and virtual ($\mathbf{c}_{\mathrm{v}}$) vectors are executed:

$$\mathbf{c}'_{\mathrm{o}} = a\mathbf{c}_{\mathrm{o}} - b\mathbf{c}_{\mathrm{v}} \qquad \text{and} \qquad \mathbf{c}'_{\mathrm{v}} = b\mathbf{c}_{\mathrm{o}} + a\mathbf{c}_{\mathrm{v}}, \tag{11.2}$$

where a and b are the elements of the rotation matrix, and the new MO vectors $\mathbf{c}'_o$ and $\mathbf{c}'_v$ are denoted by primes.

The profiles of the serial PDIAG version for the OM3 calculations on the proteins in our test set are given in Table 11.2. On average, FMO and JACOBI consume ∼45% and ∼55% of the CPU time, respectively. The other operations are negligible (<1%) and can be safely excluded from optimization.

Table 11.2 *Percentages (%) of computation time in the PDIAG subroutine consumed by FMO, JACOBI, and other tasks in the OM3 calculations on a single CPU core*

Notation	P_{020}	P_{063}	P_{086}	P_{100}	P_{125}	P_{156}	P_{166}	P_{221}
FMO	47.2	40.8	45.1	41.8	42.5	44.7	42.2	42.3
JACOBI	51.6	58.8	54.5	57.9	57.2	55.1	57.6	57.5
Others	1.3	0.5	0.4	0.3	0.2	0.2	0.2	0.2

Table 11.3 *Speedups of the FMO and JACOBI steps in the PDIAG subroutine on the multi-CPU $C_{[6C]}$, $C_{[12C]}$, and GPU-only $C_{[1G]}$ and $C_{[2G]}$ computing setups over the serial setup*

	FMO				JACOBI			
	$C_{[6C]}$	$C_{[12C]}$	$C_{[1G]}$	$C_{[2G]}$	$C_{[6C]}$	$C_{[12C]}$	$C_{[1G]}$	$C_{[2G]}$
P_{020}	5.2	7.8	5.9	5.2	3.4	5.1	2.6	3.6
P_{063}	5.7	10.2	16.2	18.6	1.6	1.6	4.4	7.9
P_{086}	5.8	10.6	19.6	22.4	1.4	1.4	4.6	8.6
P_{100}	5.8	10.7	20.0	23.1	1.3	1.2	4.5	8.1
P_{125}	5.8	10.8	20.3	25.4	1.2	1.2	5.0	9.4
P_{156}	5.8	11.3	21.0	30.4	1.2	1.2	4.4	8.6
P_{166}	5.8	11.3	20.6	31.9	1.2	1.2	4.4	8.6
P_{221}	5.5	10.6	20.8	32.9	1.4	1.4	5.1	9.9

The FMO step contains only the DGEMM calls for the matrix multiplications. The relevant speedups with different computing configurations are summarized in Table 11.3. Since DGEMM is compute-bound, FMO scales well with respect to the number of parallel processors in the CPU-only setups. One single GPU-accelerated FMO step can be as much as 20× faster than on one CPU core. The setup with two GPU devices may further increase the speedup to more than 30-fold, being about 1.6× faster than on $C_{[1G]}$. The best performance for a small protein like P_{020} is achieved with the CPU-only setup of 12 cores, however. This is because a GPU is designed for massively parallel tasks that a small system will not fully exploit, and some inevitable overhead such as CPU–GPU data transfer may hurt the overall performance of a smaller calculation.

The GPU-oriented optimization of the JACOBI step is demanding. The technical details can be found in our paper [39]. The resulting speedups are shown in Table 11.3. As one 2×2 rotation given in Eq. (11.2) involves six memory accesses (four reads and two writes) and six floating-point operations, the performance of JACOBI is fully determined by the memory bandwidth. In the case of P_{020}, the MO coefficient matrix is small enough (4.3 MiB) to completely fit into the CPU cache (12 MiB per chip). Modest speedups of 3.4 and 5.1 are therefore achieved on the $C_{[6C]}$ and $C_{[12C]}$ setups, respectively. On the other hand, numerous cache misses can occur for larger proteins starting from P_{063}. The performance on the CPU-only platform will then be determined entirely by the available memory bandwidth. The obtained speedup rapidly falls down to 1.2, no matter how many CPU cores are in use for parallelization. On the contrary, JACOBI on a single GPU benefits from the enhanced memory bandwidth (155 GB/s vs. 64 GB/s for two CPUs), and speedups of around 4.5-fold are consistently achieved in the benchmarks except for the smallest case, P_{020}. Addition of a second GPU doubles the total memory bandwidth, and the equal distribution of horizontal blocks of the coefficient matrix among the available devices enables the rotations to be carried out independently on each device (X. Wu, A. Koslowski, W. Thiel, unpublished results). The overall speedup on the $C_{[2G]}$ setup for P_{221} is 10, which is 1.9× higher than that on a single GPU ($C_{[1G]}$).

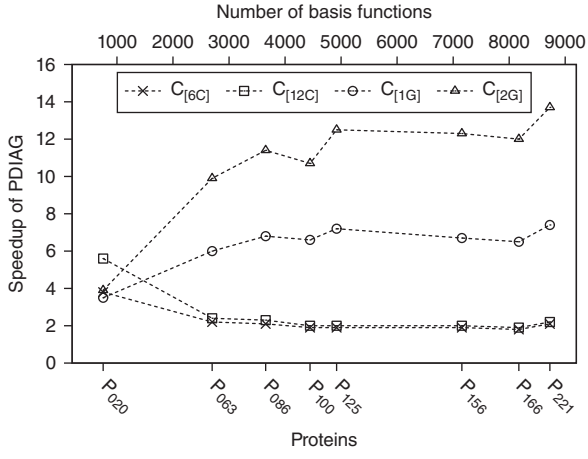


Figure 11.5 *Speedups of the PDIAG subroutine in the OM3 calculations on the multi-CPU $C_{[6C]}$, $C_{[12C]}$, and GPU-only $C_{[1G]}$ and $C_{[2G]}$ computing setups over the serial configuration*

Since the JACOBI step consumes a slightly higher fraction of the CPU time (between 55% and 60% for most proteins in our test set) than FMO in the PDIAG subroutine for the serial configuration, and since JACOBI benefits less from parallelization than PDIAG on all computing setups, the overall speedups of PDIAG shown in Figure 11.5 resemble those of JACOBI (see Table 11.3), but with some additional performance benefits from the FMO step. The highest speedup is 13.7 for P_{221} on $C_{[2G]}$, which is again 1.9× higher than that on a single GPU ($C_{[1G]}$).

11.4.3 Orthogonalization Corrections in OM3

The OM3 method [64] accounts for Pauli exchange repulsion by explicitly adding the orthogonalization corrections ($V^{\mathrm{ORT}}_{\mu_A\nu_B}$) to the core Hamiltonian of the Fock matrix:

$$V^{\mathrm{ORT}}_{\mu_A\nu_B} = -\frac{1}{2}G_1^{AB}\sum_{\lambda_C}(S_{\mu_A\lambda_C}\beta_{\lambda_C\nu_B} + \beta_{\mu_A\lambda_C}S_{\lambda_C\nu_B}) \qquad (\mathrm{C} \neq \mathrm{A} \text{ and } \mathrm{C} \neq \mathrm{B}),$$

where S and β denote elements of the overlap and resonance matrices, respectively, and G_1^{AB} is defined in terms of parameters that can be adjusted to fit reference data. μ_A, ν_B, and λ_C are AOs at atoms A, B, and C, respectively. If A and B are the same atom, $V^{\mathrm{ORT}}_{\mu_A\nu_B}$ is a correction to a one-center term; otherwise it refers to a two-center element. Inclusion of the latter three-center contributions leads to qualitative improvements over the MNDO-type methods for calculated molecular properties, such as rotational barriers, relative energies of isomers, hydrogen bonds, and vertical excitation energies [1, 65–67].

Even though the ORTCOR subroutine consumes only ∼1% of the wall clock time for the $C_{[1C]}$ setup, we implemented a dedicated algorithm utilizing multiple GPUs in an attempt to harness all available computing power. The ORTCOR performance for various setups is depicted in Figure 11.6. The technical details will be presented elsewhere.

The speedup of the ORTCOR subroutine scales reasonably well on the symmetric multi-CPU setups. For example, 5.5- and 10.1-fold performance boosts are feasible on the $C_{[6C]}$ and $C_{[12C]}$ setups, respectively. ORTCOR is accelerated up to 28-fold for medium-sized proteins like P_{063} on a single GPU ($C_{[1G]}$ setup), but thereafter the speedup decreases again with increasing system size to ∼20 for the largest proteins in our test set. The speedup on the $C_{[2G]}$ setup can reach 35-fold for a moderately

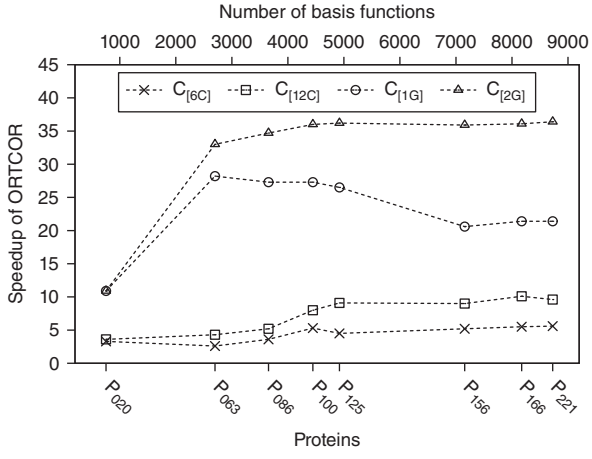


Figure 11.6 *Speedups of the ORTCOR subroutine in the OM3 calculations on the multi-CPU $C_{[6C]}$, $C_{[12C]}$, and GPU-only $C_{[1G]}$ and $C_{[2G]}$ computing setups over the serial configuration*

sized protein, and there is no performance deterioration for larger proteins. Moreover, the multi-GPU ORTCOR scales well compared to a single GPU device for sufficiently large proteins. For example, ORTCOR is 1.7× faster on $C_{[2G]}$ than on $C_{[1G]}$ for P_{221}.

11.5 Performance

Since a user will of course never run an individual subroutine by itself, the overall speedups for the OM3 calculations on proteins are more relevant in practice. They are presented in Figure 11.7.

The performance of the OM3 calculations on the CPU-only platform can hardly be improved by using more processor cores. The speedups quickly reach a saturation point and never exceed 4. The

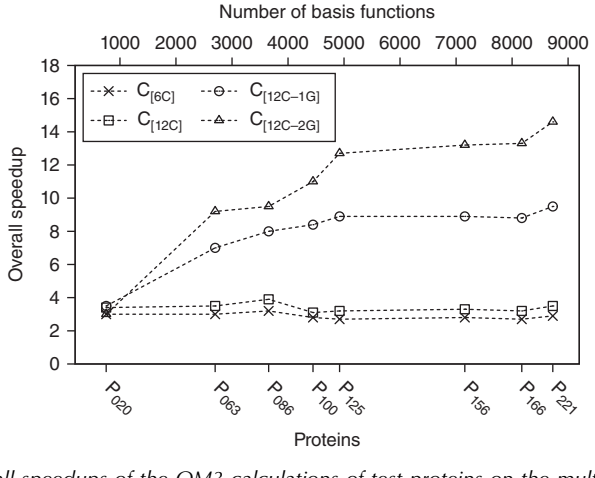


Figure 11.7 *Overall speedups of the OM3 calculations of test proteins on the multi-CPU $C_{[6C]}$, $C_{[12C]}$, and hybrid CPU–GPU $C_{[12C–1G]}$ and $C_{[12C–2G]}$ computing setups over the serial configuration*

mean values averaged over the proteins in the test set are 2.9 and 3.4 on $C_{[6C]}$ and $C_{[12C]}$, respectively. Moreover, the individual speedups seem to be almost invariant with respect to the size of the protein. Thus neither using more CPU cores nor increasing the system size yields higher speedups on the CPU-only setup. At first glance, this conclusion seems to contradict our previous result that the speedup of an MNDO calculation on fullerene C_{540} could reach 7.7 on Cray Y-MP with eight vector processors [84]. This apparent discrepancy can be resolved by considering the relevant arithmetic operations and the differences in the computer architectures. Concerning the computational bottlenecks mentioned in the preceding section, only three subroutines (BORDER, DIIS, and ORTCOR) of the five hotspots in the OM3 calculations can be well accelerated on current hardware by using additional CPU cores (see Figures 11.3, 11.4, and 11.6), whereas neither FDIAG nor PDIAG, which consume ~65% of the wall clock time, scale favorably with the number of cores (see Figures 11.2 and 11.5). This is because the former three are primarily dominated by compute-bound routines, which demand more arithmetic power than memory bandwidth. On the other hand, both diagonalization subroutines are composed of bandwidth-bound operations that would parallelize well on more CPU cores if and only if the demand for memory bandwidth could be satisfied in the first place. The theoretical floating-point peak performance of the two Xeon X5690 CPUs (a total of 166 GFlop/s) exceeds that of the Cray Y-MP (2.6 GFlop/s) by a factor of 64. The theoretical memory bandwidth of our current Xeon server (64 GB/s), however, is merely 2× greater than that of the 25-year-old Cray Y-MP (32 GB/s). Therefore, a tremendously inadequate memory bandwidth prevents the performance boost on a computer system including only parallel superscalar CPUs.

Because of the advantages of GPUs with regard to floating-point peak performance and memory bandwidth, the speedups achieved for the OM3 calculations on GPUs are monotonously growing with the size of the proteins and the number of GPUs (see Figure 11.7). Although the hybrid CPU–GPU platform provides higher speedups than the CPU-only platform for most bottlenecks, there may be exceptions in the case of calculations on small proteins like P_{020}. This may be due to the CPU–GPU communication overhead, to the unfavorable behavior of certain subroutines for small systems on a hybrid platform compared to a CPU-only setup (especially PDIAG, see Figure 11.5), or to the less optimized non-GPU routines becoming more dominant. For example, the CPU-only computation on P_{020} takes 41% of the wall clock time for the $C_{[12C-2G]}$ setup (see Figure 11.8, label "others"). Thus the overall performance of the OM3 calculations for P_{020} is rather similar on all computing setups. On the other hand, the acceleration on the hybrid CPU–GPU and CPU-only platforms is quite different for

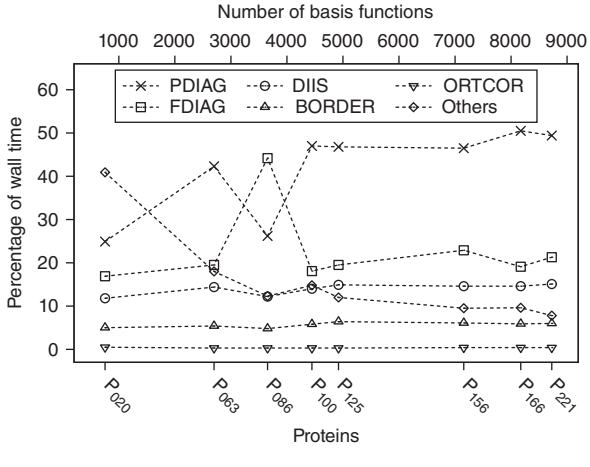


Figure 11.8 *Profiles of the OM3 calculations for the test proteins on the $C_{[12C–2G]}$ computing setup*

large calculations. The speedups of the OM3 calculations for P_{221} reach 9.5 and 14.6 on the $C_{[12C\text{-}1G]}$ and $C_{[12C\text{-}2G]}$ setups, respectively. The relative speedup of $C_{[12C\text{-}2G]}$ over $C_{[12C\text{-}1G]}$ is ~1.5 for the OM3 calculations of large proteins. Further performance increases are thus very likely when more GPU devices are employed in even larger semiempirical quantum chemical calculations.

Finally, we inspect the profiles of the OM3 calculations on the hybrid $C_{[12C\text{-}2G]}$ setup (see Figure 11.8). DIIS, BORDER, and ORTCOR are the three subroutines most accelerated on the GPU, thus their combined share of the wall clock time is just about half of that on $C_{[1C]}$. On average, the shares of DIIS, BORDER, and ORTCOR amount to 31.4%, 9.4%, 1.1% and 14.0%, 5.7%, 0.4% on the $C_{[1C]}$ and $C_{[12C\text{-}2G]}$ setups, respectively. The speedups for FDIAG and PDIAG are not as good as those for the former three routines, and hence their combined share on the $C_{[12C\text{-}2G]}$ setup is increased to 64.4% on average. The remaining subroutines (e.g., for integral evaluation and Fock matrix formation) have not yet been ported to a GPU, but are executed in parallel using multiple CPU cores (via OpenMP). They become the bottlenecks for small protein calculations with a time share of 40.9% in P_{020}, which gradually decreases with system size, down to 7.8% for a large protein like P_{221}. We may thus anticipate some further improvement of the overall performance with dedicated multi-GPU kernels for the semiempirical integral evaluation and Fock matrix construction.

11.6 Applications

Given the code developments outlined above, it has now become a routine task to carry out semiempirical quantum chemical calculations for large biomolecules, such as proteins, on a hybrid CPU–GPU computing platform. We have carried out full geometry optimizations of three proteins with α-helix, β-sheet, and random coil structures (see Figure 11.9), which were chosen from a collection of proteins used in previous work [85]. Six different semiempirical methods were applied, namely MNDO, AM1, PM3, and OMx (x = 1, 2, and 3). The optimizations were terminated when the gradient vector norm dropped below a preselected threshold value ($|\mathbf{g}| \leq 1.0$ kcal · mol^{-1} · Å^{-1}). The quality of the computed structures was assessed in terms of the conformation of the main chain by using the PROCHECK package [86], in comparison with the structures determined in aqueous solution by nuclear magnetic resonance (NMR) experiments. It should be stressed that the results given here are just for demonstration, since more realistic simulations would require more elaborate approches (e.g., including explicit solvent).

The backbones of the proteins are shown in Figure 11.9. Highly regular local structures imposed by hydrogen bonds are found in P_A and P_B, whereas P_C possesses an unfolded polypeptide chain. The backbone conformation of a protein is determined by a pair of less rigid dihedral angles $[\phi, \psi]$ at

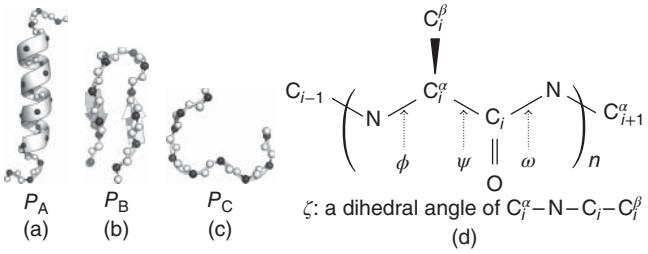


Figure 11.9 *Experimental structures of (a) P_A (PDB ID: 2AP7, 80% α-helix), (b) P_B (PDB ID: 2EVQ, 50% β-strands), and (c) P_C (PDB ID: 1LVR, 100% random coil). Only the backbone atoms are shown, with the C^α atoms represented by black balls. Four dihedral angles (ϕ, ψ, ω, and ζ) in a residue serve as stereochemical metrics, see the schematic sketch in (d)*

Table 11.4 *Statistics (%) for* $[\phi, \psi]$ *in the most favored* ($P^0_{\phi,\psi}$) *and additionally allowed* ($P^1_{\phi,\psi}$) *regions of the Ramachandran plot and standard deviations (°) of* ω *and* ζ

	P_A				P_B				P_C			
	$P^0_{\phi,\psi}$	$P^1_{\phi,\psi}$	σ_ω	σ_ζ	$P^0_{\phi,\psi}$	$P^1_{\phi,\psi}$	σ_ω	σ_ζ	$P^0_{\phi,\psi}$	$P^1_{\phi,\psi}$	σ_ω	σ_ζ
Expt.	100.0	0.0	0.8	0.6	87.5	12.5	2.6	0.7	42.9	42.9	2.3	1.8
MNDO	91.7	8.3	9.0	1.3	87.5	12.5	14.9	0.5	28.6	57.1	17.0	2.5
AM1	75.0	16.7	9.7	1.0	100.0	0.0	15.8	1.0	28.6	71.4	7.0	1.9
PM3	75.0	25.0	15.3	1.3	87.5	12.5	19.7	1.0	42.9	57.1	22.8	1.7
OM1	81.8	18.2	12.4	1.0	87.5	12.5	9.4	0.9	28.6	71.4	12.5	2.5
OM2	83.3	16.7	8.6	1.1	87.5	12.5	10.4	1.1	42.9	42.9	10.5	2.1
OM3	83.3	16.7	7.6	1.1	87.5	12.5	15.7	1.2	42.9	42.9	14.3	2.5

Results for the experimental structures of P_A, P_B, and P_C are compared with those calculated by semiempirical quantum chemical methods.

the C^α-atom [87] and a stiff torsion angle ω of the peptide bond. ω is usually restricted to be around 180° for an energetically more favorable trans conformer due to the partial double bond character of the amide bond, which prevents facile rotation. In addition, a virtual dihedral angle ζ is defined between C^α_i–N and noncovalently bound $C_i \cdots C^\beta_i$ as a measure of chirality at the central C^α_i atom of the amino acid [88].

PROCHECK divides a Ramachandran map into four regions: most favored, additionally allowed, generously allowed, and disallowed. The shares of the first two distributions, $P^0_{\phi,\psi}$ and $P^1_{\phi,\psi}$, for P_A, P_B, and P_C are listed in Table 11.4. In most cases, $P^0_{\phi,\psi}$ and $P^1_{\phi,\psi}$ add up to the total population. Neither experimental nor theoretically optimized protein structures are spoiled by disallowed $[\phi, \psi]$ combinations. Since more regular secondary structures exist in P_A and P_B than in the disordered P_C, significantly higher values for $P^0_{\phi,\psi}$ are obtained for the former two proteins. MNDO, AM1, PM3, and OM1 predict a higher $[\phi, \psi]$ population in the additionally allowed region for P_C, whereas OM2, OM3, and the NMR experiment give equal values for $P^0_{\phi,\psi}$ and $P^1_{\phi,\psi}$. Although the deficiencies of the original MNDO method for the description of hydrogen bonds are known from early studies [89, 90], its actual performance for the proteins in the test set seems rather satisfactory. Both the α-helix (in P_A) and β-strand (in P_B) structures are found, and reasonable $[\phi, \psi]$ distributions are retained in the optimized structures.

All semiempirical methods predict greater deviations from planarity around the peptide bond than deduced from experiment (see the σ_ω values). Such deviations from planarity in the peptide group have already been reported in earlier theoretical studies [7, 91, 92]: the sp^2-hybrid nitrogen in a peptide bond should be planar, but it tends to be pyramidalized in semiempirical calculations. The average value of ζ for L-amino acids is $33.81 \pm 4.17°$ [88]. The σ_ζ values from experiment and from semiempirical calculations are rather small and of similar quality, indicating a good description of the local environment of the sp^3-C^α_i atoms in the main chains.

11.7 Conclusion

In this chapter, we have presented a profile-guided optimization of the semiempirical quantum chemical MNDO program on a hybrid CPU–GPU platform. OM3 calculations on a set of eight proteins were used to guide the code development and to assess the performance. The computational bottlenecks on one single CPU core were identified as the diagonalization of the Fock matrix (FDIAG), fast pseudodiagonalization (PDIAG), SCF acceleration (DIIS), density matrix formation (BORDER), and computation of the orthogonalization corrections in OM3 (ORTCOR), which cover altogether ∼99% of the wall clock time in the test runs. Standard library routines and special finely tuned kernels

targeting multiple GPU devices were employed to accelerate these routines, whereas the relevant remaining subroutines (∼1% of the computation time) were run in parallel using multiple CPU cores (via OpenMP) to achieve optimum performance on the hybrid CPU–GPU platform.

We have identified severe restraints to parallelize the semiempirical calculations on currently available CPU-only computing architectures. No matter how many processor cores are utilized in a calculation, a ceiling of the overall acceleration is reached rapidly because of the limitations imposed by the hardware memory bandwidth. On the other hand, the speedup of the calculations on the hybrid CPU–GPU platform rises continuously with increasing system size and reaches one order of magnitude in large protein calculations. The overall performance can be further improved through the use of multiple GPUs.

As an illustrative application, geometry optimizations of three typical proteins with α-helix, β-sheet, and random coil structures were carried out by means of the MNDO, AM1, PM3, and OMx ($x = 1$, 2, and 3) methods. These calculations produced qualitatively reasonable conformations of the main chains (with regard to the usual metrics for assessing protein backbone structures) but showed some deviation from experiment by giving slightly nonplanar peptide bonds. We are confident that such quantitative deficiencies can be ameliorated in future semiempirical method development. This will enhance the impact of the current code development work on hybrid CPU–GPU platforms, which has enabled semiempirical quantum chemical calculations on large systems such as proteins with thousands of atoms.

Acknowledgement

This work was supported by an ERC Advanced Grant (OMSQC).

References

1. Thiel, W. (2014) Semiempirical quantum-chemical methods. *WIREs Comput. Mol. Sci.*, **4**, 145–157.
2. Hehre, W.J., Radom, L., Schleyer, P.v.R. and Pople, J.A. (1986) *Ab Initio Molecular Orbital Theory*, John Wiley & Sons, Inc., New York.
3. Pople, J.A., Santry, D.P. and Segal, G.A. (1965) Approximate self-consistent molecular orbital theory. I. Invariant procedures. *J. Chem. Phys.*, **43**, S129–S135.
4. Parr, R.G. and Yang, W. (1989) *Density-Functional Theory of Atoms and Molecules*, Oxford University Press, USA.
5. Thiel, W. (1996) Perspectives on semiempirical molecular orbital theory. *Adv. Chem. Phys.*, **93**, 703–757.
6. Ehresmann, B., de Groot, M.J., Alex, A. and Clark, T. (2004) New molecular descriptors based on local properties at the molecular surface and a boiling-point model derived from them. *J. Chem. Inf. Comput. Sci.*, **44**, 658–668.
7. Stewart, J.J.P. (2009) Application of the PM6 method to modeling proteins. *J. Mol. Model.*, **15**, 765–805.
8. Wu, X., Thiel, W., Pezeshki, S. and Lin, H. (2013) Specific reaction path hamiltonian for proton transfer in water: reparameterized semiempirical models. *J. Chem. Theory Comput.*, **9**, 2672–2686.
9. Fabiano, E., Lan, Z., Lu, Y. and Thiel, W. (2011) Nonadiabatic trajectory calculations with Ab initio and semiempirical methods, in *Conical Intersections: Theory, Computation and Experiment* (eds W. Domcke, D.R. Yarkony and H. Koppel), World Scientific Publishing Company, pp. 463–496.
10. Pople, J.A. (2003) Quantum chemical models, in *Nobel Lectures in Chemistry (1996–2000)* (ed. I. Grenthe), World Scientific Publishing Company, Singapore, pp. 246–260.

11. Thiel, W. and Green, D.G. (1995) The MNDO94 code: parallelization of a semiempirical qauntum-chemical program, in *Methods and Techniques in Computational Chemistry: METECC-95* (eds E. Clementi and G. Corongiu), STEF, Cagliari, pp. 141–168.
12. Kirk, D.B. and Hwu, W.M.W. (2012) *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann Publishers.
13. Stone, J.E., Hardy, D.J., Ufimtsev, I.S. and Schulten, K. (2010) GPU-accelerated molecular modeling coming of age. *J. Mol. Graph. Model.*, **29**, 116–125.
14. Farber, R.M. (2011) Topical perspective on massive threading and parallelism. *J. Mol. Graph. Model.*, **30**, 82–89.
15. Anderson, A., Goddard, W.A. III and Schröder, P. (2007) Quantum Monte Carlo on graphical processing units. *Comput. Phys. Commun.*, **177**, 298–306.
16. Kim, J., Rodgers, J.M., Athènes, M. and Smit, B. (2011) Molecular Monte Carlo simulations using graphics processing units: to waste recycle or not? *J. Chem. Theory Comput.*, **7**, 3208–3222.
17. Yasuda, K. (2008) Two-electron integral evaluation on the graphics processor unit. *J. Comput. Chem.*, **29**, 334–342.
18. Ufimtsev, I.S. and Martínez, T.J. (2008) Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation. *J. Chem. Theory Comput.*, **4**, 222–231.
19. Asadchev, A., Allada, V., Felder, J., Bode, B.M., Gordon, M.S. and Windus, T.L. (2010) Uncontracted Rys quadrature implementation of up to G functions on graphical processing units. *J. Chem. Theory Comput.*, **6**, 696–704.
20. Wilkinson, K.A., Sherwood, P., Guest, M.F. and Naidoo, K.J. (2011) Acceleration of the GAMESS-UK electronic structure package on graphical processing units. *J. Comput. Chem.*, **32**, 2313–2318.
21. Miao, Y. and Merz, K.M. (2013) Acceleration of electron repulsion integral evaluation on graphics processing units via use of recurrence relations. *J. Chem. Theory Comput.*, **9**, 965–976.
22. Titov, A.V., Ufimtsev, I.S., Luehr, N. and Martinez, T.J. (2013) Generating efficient quantum chemistry codes for novel architectures. *J. Chem. Theory Comput.*, **9**, 213–221.
23. Yasuda, K. (2008) Accelerating density functional calculations with graphics processing unit. *J. Chem. Theory Comput.*, **4**, 1230–1236.
24. Ufimtsev, I.S. and Martínez, T.J. (2009) Quantum chemistry on graphical processing units. 2. Direct self-consistent-field implementation. *J. Chem. Theory Comput.*, **5**, 1004–1015.
25. Ufimtsev, I.S. and Martínez, T.J. (2009) Quantum chemistry on graphical processing units. 3. Analytical energy gradients, geometry optimization, and first principles molecular dynamics. *J. Chem. Theory Comput.*, **5**, 2619–2628.
26. Ufimtsev, I.S. and Martínez, T.J. (2008) Graphical processing units for quantum chemistry. *Comput. Sci. Eng.*, **10**, 26–34.
27. Genovese, L., Ospici, M., Deutsch, T., Méhaut, J.-F., Neelov, A. and Goedecker, S. (2009) Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *J. Chem. Phys.*, **131**, 034103 (8 pages).
28. Luehr, N., Ufimtsev, I.S. and Martínez, T.J. (2011) Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs). *J. Chem. Theory Comput.*, **7**, 949–954.
29. Andrade, X. and Genovese, L. (2012) Harnessing the power of graphic processing units, in *Fundamentals of Time-Dependent Density Functional Theory*, Lecture Notes in Physics, vol. **837** (eds M.A. Marques, N.T. Maitra, F.M. Nogueira, E. Gross and A. Rubio), Springer-Verlag, pp. 401–413.
30. Andrade, X. and Aspuru-Guzik, A. (2013) Real-space density functional theory on graphical processing units: computational approach and comparison to Gaussian basis set methods. *J. Chem. Theory Comput.*, **9**, 4360–4373.

31. Isborn, C.M., Luehr, N., Ufimtsev, I.S. and Martínez, T.J. (2011) Excited-state electronic structure with configuration interaction singles and Tamm-Dancoff time-dependent density functional theory on graphical processing units. *J. Chem. Theory Comput.*, **7**, 1814–1823.
32. Vogt, L., Olivares-Amaya, R., Kermes, S., Shao, Y., Amador-Bedolla, C. and Aspuru-Guzik, A. (2008) Accelerating resolution-of-the-identity second-order Møller-Plesset quantum chemistry calculations with graphical processing units. *J. Phys. Chem. A*, **112**, 2049–2057.
33. Olivares-Amaya, R., Watson, M.A., Edgar, R.G., Vogt, L., Shao, Y. and Aspuru-Guzik, A. (2010) Accelerating correlated quantum chemistry calculations using graphical processing units and a mixed precision matrix multiplication library. *J. Chem. Theory Comput.*, **6**, 135–144.
34. Watson, M., Olivares-Amaya, R., Edgar, R.G. and Aspuru-Guzik, A. (2010) Accelerating correlated quantum chemistry calculations using graphical processing units. *Comput. Sci. Eng.*, **12**, 40–51.
35. DePrince, A.E. and Hammond, J.R. (2011) Coupled cluster theory on graphics processing units I. The coupled cluster doubles method. *J. Chem. Theory Comput.*, **7**, 1287–1295.
36. Ma, W., Krishnamoorthy, S., Villa, O. and Kowalski, K. (2011) GPU-based implementations of the noniterative regularized-CCSD(T) corrections: applications to strongly correlated systems. *J. Chem. Theory Comput.*, **7**, 1316–1327.
37. Bhaskaran-Nair, K., Ma, W., Krishnamoorthy, S., Villa, O., van Dam, H.J.J., Aprà, E. and Kowalski, K. (2013) Noniterative multireference coupled cluster methods on heterogeneous CPU-GPU systems. *J. Chem. Theory Comput.*, **9**, 1949–1957.
38. Asadchev, A. and Gordon, M.S. (2013) Fast and flexible coupled cluster implementation. *J. Chem. Theory Comput.*, **9**, 3385–3392.
39. Wu, X., Koslowski, A. and Thiel, W. (2012) Semiempirical quantum chemical calculations accelerated on a hybrid multicore CPU-GPU computing platform. *J. Chem. Theory Comput.*, **8**, 2272–2281.
40. Maia, J.D.C., Urquiza Carvalho, G.A., Mangueira, C.P., Santana, S.R., Cabral, L.A.F. and Rocha, G.B. (2012) GPU linear algebra libraries and GPGPU programming for accelerating MOPAC semiempirical quantum chemistry calculations. *J. Chem. Theory Comput.*, **8**, 3072–3081.
41. Dewar, M.J.S. (1969) *The Molecular Orbital Theory of Organic Chemistry*, McGraw-Hill Series in Advanced Chemistry, McGraw-Hill.
42. Pople, J.A. and Beveridge, D.L. (1970) *Approximate Molecular Orbital Theory*, McGraw-Hill Series in Advanced Chemistry, McGraw-Hill.
43. Murrell, J.N. and Harget, A.J. (1972) *Semi-Empirical Self-Consistent-Field Molecular Orbital Theory of Molecules*, Wiley-Interscience.
44. Thiel, W. (1988) Semiempirical methods: current status and perspectives. *Tetrahedron*, **44**, 7393–7408.
45. Thiel, W. (1997) Computational methods for large molecules. *J. Mol. Struct. THEOCHEM*, **398–399**, 1–6.
46. Thiel, W. (2000) Semiempirical methods, in *Modern Methods and Algorithms of Quantum Chemistry Proceedings*, 2nd edn (ed. J. Grotendorst), John von Neumann Institute for Computing, Jülich, pp. 261–283.
47. Thiel, W. (2005) Semiempirical quantum-chemical methods in computational chemistry, in *Theory and Applications of Computational Chemistry: The First Forty Years* (eds C.E. Dykstra, G. Frenking, K.S. Kim and G.E. Scuseria), Elsevier, Amsterdam, pp. 559–580.
48. Stewart, J.J.P. (1990) Semiempirical molecular orbital methods, in *Reviews in Computational Chemistry* (eds K.B. Lipkowitz and D.B. Boyd), John Wiley & Sons, Inc., pp. 45–81.
49. Zerner, M.C. (1991) Semiempirical molecular orbital methods, in *Reviews in Computational Chemistry* (eds K.B. Lipkowitz and D.B. Boyd), John Wiley & Sons, Inc., pp. 313–365.
50. Stewart, J.J.P. (1990) MOPAC: a semiempirical molecular orbital program. *J. Comput.-Aided Mol. Des.*, **4**, 1–103.

51. Parr, R.G. (1952) A method for estimating electronic repulsion integrals over LCAO MO's in complex unsaturated molecules. *J. Chem. Phys.*, **20**, 1499.
52. Pople, J.A. (1953) Electron interaction in unsaturated hydrocarbons. *Trans. Faraday Soc.*, **49**, 1375–1385.
53. Dewar, M.J.S. and Thiel, W. (1977) A semiempirical model for the two-center repulsion integrals in the NDDO approximation. *Theor. Chim. Acta*, **46**, 89–104.
54. Dewar, M.J.S. and Thiel, W. (1977) Ground states of molecules. 38. The MNDO method. Approximations and parameters. *J. Am. Chem. Soc.*, **99**, 4899–4907.
55. Dewar, M.J.S. and Thiel, W. (1977) Ground states of molecules. 39. MNDO results for molecules containing hydrogen, carbon, nitrogen, and oxygen. *J. Am. Chem. Soc.*, **99**, 4907–4917.
56. Dewar, M.J.S., Zoebisch, E.G., Healy, E.F. and Stewart, J.J.P. (1985) Development and use of quantum mechanical molecular models. 76. AM1: a new general purpose quantum mechanical molecular model. *J. Am. Chem. Soc.*, **107**, 3902–3909.
57. Stewart, J.J.P. (1989) Optimization of parameters for semiempirical methods I. Method. *J. Comput. Chem.*, **10**, 209–220.
58. Stewart, J.J.P. (2004) Comparison of the accuracy of semiempirical and some DFT methods for predicting heats of formation. *J. Mol. Model.*, **10**, 6–12.
59. Stewart, J.J.P. (2007) Optimization of parameters for semiempirical methods V: modification of NDDO approximations and application to 70 elements. *J. Mol. Model.*, **13**, 1173–1213.
60. Stewart, J.J.P. (2013) Optimization of parameters for semiempirical methods VI: more modifications to the NDDO approximations and re-optimization of parameters. *J. Mol. Model.*, **19**, 1–32.
61. Repasky, M.P., Chandrasekhar, J. and Jorgensen, W.L. (2002) PDDG/PM3 and PDDG/MNDO: improved semiempirical methods. *J. Comput. Chem.*, **23**, 1601–1622.
62. Kolb, M. and Thiel, W. (1993) Beyond the MNDO model: methodical considerations and numerical results. *J. Comput. Chem.*, **14**, 775–789.
63. Weber, W. and Thiel, W. (2000) Orthogonalization corrections for semiempirical methods. *Theor. Chem. Acc.*, **103**, 495–506.
64. Scholten, M. (2003) Semiempirische Verfahren mit Orthogonalisierungskorrekturen: Die OM3 Methode. PhD thesis. Universität Düsseldorf, Düsseldorf.
65. Otte, N., Scholten, M. and Thiel, W. (2007) Looking at self-consistent-charge density functional tight binding from a semiempirical perspective. *J. Phys. Chem. A*, **111**, 5751–5755.
66. Korth, M. and Thiel, W. (2011) Benchmarking semiempirical methods for thermochemistry, kinetics, and noncovalent interactions: OM*x* methods are almost as accurate and robust as DFT-GGA methods for organic molecules. *J. Chem. Theory Comput.*, **7**, 2929–2936.
67. Silva-Junior, M.R. and Thiel, W. (2010) Benchmark of electronically excited states for semiempirical methods: MNDO, AM1, PM3, OM1, OM2, OM3, INDO/S, and INDO/S2. *J. Chem. Theory Comput.*, **6**, 1546–1564.
68. Gargaro, A.R., Bloomberg, G.B., Dempsey, C.E., Murray, M. and Tanner, M.J.A. (1994) The solution structures of the first and second transmembrane-spanning segments of band 3. *Eur. J. Biochem.*, **221**, 445–454, PDB ID: 1BTQ.
69. O'Neill, J.W., Kim, D.E., Johnsen, K., Baker, D. and Zhang, K.Y. (2001) Single-site mutations induce 3D domain swapping in the B1 domain of protein L from Peptostreptococcus magnus. *Structure*, **9**, 1017–1027, PDB ID: 1K50.
70. Rubini, M., Lepthien, S., Golbik, R. and Budisa, N. (2006) Aminotryptophan-containing barstar: structure-function tradeoff in protein design and engineering with an expanded genetic code. *Biochim. Biophys. Acta*, **1764**, 1147–1158, PDB ID: 2HXX.
71. Ciatto, C., Bahna, F., Zampieri, N., VanSteenhouse, H.C., Katsamba, P.S., Ahlsen, G., Harrison, O.J., Brasch, J., Jin, X., Posy, S., Vendome, J., Ranscht, B., Jessell, T.M., Honig, B. and Shapiro, L. (2010) T-cadherin structures reveal a novel adhesive binding mechanism. *Nat. Struct. Mol. Biol.*, **17**, 339–347, PDB ID: 3K6F.

72. Fedorov, A.A., Magnus, K.A., Graupe, M.H., Lattman, E.E., Pollard, T.D. and Almo, S.C. (1994) X-ray structures of isoforms of the actin-binding protein profilin that differ in their affinity for phosphatidylinositol phosphates. *Proc. Natl. Acad. Sci. U. S. A.*, **91**, 8636–8640, PDB ID: 1ACF.
73. Choi, J., Choi, S., Chon, J.K., Choi, J., Cha, M.-K., Kim, I.-H. and Shin, W. (2005) Crystal structure of the C107S/C112S mutant of yeast nuclear 2-Cys peroxiredoxin. *Proteins*, **61**, 1146–1149, PDB ID: 2A4V.
74. Vaaje-Kolstad, G., Bøhle, L.A., Gåseidnes, S., Dalhus, B., Bjørås, M., Mathiesen, G. and Eijsink, V.G. (2012) Characterization of the chitinolytic machinery of Enterococcus faecalis V583 and high-resolution structure of its oxidative CBM33 enzyme. *J. Mol. Biol.*, **416**, 239–254, PDB ID: 4A02.
75. Tsukazaki, T., Mori, H., Echizen, Y., Ishitani, R., Fukai, S., Tanaka, T., Perederina, A., Vassylyev, D.G., Kohno, T., Maturana, A.D., Ito, K. and Nureki, O. (2011) Structure and function of a membrane component SecDF that enhances protein export. *Nature*, **474**, 235–238, PDB ID: 3AQO.
76. Pulay, P. (1982) Improved SCF convergence acceleration. *J. Comput. Chem.*, **3**, 556–560.
77. Stewart, J.J.P., Császár, P. and Pulay, P. (1982) Fast semiempirical calculations. *J. Comput. Chem.*, **3**, 227–228.
78. Thiel, W. (2012) MNDO99 CVS Development Version. Tech Rep, Mülheim an der Ruhr, Germany.
79. Dongarra, J., Dong, T., Gates, M., Haidar, A., Tomov, S. and Yamazaki, I. (2012) MAGMA: A new generation of linear algebra libraries for GPU and multicore architectures.
80. Häser, M. and Ahlrichs, R. (1989) Improvements on the direct SCF method. *J. Comput. Chem.*, **10**, 104–111.
81. Haidar, A., Solcà, R., Gates, M., Tomov, S., Schulthess, T. and Dongarra, J. (2013) Leading edge hybrid multi-GPU algorithms for generalized eigenproblems in electronic structure calculations, in *Supercomputing*, Lecture Notes in Computer Science, vol. **7905** (eds J. Kunkel, T. Ludwig and H. Meuer), Springer-Verlag, Berlin, Heidelberg, pp. 67–80.
82. Rohr, D., Bach, M., Kretz, M. and Lindenstruth, V. (2011) Multi-GPU DGEMM and high performance Linpack on highly energy-efficient clusters. *IEEE Micro*, **31**, 18–27.
83. Spiga, F. and Girotto, I. (2012) phiGEMM: a CPU-GPU library for porting quantum ESPRESSO on hybrid systems. Proceeding of 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2012), pp. 368–375.
84. Bakowies, D. and Thiel, W. (1991) MNDO study of large carbon clusters. *J. Am. Chem. Soc.*, **113**, 3704–3714.
85. Kulik, H.J., Luehr, N., Ufimtsev, I.S. and Martínez, T.J. (2012) *Ab initio* quantum chemistry for protein structures. *J. Phys. Chem. B*, **116**, 12501–12509.
86. Laskowski, R.A., MacArthur, M.W., Moss, D.S. and Thornton, J.M. (1993) PROCHECK: a program to check the stereochemical quality of protein structures. *J. Appl. Crystallogr.*, **26**, 283–291.
87. Ramachandran, G.N., Ramakrishnan, C. and Sasisekharan, V. (1963) Stereochemistry of polypeptide chain configurations. *J. Mol. Biol.*, **7**, 95–99.
88. Morris, A.L., MacArthur, M.W., Hutchinson, E.G. and Thornton, J.M. (1992) Stereochemical quality of protein structure coordinates. *Proteins*, **12**, 345–364.
89. Burstein, K.Y. and Isaev, A.N. (1984) MNDO calculations on hydrogen bonds. Modified function for core-core repulsion. *Theor. Chim. Acta*, **64**, 397–401.
90. Goldblum, A. (1987) Improvement of the hydrogen bonding correction to MNDO for calculations of biochemical interest. *J. Comput. Chem.*, **8**, 835–849.
91. Möhle, K., Hofmann, H.-J. and Thiel, W. (2001) Description of peptide and protein secondary structures employing semiempirical methods. *J. Comput. Chem.*, **22**, 509–520.
92. Seabra, G.de.M., Walker, R.C. and Roitberg, A.E. (2009) Are current semiempirical methods better than force fields? A study from the thermodynamics perspective. *J. Phys. Chem. A*, **113**, 11938–11948.

12

GPU Acceleration of Second-Order Møller–Plesset Perturbation Theory with Resolution of Identity

Roberto Olivares-Amaya[1], *Adrian Jinich*[2], *Mark A. Watson*[1] *and Alán Aspuru-Guzik*[2]
[1]*Department of Chemistry, Princeton University, Princeton, NJ, USA*
[2]*Department of Chemistry and Chemical Biology, Harvard University, Cambridge, MA, USA*

Second order Møller–Plesset perturbation (MP2) theory [1] is a widely used and one of the computationally least expensive post-SCF correlated treatments for electronic structure calculations. In this chapter we review methods used to reduce the computational expense of MP2 calculations for larger systems and then highlight efforts to GPU-accelerate one such method termed resolution-of-the-identity MP2. The theoretical background of the approach is discussed, followed by the specifics of how RI-MP2 was adapted to GPUs. Discussion will focus on matrix algebra optimizations, in particular the use of mixed precision in matrix multiplications. The computational performance of the approach is evaluated and discussed with respect to mathematical precision, hardware, and molecule size. Finally, example applications to biomolecules are considered.

12.1 Møller–Plesset Perturbation Theory with Resolution of Identity Approximation (RI-MP2)

One of the most widely used and computationally least expensive correlated treatments for electronic structure is MP2 theory [1]. MP2 is known to produce equilibrium geometries of comparable accuracy to density functional theory (DFT) [2], but, unlike many popular DFT functionals, is able to capture long-range correlation effects such as the dispersion interaction. For many weakly bound systems where DFT results are unreliable, MP2 is essentially the least expensive and most reliable alternative [3].

Electronic Structure Calculations on Graphics Processing Units:
From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

The MP2 method has been useful in the study of biochemistry, such as in the study of larger biomolecules [4] and the study of dispersion energy in protein folding [5]. It has also found use in the generation of force fields for metal-organic frameworks [6] and, more recently, in the study of interactions in organic photovoltaic materials [7]. Recently, the use of MP2 in extended systems [8] has yielded interesting results in the study of adsorption energies [9].

Several methods have been applied and developed to extend the applicability of MP2 to larger systems. In this chapter, we will be focusing on the resolution-of-the-identity (RI) methods, also known as RI [10–13]. However, several other successful methods have also been developed, including local-MP2 [14], Cholesky decomposition (CD) [15, 16], and divide-and-conquer (DC) methods [17, 18].

The RI approximation reduces the number of operations by at least an order of magnitude for a triple-zeta basis set calculation. In RI-MP2, a larger basis set entails a larger relative speedup [19]. Therefore, RI-MP2 enables both the study of larger systems and also a more accurate representation, as it allows treatments of larger basis sets. Several steps in this method are matrix–matrix multiplications, which allow for a facile translation into the single-instruction multiple-data (SIMD) calculation paradigm, and hence to be further accelerated with graphics processing units (GPUs).

We begin describing the Hartree–Fock (HF) ansatz (guess) by using the electronic Hamiltonian under the Born–Oppenheimer (BO) approximation [20]:

$$\begin{aligned}\hat{H}_{\text{el}} &= -\frac{1}{2}\sum_{i}^{N}\nabla_i^2 - \sum_{i}^{N}\sum_{A}^{M}\frac{Z_A}{\mathrm{r}_{iA}} + \sum_{i}^{N}\sum_{j>i}^{N}\frac{1}{\mathrm{r}_{ij}} \\ &= \sum_{i}^{N}\hat{h}_i + \sum_{i}^{N}\sum_{j>i}^{N}\frac{1}{\mathrm{r}_{ij}},\end{aligned} \tag{12.1}$$

for a system with N electrons and M atoms, where $\hat{h}_i = -\frac{1}{2}\nabla_i^2 - \sum_A \frac{Z_A}{\mathrm{r}_{iA}}$.

The HF method approximates the wave function as a product of molecular orbitals (MOs) ϕ_i, which are wave functions for a single electron under a one-electron Hamiltonian, the Fock operator:

$$\hat{f}_i = \hat{h}_i + \sum_{j=1}^{N/2}[2\hat{J}_j - \hat{K}_j], \tag{12.2}$$

where $\hat{J}$ and $\hat{K}$ represent the Coulomb and exchange operators, respectively [21]. The associated HF equations $\hat{f}\psi_i = \epsilon_i\psi_i$ yield a complete set of eigenfunctions. This set of equations is typically solved using the Roothaan–Hall self-consistent field equations. The method scales cubically with system size: $\mathcal{O}(M^3)$ [22].

As mentioned above and in Chapter 3, post-HF methods are able to systematically recover the electronic energy. Multi-configurational methods are able to obtain the so-called static correlation, as the wave function ansatz requires more than a single Slater determinant to properly describe its ground state. Dynamic correlation arises from the instantaneous Coulombic repulsion. As HF is a mean-field, single-determinant method, it is unable to capture either of these terms. MP2 theory is able to obtain some of the dynamic correlation, which makes it an efficient method to obtain quantitative predictions for systems close to their equilibrium structures.

Perturbation theory methods partition the Hamiltonian $\hat{H} = H_0 + V$. In MP2 theory, the electronic Hamiltonian is divided as

$$\hat{H} = \sum_i \hat{f}_i + \hat{\Phi}, \tag{12.3}$$

where $\hat{\Phi}$ represents the perturbation potential:

$$\hat{\Phi} = \hat{H} - \sum_i \hat{f}_i.$$

Deriving MP2 theory reveals that first-order solution yields the HF energy: $E_{\mathrm{HF}} = E^0 + E^{(1)}$. The expression for computing the second-order energy is

$$E^{(2)} = \sum_{ijab} \frac{(ia|jb)^2 + \frac{1}{2}[(ia|jb) - (ib|ja)]^2}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b} \tag{12.4}$$

in terms of the eigenfunctions of the Fock operator with eigenvalues ϵ, where the occupied MOs are labeled by i, j; the virtual MOs are labeled by a, b; and the MO integrals

$$(ij|ab) = \sum_{\mu\nu\lambda\sigma} C_{\mu i} C_{\nu j} C_{\lambda a} C_{\sigma b} (\mu\nu|\lambda\sigma) \tag{12.5}$$

are obtained by transforming the atomic orbital (AO) electron repulsion integrals (ERIs)

$$(\mu\nu|\lambda\sigma) = \int\int \frac{\phi_\mu(\mathrm{r}_1)\phi_\nu(\mathrm{r}_1)\phi_\lambda(\mathrm{r}_2)\phi_\sigma(\mathrm{r}_2)}{|\mathrm{r}_1 - \mathrm{r}_2|} \mathrm{d}\mathrm{r}_1 \mathrm{d}\mathrm{r}_2, \tag{12.6}$$

where $C_{\mu i}$ represents the matrix elements of MO coefficients describing the expansion of each MO as a linear combination of AOs.

There have been efforts to reduce the prefactor in MP2 at a very small cost to accuracy. The two-center, four-index ERI in Eq. (12.6) becomes highly linearly dependent with increasing AO basis set size, and so it is possible to expand the products of electrons 1 and 2 in a basis set of auxiliary functions P [19]:

$$\rho_{\mu\nu}(\mathrm{r}) = \eta_\mu(\mathrm{r})\eta_\nu(\mathrm{r}) \approx \tilde{\rho}_{\mu\nu}(\mathrm{r}) = \sum_P C_{\mu\nu,P} P(\mathrm{r})$$

with a dimension smaller than the original product space. If one minimizes the error in the Coulomb ERIs, then it is possible to approximate Eq. (12.6) as only two- and three-index quantities:

$$\widetilde{(\mu\nu|\lambda\sigma)} = \sum_{P,Q} (\mu\nu|P)(P|Q)^{-1}(Q|\lambda\sigma). \tag{12.7}$$

This method is called RI because of the insertion of the following term [10, 11]:

$$I = \sum_m |m)(m| \approx \sum_{P,Q} |P)(P|Q)^{-1}(Q|,$$

but it has also been referred to as a density-fitting scheme [23]. We can now obtain the approximate $(ia|jb)$ integrals using matrix multiplications for RI-MP2:

$$\widetilde{(ia|jb)} \approx \sum_Q B_{ia,Q} B_{jb,Q}, \tag{12.8}$$

$$B_{ia,Q} = \sum_P (ia|P)(P|Q)^{-1/2}. \tag{12.9}$$

The advantage of the RI method lies in reducing the $\mathcal{O}(N^4)$dependence with respect to the AOs to $\mathcal{O}(N^3)$. This enables the use of this method for larger basis sets. The cost with respect to molecular size still remains at $\mathcal{O}(N^5)$. However, the prefactor is reduced by an order of magnitude, and most of the operations can be cast as efficient matrix multiplications [23].

We accelerated the RI-MP2 method, principally by noticing that Eqs. (12.8)–(12.9) can be calculated more efficiently using the stream-processing paradigm with GPUs. The hurdles of GPU programming were overcome using CUDA and its basic linear algebra subprograms implementation CUBLAS [24]. As discussed in Chapter 2, one of the main issues in designing an efficient GPU implementation is minimizing the CPU–GPU communication. Therefore, the algorithm included a method to batch the matrices up to fill the available GPU memory, since acceleration increases as the size of the matrix grows. Since memory in GPU cards is typically smaller than CPU RAM, we also devised a matrix cleaver to enable a full RI-MP2 implementation for GPUs.

12.1.1 Cleaving General Matrix Multiplies (GEMMs)

Consider the matrix multiplication

$$C = A \cdot B, \tag{12.10}$$

where A is an $(m \times k)$ matrix and B is an $(k \times n)$ matrix, making C an $(m \times n)$ matrix. We can divide A into a column vector of $r + 1$ matrices

$$A = \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_r \end{pmatrix}, \tag{12.11}$$

where each entry A_i is a $(p_i \times k)$ matrix, and $\sum_{i=0}^{r} p_i = m$. In practice, all the p_i will be the same, with the possible exception of p_r, which will be an edge case. In a similar manner, we can divide B into a row vector of $s + 1$ matrices

$$B = \begin{pmatrix} B_0 & B_1 & \dots & B_s \end{pmatrix}, \tag{12.12}$$

where each B_j is an $(k \times q_j)$ matrix and $\sum_{j=0}^{s} q_j = n$. Again, all the q_j will be the same, with the possible exception of q_s. We then form the outer product of these two vectors

$$C = \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_r \end{pmatrix} \cdot \begin{pmatrix} B_0 & B_1 & \dots & B_s \end{pmatrix}, \tag{12.13}$$

$$= \begin{pmatrix} A_0 \cdot B_0 & A_0 \cdot B_1 & \dots & A_0 \cdot B_s \\ A_1 \cdot B_0 & A_1 \cdot B_1 & & A_1 \cdot B_s \\ \vdots & & \ddots & \\ A_r \cdot B_0 & & & A_r \cdot B_s \end{pmatrix}. \tag{12.14}$$

Each individual $C_{ij} = A_i B_j$ is a $(p_i \times q_j)$ matrix, and can be computed independently of all the others. Generalizing this to a full `*GEMM` implementation (i.e., `DGEMM` or `SGEMM`), which includes the possibility of transposes being taken, is tedious but straightforward.

We have implemented this approach on GPUs as a complete replacement for `*GEMM`. The p_i and q_j values are chosen such that each sub-multiplication fits within the currently available GPU memory. Each multiplication is staged through the GPU, and the results assembled on the CPU. This process is hidden from the user code, which simply sees a standard `*GEMM` call.

12.1.2 Other MP2 Approaches

Here we briefly detail the local MP2 (LMP2) method [14, 25, 26] and its RI counterpart [23] as an example of a method that reduces the scaling of MP2 by using a set of localized MOs. The virtual space is spanned by a basis of nonorthogonal orbitals obtained by projecting out the occupied orbital space. These are named projected atomic orbitals (PAOs). One can then effectively group the excitations from pairs of the localized MOs that are spatially close to pairs of the PAOs. This scheme reduces the number of excitations for each pair to be proportional to the square of the number of occupied pairs: $N^2_{[ij]}$. Integrals from orbitals that are distant can be either neglected or calculated using multipole expansions. In this sense, Werner *et al.* [23] developed a density-fitting LMP2 method to reduce the scaling in the RI-MP2 steps and achieve linear scaling with system size. In this method, the bottleneck for large molecules becomes matrix multiplications, and an approach similar to what was developed for RI-MP2 becomes ideal to further accelerate the calculations.

Another effort to approach the study of large-molecule calculations using MP2 is the DC method [17, 27]. In the DC-MP2 method, the correlation energy of the system is evaluated by summing up the correlation energies of the subsystems such that finding a correct partitioning becomes fundamental. Particularly, Katouda and Nakajima [18, 28] present a DC-MP2 approach where they parallelize the method into a coarse-grain approach, where they assign each subsystem to a group of processors, and a fine-grain approach where the MP2 correlation energy can be calculated. Parallel GPU approaches could become useful in enhancing these techniques, either by exploiting the SIMD paradigm or by using multiple graphics cards.

We finish this section by mentioning that other methods have been implemented for GPUs, such as coupled cluster [29–31] and DFT [32], as presented in detail in the other chapters of this book. The matrix multiplication approach has been extended to post-HF methods such as coupled-cluster [30], as well as phenomenological hierarchical equations of motion [33]. Other methods, such as the matrix–matrix multiplication approach by Hanrath and Engels-Putzka [34] could be greatly enhanced by the approach we currently use in RI-MP2. In the latter, 87% of the calculations are `DGEMM` operations.

12.2 A Mixed-Precision Matrix Multiplication Library

Because of the initial constraints of GPU architectures and, in general, owing to a desire to speed up calculations, there has been a renewed desire to exploit single-precision (SP) arithmetic. GPUs with double-precision (DP) support are available, although DP operations are still typically 2–8×slower than SP operations. Moreover, one would like to be able to exploit additional resources by using robust algorithms for a wide array of device architectures [35–37]. These include methods such as mixed precision for ERIs [38–40], RI-MP2 [41, 42] and for coupled-cluster doubles [30]. On the other hand, Vysotskiy and Cederbaum have shown that it is possible to obtain accurate quantum chemistry in single precision using the CD [43].

In this section, we will provide an overview of the different quantum chemistry techniques that have recently been developed for mixed and single precision. Mixed-precision methods rely on a way to effectively separate elements that should be calculated in double precision, so as to not lose accuracy, and those that can be in single precision, to gain speedup.

Early on, Yasuda [38] implemented the evaluation of ERIs on GPUs. He noticed that, roughly, terms in the range $[10^{-n}, 10^{-n+1}]$ give an energy error of $10^{-n}\sqrt{N(n)}$, where $N(n)$ is the number of terms with exponent n. Studying a histogram of $N(n)$ against n revealed that most of the numerical error was due to the largest terms evaluated in single precision. To divide between large and small terms, Yasuda used the density-weighted ($P_{\mu\nu}$) Schwarz bound

$$|(\mu\nu|\lambda\sigma)||P_{\lambda\sigma}| \leq (\mu\nu|\mu\nu)^{1/2}(\lambda\sigma|\lambda\sigma)^{1/2}|P_{\lambda\sigma}| \tag{12.15}$$

to separate the ERIs given a threshold λ_{GPU}.

In a similar manner, Luehr *et al.* [39] used Eq. (12.15) to dynamically control the precision for each SCF iteration. The metric was controlled by taking the maximum element of the direct inversion in the iterative subspace (DIIS) error vector and using a power-law fit of the error given a threshold

$$\text{Err}(\text{Thre}) = 2.0 \times 10^{-6}\text{Thre}^{0.7},$$

such that for a given error (Err), one can find a threshold (Thre). As the SCF iterations increase, it then becomes feasible to obtain DP results using a minimum number of DP operations. Specifically, the number of SCF iterations did not grow for the molecules Luehr *et al.* tested. The precision error did not exceed the μE_h regime.

The partition strategy of using the Schwarz inequality has also been used outside the GPU community. Vysotskiy and Cederbaum studied the effect of using single precision to obtain MP2 correlation using the CD [43].

Interestingly, the distribution of the true errors

$$|f(\widetilde{ai|bj}) - (\widetilde{ai|bj})|, \tag{12.16}$$

where $f(\cdot)$ indicates the ERI is calculated in SP and the tilde indicates the ERIs are determined using CD [15, 16], is of the order of 10^{-8}–10^{-14} and peaks at 10^{-11}, while the evaluation of the right-hand side shows a peak at 10^{-6}. The authors empirically found that the Cholesky vectors are of the same magnitude and are largely free from large components. The error in single precision becomes negligible as they showed for a relatively large system (taxol with the ANO-L-VDZP basis).

The roundoff error is an important issue in determining the numerical stability of algorithms. The additional two algorithms developed below explore this issue. Ultimately, when using limited-precision arithmetic, the associativity law of addition does not hold since the summation order becomes relevant when summing up large numbers and small numbers. In the case of SP, the precision is limited to about seven digits. Smaller numbers will quickly lose their accuracy when being added or subtracted with relatively larger ones. Methods to determine the numerical stability of algorithms have been developed and have especially been targeted to explore correlation methods in quantum chemistry [44].

We originally used pure SP cards, as those were the only ones available at the time [45]. The error of our RI-MP2 implementation with these cards naturally increased as we tested with larger systems, until eventually the error was beyond 1 kcal/mol (i.e., chemical accuracy). The large errors led to proposing methods of mixed precision [41, 42]. As mentioned above, the main problem for using single precision is that, when performing a product between a large and a small number, the 6–7 significant figures are often insufficient to achieve chemical accuracy. We explored two schemes of partitioning matrix elements so that they could be calculated using SP matrix multiplication (`SGEMM` in BLAS library) in the GPU.

The first method is *bitwise partitioning*. Consider splitting a DP floating-point number $A = m * 2^k$,

$$A \approx A^u + A^l, \tag{12.17}$$

where A^u and A^l are SP numbers storing the uppermost n_u and the next lowest n_l significant bits of m, respectively. Then, if we apply the multiplication of two scalars A, B using bitwise partitioning, we can approximate the full DP multiplication as four (or three for expediency) SP multiplications:

$$\begin{aligned} AB &\approx A^u B^u + A^u B^l + A^l B^u + A^l B^l \\ AB &\approx A^u B^u + A^u B^l + A^l (B^u + B^l), \end{aligned} \tag{12.18}$$

where in the last term we have used the SP cast of B, where we do not consider the error to be of a different order of magnitude compared to $A^l B^u$. We can generalize Eq. (12.18) for matrix multiplication:

$$\mathbf{AB} \approx \mathbf{A}^u \mathbf{B}^u + \mathbf{A}^u \mathbf{B}^l + \mathbf{A}^l (\mathbf{B}^u + \mathbf{B}^l), \tag{12.19}$$

where we can use Eq. (12.17) for each element of $\mathbf{X} \in \{\mathbf{A}, \mathbf{B}\}$.

All the multiplications may be evaluated efficiently using the CUBLAS `SGEMM` library routines on the GPU. The results may then be accumulated in DP on the CPU to yield the final approximation for $\mathbf{AB}$. We must also consider the round-off error due to multiply–add operations (Figure 12.1). That is, for matrix $\mathbf{A}$, consisting of $M \times K$ elements, multiplied by matrix $\mathbf{B}$, consisting of $K \times N$, there are $M \times N$ dot products of length K. This will affect the precision of Eq. (12.19).

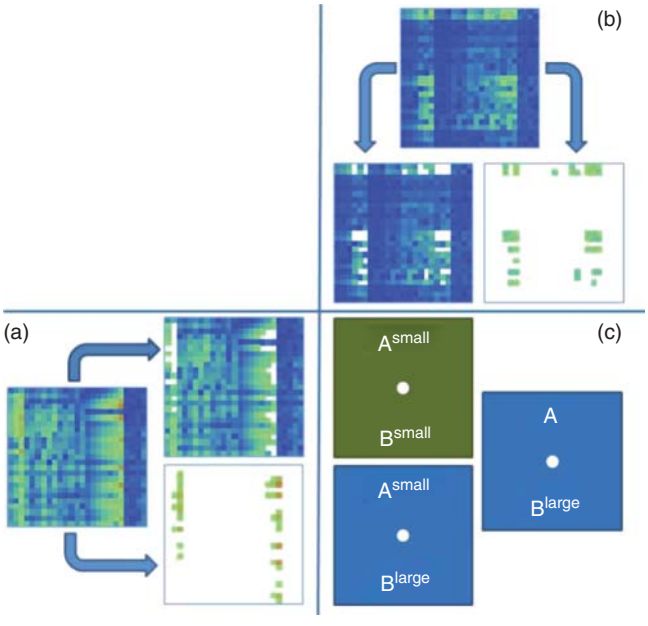


Figure 12.1 *Pictorial representation of the heterogeneous* `MGEMM` *algorithm. (a, b)* **A** *and* **B** *matrices and their separation into matrices with large and small elements, given a cutoff parameter* δ*. (c) Components of the final* **AB** *matrix, where the light gray (green in ebook) component involves a dense matrix multiplication computed with CUBLAS-SGEMM, while the dark grey (blue in ebook) components involve sparse matrix multiplications computed with a* `DAXPY`*-like algorithm (explained in the text). The blocks follow the nomenclature shown in Eq. (12.20)*

While "bitwise enhancement" could show promising results since the acceleration of this method depends only on the size of the matrices, it is only 3 times faster than CPU dgemm in the best case scenario. It is possible to obtain an enhancement in accuracy (measured as the quotient of the root-mean-squared (RMS) deviation between `SGEMM` over `MGEMM`). However, this is effective only when all the matrix elements are of the same order or magnitude [42].

We implemented a heterogeneous algorithm as our second approach (see Figure 12.1). It involves separating the matrix multiplication $\mathbf{C} = \mathbf{AB}$, by splitting **A** and **B** into "large" and "small" components, giving

$$\begin{aligned}\mathbf{C} &= (\mathbf{A}^{\text{large}} + \mathbf{A}^{\text{small}})(\mathbf{B}^{\text{large}} + \mathbf{B}^{\text{small}}) \\ &= \mathbf{AB}^{\text{large}} + \mathbf{A}^{\text{large}}\mathbf{B}^{\text{small}} + \mathbf{A}^{\text{small}}\mathbf{B}^{\text{small}},\end{aligned} \tag{12.20}$$

where we have taken the simple approach of introducing a cutoff value δ to define the split. That is, if $|X_{ij}| > \delta$, the element is considered "large"; otherwise it is considered "small." The $\mathbf{A}^{\text{small}}\mathbf{B}^{\text{small}}$ term consists entirely of "small" numbers, and can be run with reasonable accuracy in single precision on the GPU. The strategy of summing together small elements has also been used for CD approaches [43], since most of the elements of the CD are small and therefore there is no significant loss of accuracy when using SP.

The other two terms in Eq. (12.20) contain "large" numbers, and need to be calculated in double precision to achieve greater accuracy. Since each of the "large" matrices will often be sparse, these terms each consist of a dense-sparse multiplication (i.e., similar to `DAXPY`). Therefore, we only store the nonzero elements for the "large" matrices.

We present benchmark calculations for matrices and quantum chemistry results for this method in the following section.

12.3 Performance of Accelerated RI-MP2

12.3.1 Matrix Benchmarks

As we are interested in accelerating quantum chemistry, the matrices calculated in RI-MP2 are of large size N and contain both large and small elements that are separated by orders of magnitude. Therefore, in the case of these methods, the heterogeneous approach seemed to be most effective to tackle mixed precision.

Our calculations were performed with an Intel Xeon E5472 (Harpertown) CPU clocked at 3.0 GHz attached to an NVIDIA Tesla C1060 (packaged into a Tesla S1070). The GPU calls were limited to 256 MB of RAM to model a more restricted GPU in a typical BOINC (Berkeley Open Infrastructure for Network Computing) client [46–49].

We benchmark the heterogeneous `MGEMM` algorithm described in Section 12.2. Figure 12.2 shows the speedup of `*GEMM` calls on the GPU with respect to the size N of an $N \times N$ matrix, relative to the time taken for the corresponding `DGEMM` call on the CPU in serial.

In order to test the mixed-precision approach, we used matrices with random values in the range $[-1, 1]$. These were then "salted" with a fraction f_{salt} of values 2 orders of magnitude larger, in the range of $[90, 110]$. We tested three different salted fractions: $f_{\text{salt}} = 10^{-2}$, 10^{-3}, and 10^{-4}. The size of the cutoff parameter δ was chosen such that all the salted elements were considered "large." All timings were averaged over 10 runs.

Running CUBLAS `SGEMM` is approximately 17.1 times faster than running `DGEMM` on the CPU for a matrix of size $10,048 \times 10,048$. This represents an upper bound for the speedups we can hope to

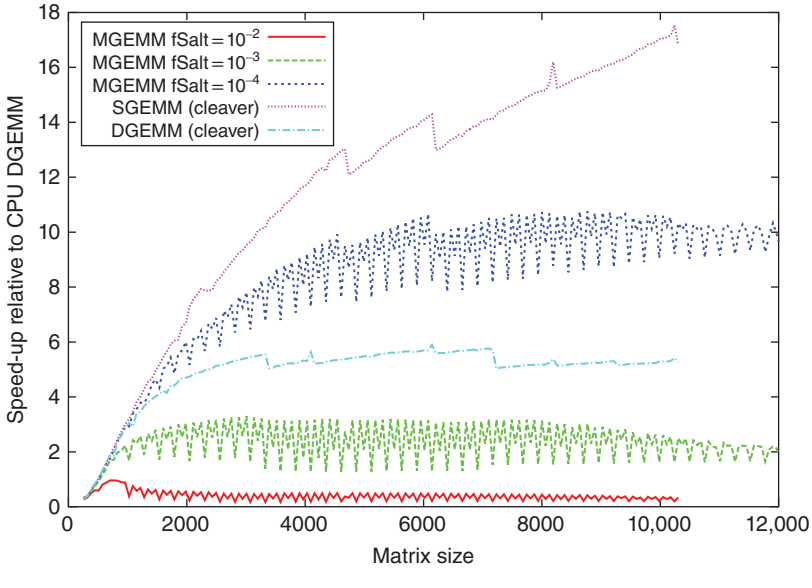


Figure 12.2 *Speedup for various `*GEMM` calls as a function of matrix size. Most elements were in the range* $[-1, 1]$*, with the "salt" values in the range* $[90, 110]$*. Times are scaled relative to running `DGEMM` on the CPU*

obtain with MGEMM for such matrices. Leveraging the GPU for small matrices is not effective because of well-known overheads such as memory transfer and access latencies.

The MGEMM speedups are strongly dependent on f_{salt}, which determines how much of the calculation is done in double precision on the CPU. For $f_{\text{salt}} = 10^{-4}$, the speedups are approximately 10×, but for $f_{\text{salt}} = 10^{-3}$ the speedups decrease fivefold (2×) relative to CPU DGEMM. As we increase f_{salt} to 10^{-2}, MGEMM becomes slower than CPU DGEMM.

Since both single and double precision are now available in GPUs, it would be possible to perform the three matrix multiplications of Eq. (12.20) within the GPU. As two of these matrix multiplications are sparse, and not performed as DGEMM, the potential performance gain may be limited. Nonetheless, performing all operations on the GPU could help reduce overheads.

Next we consider the accuracy enhancement when using MGEMM compared to SGEMM. We study the RMS errors of each matrix element relative to CPU DGEMM for different matrix sizes. All the matrices were initialized with uniform random values in the range $[-1, 1]$. We tested separately two different ranges for the salting values grouped into two ranges: $[90, 110]$ and $[9990, 10,010]$, and also considering various salting fractions.

We show the results in Figure 12.3. The SGEMM result shows an error around 3 orders of magnitude larger than the other MGEMM tests. In general, the error progressively increases as the matrix size becomes larger. However, the errors are the same regardless of the fraction or size of the salted elements. These trends are also reflected when studying the maximum absolute error [41].

Additionally, the limiting MGEMM errors are identical to the SGEMM errors for a pair of *unsalted* random matrices on $[-1, 1]$ because the MGEMM algorithm guarantees that all the salted contributions are computed on the CPU. Indeed, if the salts were larger or more numerous, the SGEMM errors would be even larger, but the MGEMM errors would be unchanged. This is in contrast to the behavior of the bitwise MGEMM algorithm [42].

To study the effect of MGEMM on matrices related to quantum chemistry, we use those of taxol in a cc-pVDZ basis. The precise fractions of large and small elements for the taxol case are plotted in Figure 12.4 with varying the cutoff parameter δ for both the steps 3 and 4 matrices. We should

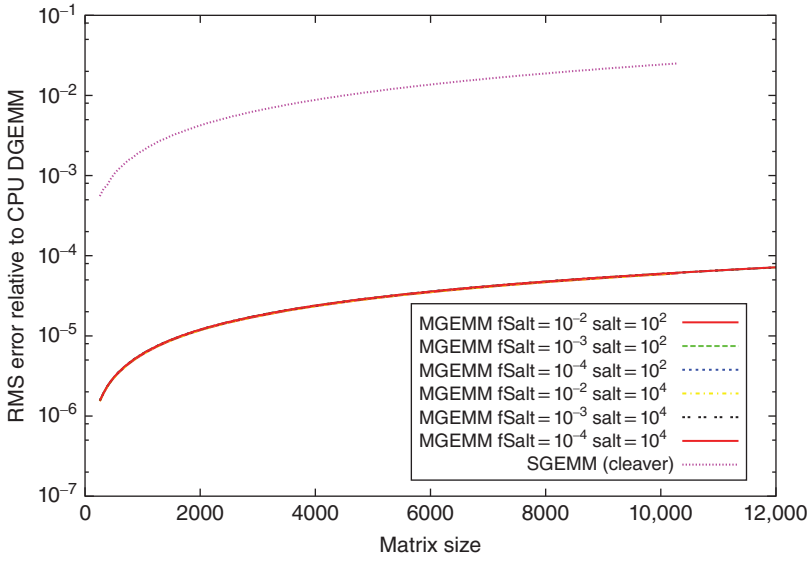


Figure 12.3 *RMS error in a single matrix element for various GEMM calls as a function of matrix size compared to CPU DGEMM. Background elements were in the range* $[-1, 1]$*, with the "salt" values in the range* $[90, 110]$ *or* $[9990, 10,010]$*. MGEMM gives identical results for all parameters*

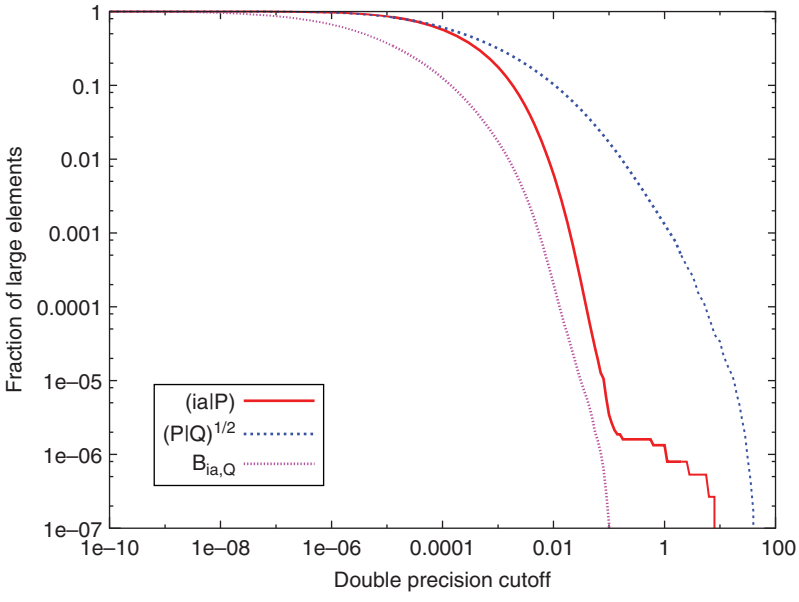


Figure 12.4 *Fraction of "large" elements as a function of the cutoff parameter δ for the taxol RI-MP2 matrices in steps 3 and 4 of the algorithm outlined in Section 12.1*

note that these curves are only for one particular batch, as explained in Section 12.1, and not the full matrices. We chose this batch to have the most conservative matrices for our plots, that is, those that had large elements across the broadest range of δ values.

It is significant that the step 3 matrices have a greater fraction of large elements than the step 4 matrices, and, specifically, the $(P|Q)^{-1/2}$ matrix has the largest elements of all. This means that for a constant δ value, we can expect `MGEMM` to introduce larger errors in the step 3 matrix multiplications than in step 4.

In this test case, we notice a continuous decay of the magnitude of matrix element values across many orders of magnitude. In the model matrices, the elements were randomly chosen to peak around two different values, so the distribution would resemble a step function. In Figure 12.2, `MGEMM` was seen to outperform `DGEMM` for a fraction of salts of order 10^{-4}. Comparing to 12.4, this suggests that δ should be greater than 0.01 to ensure significant `MGEMM` speedups when considering the $(ia|P)$ and $B_{ia,Q}$ matrices, while the fraction of large elements in the $(P|Q)^{-1/2}$ matrices becomes this small only for δ values of order 10.

An MGEMM study of these matrices reveals that in all cases the maximum errors are only of order 10^{-6} in the worst case, and there is only a modest decrease in accuracy of `MGEMM` with respect to `DGEMM`. We can estimate an upper bound on the error of each element for different δ values from Figure 12.3. Since the matrix dimension is approximately 4000, the choice $\delta = 0.1$ would give a conservative error bound of approximately $4000 \times 10^{-6} \times 0.1$, which is of order 10^{-4}. However, because the matrices do not have a "constant background" of 0.1, this estimate is very conservative.

In the current implementation, the main issue affecting the efficiency of `MGEMM` is the ratio of large to small elements in the input matrices, but in general we can also expect the sparsity structure to impact performance. In cases where the structure is known in advance, a more specialized treatment could give worthwhile speedups, but this is beyond the scope of this chapter. Moreover, it could be advantageous to define a more dynamic δ value for different steps in an algorithm, or even for different input matrices in a manner similar to the work of Luehr *et al.* [39].

12.3.2 RI-MP2 Benchmarks

In this section, we present speedups relative to the original RI-MP2 implementation [41, 42].

For the original benchmarks, we used an AMD Athlon 5600+ CPU clocked at 2.8 GHz, combined with an NVIDIA Tesla C1060 GPU with 4 GiB of RAM. For some calculations, the GPU was limited to 256 MB of RAM, as described below. RI-MP2 calculations were performed using a modified version of Q-Chem 3.1. The more recent benchmarks were performed using an Intel Xeon(R) CPU E5-2650 processor clocked at 2.0 GHz attached to an NVIDIA Tesla K20m and using its full memory capabilities with Q-Chem 4.1, which already ships with the GPU-RI-MP2 algorithm.

For our test systems we chose a set of linear alkanes (C_8H_{18}, $C_{16}H_{34}$, $C_{24}H_{50}$, $C_{32}H_{66}$, $C_{40}H_{82}$), as well as two molecules of pharmaceutical interest: the anticancer drugs taxol ($C_{47}H_{51}NO_{14}$) and valinomycin ($C_{54}H_{90}N_6O_{18}$). We used the cc-pVDZ (double-ζ) and cc-pVTZ (triple-ζ) [50] AO basis sets throughout.

In the more recent set of benchmarks (results shown in Table 12.3), we continue studying molecules of biological relevance. To connect with the past work, we study taxol and valinomycin, and extend our work to β-cyclodextrin, ATP + 20 H_2O, and NADH + 20 H_2O.

In Table 12.1, we benchmark the reference case of using either CUBLAS `SGEMM` or `DGEMM` for each test molecule using the double-ζ basis set. The table shows the speedup in computing the RI-MP2 correlation energy and the error relative to a standard serial CPU calculation (the `DGEMM` errors are negligible). The speedups and `SGEMM` errors are greater for the larger molecules, with the largest speedups observed for valinomycin: 13.8× and 7.8×, using `SGEMM` and `DGEMM`, respectively. However, while CUBLAS `DGEMM` gives essentially no loss of accuracy, the `SGEMM` error is approximately −10.0 kcal/mol, which is well beyond what is generally accepted as chemical accuracy.

Quantum chemistry generally aims to achieve a target accuracy of 1.0 kcal/mol. In Table 12.2, we explore the performance of `MGEMM` using a constant cutoff value of $\delta = 1.0$ to try and reduce the `SGEMM` errors in Table 12.1. The results show speedups and total energy errors for each molecule in both the double-ζ and triple-ζ basis sets. In this particular case, we have limited the GPU to use only 256 MB of RAM to mimic the capability of older cards and emphasize the use of the `MGEMM` cleaver. This will naturally result in a loss of speedup compared to utilizing a larger GPU memory. In the case of taxol, the reduction is approximately 20%.

The trends in Table 12.2 are the same as in Table 12.1, but the `MGEMM` errors are approximately an order of magnitude less than the `SGEMM` errors for the larger molecules. For valinomycin in the double-ζ basis, the `SGEMM` speedup is reduced from 13.8× to 10.1× using `MGEMM`, but the error in the total energy is also reduced from −10.0 to −1.2 kcal/mol, which is now very close to chemical accuracy. It could be further reduced by choosing a δ value more appropriate for this system.

Table 12.1 *Speedups using CUBLAS `SGEMM` and `DGEMM` and total energy errors relative to CPU `DGEMM` for various molecules in a double-ζ basis*

	Speedup		`SGEMM` energy error
Molecule	`SGEMM`	`DGEMM`	(kcal/mol)
C_8H_{18}	2.1	1.9	−0.05616
$C_{16}H_{34}$	4.5	3.7	−0.12113
$C_{24}H_{50}$	6.9	5.2	−0.62661
$C_{32}H_{66}$	9.0	6.4	−0.75981
$C_{40}H_{82}$	11.1	7.2	−1.12150
Taxol	11.3	7.1	−6.26276
Valinomycin	13.8	7.8	−9.99340

Table 12.2 *MGEMM speedups and total energy errors with respect to CPU* `DGEMM` *for various molecules in a double-ζ and triple-ζ basis*

	Speedup		Energy error (kcal/mol)	
Molecule	Double-ζ	Triple-ζ	Double-ζ	Triple-ζ
C_8H_{18}	1.9	2.7	−0.01249	−0.03488
$C_{16}H_{34}$	3.8	5.6	−0.00704	−0.04209
$C_{24}H_{50}$	5.8	8.2	−0.14011	−0.33553
$C_{32}H_{66}$	7.9	9.2	−0.08111	−0.29447
$C_{40}H_{82}$	9.4	10.0	−0.13713	−0.51186
Taxol	9.3	10.0	−0.50110	−1.80076
Valinomycin	10.1	—	−1.16363	—

While CUBLAS `DGEMM` clearly has the advantage of high accuracy, if −1.2 kcal/mol is deemed an acceptable accuracy, `MGEMM` could be favored since the `DGEMM` speedup is only 7.8 × compared to 10.1×. This becomes especially relevant in the case where one is interested in energy differences instead of absolute energies. The errors are larger in triple-ζ due to a greater error accumulation since there are simply more computations. This error propagates similarly for larger molecules (and smaller basis sets).

12.4 Example Applications

In this section, we study large biomolecules using RI-MP2 as well as an initial study of metabolic reactions. While taxol (an anticancer drug) and valinomycin (a peptide antibiotic) have long been used as benchmarks of large molecules in GPU implementations [45, 51, 52], we expand the study of biomolecules by adding hydrated ATP and NAD^+, as well as β-cyclodextrin, an oligosaccharide.

12.4.1 Large-Molecule Applications

Cyclodextrins are cyclic oligosaccharides composed of different numbers of linked α-D-glucose units. Their hydrophobic cavities promote the formation of complexes with several compounds. This has made cyclodextrins useful for a range of applications in the chemical industry as well as a model system to study nonconvalent interactions and model enzyme–substrate interactions [53].

The metabolic cofactor molecules ATP and NAD^+ play a fundamental role in cellular metabolism. They are key players in phosphorylation and redox reactions, respectively. Simulations of these cofactors through computational chemistry can yield important insight into their physicochemical properties. For example, recent work has explored the mechanism of ATP hydrolysis through *ab initio* molecular dynamics simulations [54]. In addition, the diversity of geometric conformations of ATP in solution has been explored through MD simulations [55]. The electronic structure and conformational features of NAD^+, the oxidized form of the NADH cofactor, has been explored through DFT and MP2 calculations [56, 57]. Additionally, first-principles thermochemical properties obtained under different solvent conditions such as pH and cation concentrations would be useful in predicting the Gibbs reaction energies of metabolic reactions that involve these cofactors [58].

We analyzed the performance of our single-point energy estimates of ATP and NAD^+ embedded in an explicit water solvent shell obtained from our RI-MP2 GPU implementation. Each metabolite was surrounded by a cluster of 20 explicit waters.

In Table 12.3, we include an update of the benchmarks using the most recent Q-Chem release. The most recent CPU implementation of Q-Chem involves multithreading, so the speedups differ

Table 12.3 *Speedups using CUBLAS* `DGEMM` *relative to CPU* `DGEMM` *for biomolecules in a double-ζ basis*

Molecule	Number of atoms	Double-ζ
ATP + 20-H_2O	91	4.3
Taxol	113	6.0
NADH+ 20-H_2O	131	6.4
Valinomycin	168	7.6
β-Cyclodextrin	147	7.8

from the results presented above. We find that in the case of valinomycin at double-ζ, the speedup is smaller (7.6×) than on previous hardware (10.1×). On the other hand, it is now possible to perform calculations for larger systems (both in basis set and in molecule size).

12.4.2 Studying Thermodynamic Reactivity

As a further application of our RI-MP2 GPU implementation, we focused on the thermochemistry of metabolic reactions. There has been a recent surge in interest in the study of the thermodynamics of metabolism [59]. In the context of biochemistry, a deep understanding of thermodynamics is vital in analyzing the design principles of natural metabolic pathways, as well as in engineering efficient, novel pathways [60, 61].

Surprisingly, accurate standard Gibbs reaction energy values exist for only a fraction of the biochemical reactions that sustain life [62]. First-principles quantum chemical estimates of metabolic reaction thermodynamics represent a promising avenue to fill important gaps in experimental databases. Since metabolites exist in solution as ensembles of protonation states, *ab initio* methods must accurately capture the electronic structure of charged species. Toward this direction, recent work has obtained accuracies with quantum chemistry comparable to group contribution methods for isomerization reactions [63]. However, these DFT-based methods suffer from inaccuracy when dealing with highly charged protonation states.

Using accelerated RI-MP2 methods for metabolic thermochemistry can potentially enable accurate treatment of a large range of sizes of metabolites with diverse protonation states. Since metabolites are modeled as ensembles of numerous protonation states, isomers, and geometric conformers, the speedup obtained with accelerated methods is necessary to cover all of metabolism at reasonable computational cost.

We explore the increase in accuracy of reaction Gibbs energy estimates obtained by including RI-MP2 single-point electronic energies in our computational framework. Our estimated Gibbs reaction energies are compared against experimental values from perhaps the most complete database of enzymatic reaction thermochemistry at specified pH and temperature, namely the NIST-TECRDB [62].

Our procedure to obtain first-principle Gibbs reaction energies is as follows: We compute reaction Gibbs free energies from differences of absolute Gibbs energies of individual metabolites in solution. Each metabolite is represented by an ensemble of protonation states (microspecies) that exist at equilibrium concentrations at a given pH. Each protonation state is in turn represented by an ensemble of geometric conformations (conformers). We initially sample microspecies and conformers using empirical rules as implemented in Marvin Calculator Plugins (Marvin 5.12, 2013, ChemAxon http://www.chemaxon.com). We use the software Packmol [64] to randomly place explicit water molecules around the metabolite [65]. We then perform geometry optimization and normal mode analysis of these diverse initial conformers using the quantum chemistry software Orca [66]. Specifically, we employ DFT with the B3LYP functional [67–70] and the 6-31G* [71–73]

basis. Enthalpies and entropies of each conformer are then obtained through standard equations from molecular statistical thermodynamics [74]. The absolute Gibbs energies of conformers are Boltzmann-averaged to obtain the Gibbs energy of a single protonation state. We then account for pH and its effect on the Gibbs energies of the different protonation states by applying the Alberty Legendre transform. This transform yields the appropriate thermodynamic potential of each microspecies at a specified pH and ionic strength [58]. These transformed Gibbs energies are then combined into a single transformed Gibbs energy for each reactant at a given pH, temperature, and ionic strength [58].

In order to randomly sample the potential energy surface of each microspecies, we sample subsets of size 5 out of the pool of geometric conformers for each microspecies, repeating the procedure described above for 30 iterations. The median error over all iterations is taken as the measure of accuracy of our procedure.

To test for improvements in accuracy obtained with RI-MP2, we perform single-point electronic energy estimates of each DFT-optimized structure (using a double-ζ auxiliary basis), and combine these with the DFT-based values for vibrational enthalpies and entropies.

We explore the increase in accuracy obtained in modeling two central reactions in glycolysis with first-principles thermochemistry estimates with and without single-point RI-MP2 electronic energy estimates of DFT-optimized structures. The first is the isomerization of dihydroxyacetone phosphate (DHAP) to glyceraldehyde-3-phosphate (G3P). The second is the carbon-bond cleavage reaction that transforms fructose-1,6-biphosphate (FBP) to DHAP and G3P. These reactions are shown in Figure 12.5a and b, respectively.

Table 12.4 shows the accuracies obtained with and without accelerated RI-MP2 single-point energy estimates. RI-MP2 single-point estimates improve accuracy in both test reaction systems. Although the error associated with the carbon-bond cleavage reaction is still significantly large, the

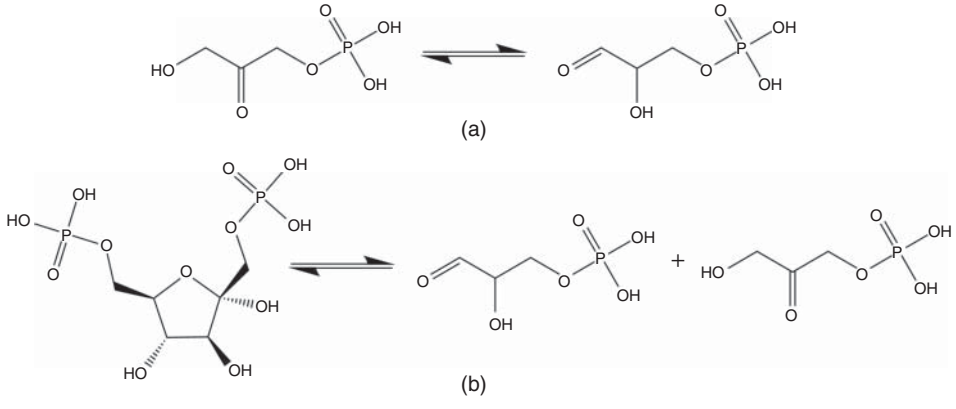


Figure 12.5 *Biochemical reactions studied. (a) Isomerization of dihydroxyacetone phosphate (DHAP) to glyceraldehyde-3-phosphate (G3P). (b) Fructose-1,6-biphosphate (FBP) to DHAP and G3P*

Table 12.4 *Comparison of Gibbs reaction energies (in kcal/mol) using RI-MP2 with respect to DFT*

	DFT		DFT + MP2 single-point	
Reaction	Median error	σ	Median error	σ
DHAP ↔ G3P	5.212	0.224	3.072	0.284
FBP ↔ DHAP + G3P	−56.068	0.725	−39.195	0.999

improvement of more than 10 kcal/mol in accuracy points to the usefulness of RI-MP2 single-point energy estimates. Recent work has shown that embedding metabolites solvated with explicit waters into implicit solvation models such as COSMO results in significantly improved accuracies [63]. Therefore, combining explicit–implicit mixed solvation schemes with RI-MP2 single-point energy estimates can potentially result in accuracies that are useful for metabolic engineering.

The molecules studied in this section are not larger than those presented in Table 12.3. Moreover, both the basis and auxiliary basis are double-ζ, so we should not expect large speedups (they are in the order of 2×). The energy improvements that we obtain can pave the way for a widespread study of the thermodynamics of metabolic reactions.

12.5 Conclusions

In this chapter, we have studied the different MP2 methods that were developed to enable the study of large systems such as biomolecules and nanomaterials. These include the RI approximation method, which by itself accelerates the MP2 method 10-fold at a negligible cost of accuracy. We find that the implementation of RI-MP2 on GPUs can accelerate the method up to an order of magnitude compared to a CPU implementation.

At the same time, several implementations to accelerate quantum chemistry methods replacing the DP calls with single precision have appeared in recent years. These have been largely motivated by the fact that the first GPUs did not support DP arithmetic. More recently, the main motivation is to obtain the same accuracy but taking advantage that SP implementations can be faster. For RI-MP2, we have shown that the use of mixed precision allows maintaining chemical accuracy in a controllable way and can provide additional acceleration. As current graphics cards support DP arithmetic, mixed-precision approaches are used to obtain acceleration in a variety of quantum chemistry algorithms.

Finally, we have studied the thermodynamics of two metabolic reactions using the GPU implementation of RI-MP2. The results are promising since RI-MP2 provides an improvement of accuracy with respect to measured thermodynamic data and this can now be obtained with reduced wall clock time.

References

1. Møller, C. and Plesset, M.S. (1934) Note on an approximation treatment for many-electron systems. *Phys. Rev.*, **46**, 618–622.
2. Frenking, G., Antes, I., Böhme, M., Dapprich, S., Ehlers, A.W., Jonas, V., Neuhaus, A., Otto, M., Stegmann, R., Veldkamp, A. and Vyboishchikov, S.F. (1996) Pseudopotential calculations of transition metal compounds: scope and limitations, in *Reviews in Computational Chemistry*, vol. **8** (eds K.B. Lipkowitz and D.B. Boyd), Wiley-VCH Verlag GmbH, New York, pp. 63–144.
3. Weigend, F., Köhn, A. and Hättig, C. (2002) Efficient use of the correlation consistent basis sets in resolution of the identity MP2 calculations. *J. Chem. Phys.*, **116**, 3175–3183.
4. Friesner, R.A. and Dunietz, B.D. (2001) Large-Scale Ab initio quantum chemical calculations on biological systems. *Acc. Chem. Res.*, **34**, 351–358.
5. He, X., Fusti-Molnar, L., Cui, G. and Merz, K.M. (2009) Importance of dispersion and electron correlation in Ab initio protein folding. *J. Phys. Chem. B*, **113**, 5290–5300.
6. Han, S., Deng, W.-Q. and Goddard, W. (2007) Improved designs of metal-organic frameworks for hydrogen storage. *Angew. Chem. Int. Ed.*, **119**, 6405–6408.
7. Jackson, N.E., Savoie, B.M., Kohlstedt, K.L., Olvera de la Cruz, M., Schatz, G.C., Chen, L.X. and Ratner, M.A. (2013) Controlling conformations of conjugated polymers and small molecules: the role of nonbonding interactions. *J. Am. Chem. Soc.*, **135**, 10475–10483.

8. Maschio, L. (2011) Local MP2 with density fitting for periodic systems: a parallel implementation. *J. Chem. Theory Comput.*, **7**, 2818–2830.
9. Boese, A.D. and Sauer, J. (2013) Accurate adsorption energies of small molecules on oxide surfaces: CO-MgO(001). *Phys. Chem. Chem. Phys.*, **15**, 16481–16493.
10. Feyereisen, M., Fitzgerald, G. and Komornicki, A. (1993) Use of approximate integrals in Ab initio theory. An application in MP2 energy calculations. *Chem. Phys. Lett.*, **208**, 359–363.
11. Weigend, F., Häser, M., Patzelt, H. and Ahlrichs, R. (1998) RI-MP2: optimized auxiliary basis sets and demonstration of efficiency. *Chem. Phys. Lett.*, **294**, 143–152.
12. Werner, H.J. and Manby, F.R. (2006) Explicitly correlated second-order perturbation theory using density fitting and local approximations. *J. Chem. Phys.*, **124**, 054114.
13. Maschio, L., Usvyat, D., Manby, F.R., Casassa, S., Pisani, C. and Schütz, M. (2007) Fast local-MP2 method with density-fitting for crystals. I. Theory and algorithms. *Phys. Rev. B*, **76**, 075101.
14. Schütz, M., Hetzer, G. and Werner, H.-J. (1999) Low-order scaling local electron correlation methods. I. Linear scaling local MP2. *J. Chem. Phys.*, **111**, 5691.
15. Koch, H., Sánchez de Merás, A. and Pedersen, T.B. (2003) Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, **118**, 9481.
16. Aquilante, F., Lindh, R. and Pedersen, T.B. (2007) Unbiased auxiliary basis sets for accurate two-electron integral approximations. *J. Chem. Phys.*, **127**, 114107.
17. Kobayashi, M., Imamura, Y. and Nakai, H. (2007) Alternative linear-scaling methodology for the second-order Møller-Plesset perturbation calculation based on the divide-and-conquer method. *J. Chem. Phys.*, **127**, 074103.
18. Katouda, M., Kobayashi, M., Nakai, H. and Nagase, S. (2011) Two-level hierarchical parallelization of second-order Møller-Plesset perturbation calculations in divide-and-conquer method. *J. Comput. Chem.*, **32**, 2756–2764.
19. Hättig, C. (2006) Beyond Hartree-Fock: MP2 and coupled-cluster methods for large systems, in *Computational Nanoscience: Do It Yourself!*, vol. **31** (eds J. Grotendorst, S. Blugel and D. Marx), John von Neumann Institute for Computing, Jülich, pp. 245–278.
20. Born, M. and Oppenheimer, R. (1927) Zur Quantentheorie der Molekeln. *Ann. Phys.*, **389**, 457–484.
21. Helgaker, T., Jørgensen, P. and Olsen, J. (2000) *Molecular Electronic-Structure Theory*, John Wiley & Sons, Ltd, Chichester.
22. Ochsenfeld, C., Kussmann, J. and Lambrecht, D. (2007) Linear-scaling methods in quantum chemistry, in *Reviews in Computational Chemistry*, vol. **23** (eds K.B. Lipkowitz and T.R. Cundari), John Wiley & Sons, Inc., Hoboken, NJ, pp. 1–82.
23. Werner, H.-J., Manby, F.R. and Knowles, P.J. (2003) Fast linear scaling second-order Møller-Plesset perturbation theory (MP2) using local and density fitting approximations. *J. Chem. Phys.*, **118**, 8149.
24. NVIDIA CUBLAS Library 1.0, `https://developer.nvidia.com/cuBLAS` (accessed 25 September 2015).
25. Pulay, P. (1983) Localizability of dynamic electron correlation. *Chem. Phys. Lett.*, **100**, 151–154.
26. Pulay, P. and Saebø, S. (1986) Orbital-invariant formulation and second-order gradient evaluation in Møller-Plesset perturbation theory. *Theor. Chim. Acta*, **69**, 357–368.
27. Kobayashi, M., Akama, T. and Nakai, H. (2006) Second-order Møller-Plesset perturbation energy obtained from divide-and-conquer Hartree-Fock density matrix. *J. Chem. Phys.*, **125**, 204106.
28. Katouda, M. and Nakajima, T. (2013) MPI/OpenMP hybrid parallel algorithm of resolution of identity second-order Møller-Plesset perturbation calculation for massively parallel multicore supercomputers. *J. Chem. Theory Comput.*, **9**, 5373–5380.

29. Ma, W., Krishnamoorthy, S., Villa, O. and Kowalski, K. (2011) GPU-based implementations of the noniterative regularized-CCSD(T) corrections: applications to strongly correlated systems. *J. Chem. Theory Comput.*, **7**, 1316–1327.
30. DePrince, A.E. and Hammond, J.R. (2011) Coupled cluster theory on graphics processing units I. The coupled cluster doubles method. *J. Chem. Theory Comput.*, **7**, 1287–1295.
31. Bhaskaran-Nair, K., Ma, W., Krishnamoorthy, S., Villa, O., van Dam, H.J.J., Aprá, E. and Kowalski, K. (2013) Noniterative multireference coupled cluster methods on heterogeneous CPU-GPU systems. *J. Chem. Theory Comput.*, **9**, 1949–1957.
32. Andrade, X. and Aspuru-Guzik, A. (2013) Real-space density functional theory on graphical processing units: computational approach and comparison to gaussian basis set methods. *J. Chem. Theory Comput.*, **9**, 4360–4373.
33. Kreisbeck, C., Kramer, T., Rodríguez, M. and Hein, B. (2011) High-performance solution of hierarchical equations of motion for studying energy transfer in light-harvesting complexes. *J. Chem. Theory Comput.*, **7**, 2166–2174.
34. Hanrath, M. and Engels-Putzka, A. (2010) An efficient matrix-matrix multiplication based antisymmetric tensor contraction engine for general order coupled cluster. *J. Chem. Phys.*, **133**, 064108.
35. Brown, P., Woods, C., McIntosh-Smith, S. and Manby, F.R. (2008) Massively multicore parallelization of Kohn-Sham theory. *J. Chem. Theory Comput.*, **4**, 1620–1626.
36. Göddeke, D., Strzodka, R. and Turek, S. (2007) Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *Int. J. Parallel Emergent Distrib. Syst.*, **22**, 221–256.
37. Susukita, R., Ebisuzaki, T., Elmegreen, B.G., Furusawa, H., Kato, K., Kawai, A., Kobayashi, Y., Koishi, T., McNiven, G.D., Narumi, T. and Yasuoka, K. (2003) Hardware accelerator for molecular dynamics: MDGRAPE-2. *Comput. Phys. Commun.*, **155**, 115–131.
38. Yasuda, K. (2008) Two-electron integral evaluation on the graphics processor unit. *J. Comput. Chem.*, **29**, 334–342.
39. Luehr, N., Ufimtsev, I.S. and Martínez, T.J. (2011) Dynamic precision for electron repulsion integral evaluation on graphical processing units (GPUs). *J. Chem. Theory Comput.*, **7**, 949–954.
40. Titov, A.V., Ufimtsev, I.S., Luehr, N. and Martinez, T.J. (2013) Generating efficient quantum chemistry codes for novel architectures. *J. Chem. Theory Comput.*, **9**, 213–221.
41. Olivares-Amaya, R., Watson, M.A., Edgar, R.G., Vogt, L., Shao, Y. and Aspuru-Guzik, A. (2010) Accelerating correlated quantum chemistry calculations using graphical processing units and a mixed precision matrix multiplication library. *J. Chem. Theory Comput.*, **6**, 135–144.
42. Watson, M., Olivares-Amaya, R., Edgar, R.G. and Aspuru-Guzik, A. (2010) Accelerating correlated quantum chemistry calculations using graphical processing units. *Comput. Sci. Eng.*, **12**, 40–51.
43. Vysotskiy, V.P. and Cederbaum, L.S. (2011) Accurate quantum chemistry in single precision arithmetic: correlation energy. *J. Chem. Theory Comput.*, **7**, 320–326.
44. Knizia, G., Li, W., Simon, S. and Werner, H.-J. (2011) Determining the numerical stability of quantum chemistry algorithms. *J. Chem. Theory Comput.*, **7**, 2387–2398.
45. Vogt, L., Olivares-Amaya, R., Kermes, S., Shao, Y., Amador-Bedolla, C. and Aspuru-Guzik, A. (2008) Accelerating resolution-of-the-identity second-order Møller-Plesset quantum chemistry calculations with graphical processing units. *J. Phys. Chem. A*, **112**, 2049–2057.
46. Bohannon, J. (2005) Grassroots supercomputing. *Science*, **308**, 310.
47. Hachmann, J., Olivares-Amaya, R., Atahan-Evrenk, S., Amador-Bedolla, C., Sánchez-Carrera, R.S., Gold-Parker, A., Vogt, L., Brockway, A.M. and Aspuru-Guzik, A. (2011) The Harvard clean energy project: large-scale computational screening and design of organic photovoltaics on the world community grid. *J. Phys. Chem. Lett.*, **2**, 2241–2251.

48. Olivares-Amaya, R., Amador-Bedolla, C., Hachmann, J., Atahan-Evrenk, S., Sánchez-Carrera, R.S., Vogt, L. and Aspuru-Guzik, A. (2011) Accelerated computational discovery of high-performance materials for organic photovoltaics by means of cheminformatics. *Energy Environ. Sci.*, **4**, 4849–4861.
49. Hachmann, J., Olivares-Amaya, R., Jinich, A., Appleton, A.L., Blood-Forsythe, M.A., Seress, L.R., Roman-Salgado, C., Trepte, K., Atahan-Evrenk, S., Er, S., Shrestha, S., Mondal, R., Sokolov, A., Bao, Z. and Aspuru-Guzik, A. (2014) Lead candidates for high-performance organic photovoltaics from high-throughput quantum chemistry—the Harvard Clean Energy Project. *Energy Environ. Sci.*, **7**, 698–704.
50. Dunning, T. Jr. (1989) Gaussian basis sets for use in correlated molecular calculations. I. The atoms boron through neon and hydrogen. *J. Chem. Phys.*, **90**, 1007–1023.
51. Yasuda, K. (2008) Accelerating density functional calculations with graphics processing unit. *J. Chem. Theory Comput.*, **4**, 1230.
52. Ufimtsev, I.S. and Martinez, T.J. (2009) Quantum chemistry on graphical processing units. 2. Direct self-consistent-field implementation. *J. Chem. Theory Comput.*, **5**, 1004–1015.
53. Breslow, R. and Dong, S.D. (1998) Biomimetic reactions catalyzed by cyclodextrins and their derivatives. *Chem. Rev.*, **98**, 1997–2012.
54. Harrison, C.B. and Schulten, K. (2012) Quantum and classical dynamics simulations of ATP hydrolysis in solution. *J. Chem. Theory Comput.*, **8**, 2328–2335.
55. Kobayashi, E., Yura, K. and Nagai, Y. (2013) Distinct conformation of ATP Molecule in solution and on protein. *Biophysics*, **9**, 1–12.
56. Guillot, B., Muzet, N., Artacho, E., Lecomte, C. and Jelsch, C. (2003) Experimental and theoretical electron density studies in large molecules: NAD^+, β-nicotinamide adenine dinucleotide. *J. Phys. Chem. B*, **107**, 9109–9121.
57. Wu, Y.-D. and Houk, K.N. (1993) Theoretical study of conformational features of NAD^+ and NADH analogs: protonated nicotinamide and 1,4-dihydronicotinamide. *J. Org. Chem.*, **58**, 2043–2045.
58. Alberty, R.A. (2003) *Thermodynamics of Biochemical Reactions*, Wiley-Interscience.
59. Henry, C.S., Broadbelt, L.J. and Hatzimanikatis, V. (2007) Thermodynamics-based metabolic flux analysis. *Biophys. J.*, **92**, 1792–1805.
60. Bar-Even, A., Noor, E., Lewis, N.E. and Milo, R. (2010) Design and analysis of synthetic carbon fixation pathways. *Proc. Natl. Acad. Sci. U.S.A.*, **107**, 8889–8894.
61. Bar-Even, A., Flamholz, A., Noor, E. and Milo, R. (2012) Thermodynamic constraints shape the structure of carbon fixation pathways. *Biochim. Biophys. Acta*, **1817**, 1646–1659.
62. Goldberg, R.N., Tewari, Y.B. and Bhat, T.N. (2004) Thermodynamics of enzyme-catalyzed reactions—a database for quantitative biochemistry. *Bioinformatics*, **20**, 2874–2877.
63. Jinich, A., Rappoport, D., Dunn, I., Sanchez-Lengelin, B., Olivares-Amaya, R., Noor, E., Bar-Even, A., Milo, R. and Aspuru-Guzik, A. (2014) Quantum chemical approach to estimating the thermodynamics of metabolic reactions. *Sci. Rep.*, **4**, Article number: 7022, doi: 10.1038/srep07022.
64. Martínez, J.M. and Martínez, L. (2003) Packing optimization for automated generation of complex system's initial configurations for molecular dynamics and docking. *J. Comput. Chem.*, **24**, 819–825.
65. Martínez, L., Andrade, R., Birgin, E.G. and Martínez, J.M. (2009) PACKMOL: a package for building initial configurations for molecular dynamics simulations. *J. Comput. Chem.*, **30**, 2157–2164.
66. Neese, F. (2012) The ORCA program system. *WIREs Comput. Mol. Sci.*, **2**, 73–78.
67. Vosko, S., Wilk, L. and Nusair, M. (1980) Accurate spin-dependent electron liquid correlation energies for local spin density calculations: a critical analysis. *Can. J. Phys.*, **58**, 1200–1211.

68. Becke, A.D. (1988) Density-functional exchange-energy approximation with correct asymptotic behavior. *Phys. Rev. A*, **38**, 3098–3100.
69. Lee, C., Yang, W. and Parr, R.G. (1988) Development of the colle and salvetti correlation-energy formula into a functional of the electron. *Phys. Rev. B*, **37**, 785–789.
70. Becke, A.D. (1993) Density-functional thermochemistry. III. The role of exact exchange. *J. Chem. Phys.*, **98**, 5648–5652.
71. Hehre, W.J., Ditchfield, R. and Pople, J. (1972) Self-consistent molecular orbital methods. XII. Further extensions of gaussian-type basis sets for use in molecular orbital studies of organic molecules. *J. Chem. Phys.*, **56**, 2257.
72. McLean, A.D. and Chandler, G.S. (1980) Contracted Gaussian basis sets for molecular calculations. I. Second row atoms, Z=11-18. *J. Chem. Phys.*, **72**, 5639.
73. Francl, M.M., Pietro, W.J., Hehre, W.J., Binkley, J.S., Gordon, M.S., DeFrees, D.J. and Pople, J.A. (1982) Self-consistent molecular orbital methods. XXIII. A polarization-type basis set for second-row elements. *J. Chem. Phys.*, **77**, 3654.
74. McQuarrie, D.A. and Simon, J.D. (1999) *Molecular Thermodynamics*, University Science Books.

13

Iterative Coupled-Cluster Methods on Graphics Processing Units

A. Eugene DePrince III[1], *Jeff R. Hammond*[2] *and C. David Sherrill*[3,4,5]

[1]*Department of Chemistry and Biochemistry, Florida State University, Tallahassee, FL, USA*
[2]*Leadership Computing Facility, Argonne National Laboratory, Argonne, IL, USA*
[3]*Center for Computational Molecular Science and Technology, Georgia Institute of Technology, GA, USA*
[4]*School of Chemistry and Biochemistry, Georgia Institute of Technology, Atlanta, GA, USA*
[5]*School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA, USA*

In this chapter, we present algorithms for iterative coupled-cluster computations in heterogeneous computing environments. Careful algorithm design with respect to simultaneous use of both CPU and GPU processors enables accelerations close to the relative performance of general matrix multiplications (DGEMM) on the different architectures. To achieve best performance, we distribute work between the CPU and GPU in terms of entire diagrams, evaluate the most expensive tensor contractions on the GPU, and interleave computation and communication for work performed on the GPU. Using these techniques, a robust implementation that utilizes either density fitting or Cholesky decomposition approximations to minimize storage requirements and data transfer can achieve a threefold speedup using a Kepler GPU and multi-core Intel Xeon CPU as compared to an optimized, parallel CPU based implementation running on the same hardware.

13.1 Introduction

As highlighted in Chapter 1 the landscape of computer hardware is constantly changing, but not all changes are equally disruptive to scientific software programmers. After approximately three decades of continuous performance improvement from steadily increasing clock frequencies and concomitant

Electronic Structure Calculations on Graphics Processing Units:
From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

decrease in semiconductor feature size, microprocessor design underwent a fundamental shift toward parallelism at the beginning of the last decade. Having reached the limits of frequency-scaling due to power constraints, node-level parallelism is now the basis for keeping pace with Moore's law. Multicore parallelism has now reached every area of the computing market: computing devices ranging from smartphones to the largest supercomputers use multicore processors. Initially, this had little impact on software engineers or users – two or four cores on a single chip was not significantly different than two or four single-core chips on a single node or even multiple nodes within the ubiquitous message-passing parallel programming model. However, in order to continue to satisfy Moore's law and thus double the performance of processors approximately every 2 years, core counts have increased steadily, with 8-core processors common for workstations in 2013 and 16-core processors recently deployed.

As discussed in Chapter 2 the rise of GPUs was due to the relative mismatch between the high-throughput workloads required for rendering and the latency-optimized nature of superscalar CPUs [1, 2]. By focusing on throughput rather than latency, GPUs are able to deliver substantially more performance than their CPU peers for specialized workloads. Over time, however, the increasing thread parallelism of SIMD vector units in CPUs and the increases in the general functionality and programmability of GPUs has reduced their differences significantly. Nevertheless, as discussed in earlier chapters there still exists a large gap in the memory bandwidth between CPUs and GPUs and carefully exploiting this can yield large performance improvements in the use of GPUs rather than CPUs.

As mentioned earlier in this book, node-level parallelism comes in two categories: multicore CPUs and manycore GPUs. In a multicore CPU, each core is capable of acting as an independent general-purpose processor, whereas GPU cores are not general purpose (e.g., they cannot run an operating system) and do not have independent instruction units. Each of these architectures pose different challenges to scientific codes.

In this chapter, we review strategies for implementing iterative coupled-cluster methods (doubles coupled-cluster [CCD] and singles and doubles coupled cluster [CCSD]) in heterogeneous computing environments to overcome the challenges of efficiently using GPUs for such calculations. We highlight preliminary investigations [3, 4] that sought to establish a best-case for performance acceleration when porting these methods to GPUs. The algorithms developed at the time were not production-level but nevertheless established a realistic upper bound to the speedups that can be expected from modern manycore architectures. We then discuss more recent work that has emphasized the role of approximate tensor decompositions in the design of an efficient, production-level implementation of CCSD. We then follow with a discussion of an implementation of GPU-accelerated CCSD that makes use of density fitting (DF) or Cholesky decomposition (CD) approximations in the construction and contraction of all two-electron repulsion integrals (ERIs) [5]. These approximations reduce the storage requirements for the ERI tensor and hence mitigate the cost of data transfers to the device. The present algorithm makes simultaneous use of both GPU and CPU resources, resulting in performance accelerations of nearly a factor of 3 when using a single Nvidia K20 (Kepler) GPU or two Nvidia C2070 (Fermi) GPUs, relative to the same algorithm executed on 6 Intel Core i7-3930K CPU cores. Finally, the performance of the GPU-accelerated DF/CD-CCSD implementation for systems with as many as 822 active basis functions is demonstrated.

13.2 Related Work

Only very recently have quantum many-body methods based upon the coupled-cluster wave function expansion been implemented on GPUs. Our own work (Refs [3–5]) focused on the development of CCSD [6] for GPUs, while Kowalski and coworkers have focused on the computationally intensive

perturbative triples correction to CCSD [CCSD(T)] [7] for both ground- and excited-states [8, 9] (see also next chapter). Another recent example of a GPU-enabled CCSD(T) algorithm can be found in Ref. [10]. Some prior attempts to implement coupled-cluster theory on GPUs have been less successful [11], indicating that this task is not as trivial as some believed and that its heavy reliance upon BLAS calls does not imply that simply using an accelerated BLAS library will yield large performance improvements.

13.3 Theory

13.3.1 CCD and CCSD

The electronic wave function in coupled-cluster theory is

$$|\Psi_{\mathrm{CC}}\rangle = e^{\hat{T}}|\Psi_0\rangle, \tag{13.1}$$

where $|\Psi_0\rangle$ represents some reference state (here, a single restricted Hartree–Fock reference determinant). The symbol, $\hat{T}$, represents an excitation operator which, for the CCSD method, contains all single and double excitations ($\hat{T} = \hat{T}_1 + \hat{T}_2$). CCSD is typically solved via projection and Jacobi iteration (with DIIS convergence acceleration in most cases [12, 13]), where the correlation energy, E_c, is given by

$$E_c = \langle\Psi_0|e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Psi_0\rangle, \tag{13.2}$$

and the single and double excitation amplitudes are given by

$$0 = \langle\Psi_i^a|e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Psi_0\rangle, \tag{13.3}$$

$$0 = \langle\Psi_{ij}^{ab}|e^{-\hat{T}}\hat{H}e^{\hat{T}}|\Psi_0\rangle. \tag{13.4}$$

We define the spin-free coupled cluster single and double (CCSD) excitation amplitudes according to the (slightly modified) expressions provided by Piecuch *et al.* [14]:

$$\begin{aligned}(f_i^i + f_j^j - f_a^a - f_b^b)t_{ij}^{ab} = v_{ij}^{ab} + P(ia,jb)[&t_{ij}^{ae}I_e^b - t_{im}^{ab}I_j^m \\ &+\frac{1}{2}v_{ef}^{ab}c_{ij}^{ef} + \frac{1}{2}c_{mn}^{ab}I_{ij}^{mn} - t_{mj}^{ae}I_{ie}^{mb} - I_{ie}^{ma}t_{mj}^{eb} \\ &+(2t_{mi}^{ea} - t_{im}^{ea})I_{ej}^{mb} + t_i^e I_{ej}'^{ab} - t_m^a I_{ij}'^{mb}].\end{aligned} \tag{13.5}$$

We also have the single excitation amplitudes, defined by

$$\begin{aligned}(f_i^i - f_a^a)t_i^a = I_e^a t_i^e - I_i'^m t_m^a \\ +I_e^m(2t_{mi}^{ea} - t_{im}^{ea}) + (2v_{ei}^{ma} - v_{ei}^{am})t_m^e \\ -v_{ei}^{mn}(2t_{mn}^{ea} - t_{mn}^{ae}) + v_{ef}^{ma}(2t_{mi}^{ef} - t_{im}^{ef}).\end{aligned} \tag{13.6}$$

Here, i, j, k, l, m, and n (a, b, c, d, e, and f) represent orbitals that are occupied (virtual) in the reference function. The tensors defined in Eqs. (13.5) and (13.6) are given by

$$I_b^a = (2v_{be}^{am} - v_{be}^{ma})t_m^e - (2v_{eb}^{mn} - v_{be}^{mn})c_{mn}^{ea}, \tag{13.7}$$

$$I_a^i = (2v_{ae}^{im} - v_{ea}^{im})t_m^e, \tag{13.8}$$

$$I_j'^i = (2v_{je}^{im} - v_{ej}^{im})t_m^e + (2v_{ef}^{mi} - v_{ef}^{im})t_{mj}^{ef}, \tag{13.9}$$

$$I_j^i = I_j'^i + I_e^i t_j^e, \tag{13.10}$$

$$I_{kl}^{ij} = v_{kl}^{ij} + v_{ef}^{ij} c_{kl}^{ef} + P(ik,jl) t_k^e v_{el}^{ij}, \tag{13.11}$$

$$I_{ci}'^{ab} = v_{ci}^{ab} - v_{ci}^{am} t_m^b - v_{ci}^{mb} t_m^a, \tag{13.12}$$

$$I_{jk}'^{ia} = v_{jk}^{ia} + v_{ef}^{ia} c_{jk}^{ef}, \tag{13.13}$$

$$\begin{aligned} I_{jb}^{ia} = {} & v_{jb}^{ia} - \frac{1}{2} v_{eb}^{im} (t_{jm}^{ea} + 2 t_j^e t_m^a) \\ & + v_{eb}^{ia} t_j^e - v_{jb}^{im} t_m^a, \end{aligned} \tag{13.14}$$

$$\begin{aligned} I_{bj}^{ia} = {} & v_{bj}^{ia} - \frac{1}{2} v_{be}^{im} (t_{mj}^{ae} + 2 t_m^a t_j^e) \\ & + v_{be}^{ia} t_j^e - v_{bj}^{im} t_m^a + \frac{1}{2} (2 v_{be}^{im} - v_{eb}^{im}) t_{mj}^{ea}. \end{aligned} \tag{13.15}$$

The symbol c_{ij}^{ab} represents a sum of singles and doubles amplitudes: $c_{ij}^{ab} = t_{ij}^{ab} + t_i^a t_j^b$. We have assumed that the molecular orbitals are canonical Hartree–Fock orbitals, and the diagonal elements of the Fock matrix, f_p^p, are orbital energies. The two-electron integrals are denoted by $v_{ij}^{ab} = (ia|jb)$. The permutation operator $P(ia,jb)$ implies the sum of two terms: $P(ia,jb) v_{ij}^{ab} = v_{ij}^{ab} + v_{ji}^{ba}$. The Einstein summation convention is assumed whereby repeated upper and lower indices are summed over an appropriate range, but note that the left-hand sides of Eqs. (13.5) and (13.6) involve no sum.

By ignoring all singly excited CC amplitudes, Eqs. (13.5)–(13.15) reduce to the CCD method. CCD is seldom used in chemical applications because single excitation amplitudes substantially improve the accuracy of the method, and CCSD is not significantly more expensive than CCD [15]. However, from an implementation standpoint, the CCD equations are conveniently compact, facilitating the design of an optimal algorithm. For this reason, we study GPU acceleration of the CCD method before proceeding to the CCSD method. Further, in CCSD, the single excitation amplitudes can be incorporated directly into the Hamiltonian, as is described in Ref. [16], and thus the CCSD equations are, in principle, no more complex than the CCD equations. Our more recent work described in Sections 13.3.2 and 13.6.3 uses these techniques.

13.3.2 Density-Fitted CCSD with a t_1-Transformed Hamiltonian

As was discussed in Chapter 2 and as a recurring theme throughout the previous chapters the two major challenges in adapting scientific algorithms for use on graphics processors are: (i) the high cost of data transfers between the host and the device and (ii) the limited global memory available on the device, relative to the host. Our initial explorations considered strategies to address these concerns by overlapping communication and computation and performing computations simultaneously on the CPU and GPU. In this section, we discuss a complementary approach that directly reduces the amount of data that needs to be transferred to and stored on the device.

The storage requirements for the ERI tensor can be reduced via approximate factorizations known as DF (the resolution of the identity) [17–24], CD [25–28], or tensor hypercontraction DF [29, 30]). Rendell *et al.* first proposed DF within coupled cluster theory as a means of reducing I/O bottlenecks associated with the $(vv|vv)$ and $(ov|vv)$ blocks of the ERI tensor in a conventional CCSD algorithm [21]. Here, o and v represent orbitals that are occupied or virtual in the reference function, respectively. In addition, the approximate CCSD method known as CC2 [31, 32] is most efficiently implemented using DF and the t_1-transformation of the Hamiltonian described in Ref. [16]. Recently, several examples of CCSD algorithms that make use of DF or CD approximations for all blocks of the ERI tensor have emerged [33–36]. DF and CD approximations do not reduce the formal scaling of CCSD. However, the associated reduction in I/O is potentially useful in computing environments

that make use of networked storage where high latency noticeably affects performance. Further, GPU CCSD implementations built upon any of these approximations immediately benefit from the reduced storage and host/device-transfer requirements for the ERI tensors.

Compact CCSD equations can be obtained by dressing the ERI tensor and Fock matrix with t_1 amplitudes as described in Ref. [16]. The doubles residual, R_{ij}^{ab} is then expressed (in a spin-free formalism) as

$$R_{ij}^{ab} = \sum_Q B_{ai}^Q B_{bj}^Q + A_{ij}^{ab} + B_{ij}^{ab} + P_{ij}^{ab}\left(\frac{1}{2}C_{ij}^{ab} + C_{ji}^{ab} + D_{ij}^{ab} + E_{ij}^{ab} + G_{ij}^{ab}\right), \tag{13.16}$$

$$A_{ij}^{ab} = \sum_{cd} t_{ij}^{cd}\left(\sum_Q B_{ac}^Q B_{bd}^Q\right), \tag{13.17}$$

$$B_{ij}^{ab} = \sum_{kl} t_{kl}^{ab}\left[\left(\sum_Q B_{ki}^Q B_{lj}^Q\right) + \left(\sum_{cd} t_{ij}^{cd}\left[\sum_Q B_{kc}^Q B_{ld}^Q\right]\right)\right], \tag{13.18}$$

$$C_{ij}^{ab} = -\sum_{kc} t_{kj}^{bc}\left[\left(\sum_Q B_{ki}^Q B_{ac}^Q\right) - \frac{1}{2}\left(\sum_{ld} t_{li}^{ad}\left[\sum_Q B_{kd}^Q B_{lc}^Q\right]\right)\right], \tag{13.19}$$

$$D_{ij}^{ab} = \frac{1}{2}\sum_{kc} u_{jk}^{bc}\left(L_{aikc} + \frac{1}{2}\left[\sum_{ld} u_{il}^{ad} L_{ldkc}\right]\right), \tag{13.20}$$

$$E_{ij}^{ab} = \sum_c t_{ij}^{ac}\left[f_{bc} - \left(\sum_{kld} u_{kl}^{bd}\left[\sum_Q B_{ld}^Q B_{kc}^Q\right]\right)\right], \tag{13.21}$$

$$G_{ij}^{ab} = -\sum_k t_{ik}^{ab}\left[f_{kj} + \left(\sum_{lcd} u_{lj}^{cd}\left[\sum_Q B_{kd}^Q B_{lc}^Q\right]\right)\right], \tag{13.22}$$

$$L_{pqrs} = 2\sum_Q B_{pq}^Q B_{rs}^Q - \sum_Q B_{ps}^Q B_{rq}^Q, \tag{13.23}$$

$$u_{rs}^{pq} = 2t_{rs}^{pq} - t_{rs}^{qp}. \tag{13.24}$$

The quantity B_{ij}^Q is a t_1-transformed three-index integral, the operator, P_{ij}^{ab}, is the same permutation operator described above, and f_{pq} denotes a (t_1-transformed) Fock matrix element. Brackets and parentheses denote the order of operations for the efficient construction of intermediates. The singles residual, R_i^a, is

$$R_i^a = f_{ai} + A_i^a + B_i^a + C_i^a, \tag{13.25}$$

$$A_i^a = \sum_{dQ}\left(\sum_{kc} u_{ki}^{cd} B_{kc}^Q\right) B_{ad}^Q, \tag{13.26}$$

$$B_i^a = -\sum_{klc} u_{kl}^{ac}\left(\sum_Q B_{ki}^Q B_{lc}^Q\right), \tag{13.27}$$

$$C_i^a = \sum_{kc} f_{kc} u_{ik}^{ac}. \tag{13.28}$$

With the exception of the use of three-index integrals, these equations are very similar to those presented in Ref. [16].

The effects of single excitations are incorporated into the Hamiltonian by dressing the three-index integrals as

$$B_{rs}^{Q} = \sum_{\mu} X_{\mu r} \left(\sum_{\nu} Y_{\nu s} B_{\mu\nu}^{Q} \right). \tag{13.29}$$

Here, the indices μ and ν represent atomic orbital (AO) basis functions, and $B_{\mu\nu}^{Q}$ represents a three-index integral in the AO basis. The matrices $\mathbf{X}$ and $\mathbf{Y}$ are modified AO/MO transformation matrices that absorb the singles amplitudes:

$$\mathbf{t}_1 = \begin{pmatrix} 0 & 0 \\ t_i^a & 0 \end{pmatrix}, \tag{13.30}$$

$$\mathbf{X} = \mathbf{C}(1 - \mathbf{t}_1^T), \tag{13.31}$$

$$\mathbf{Y} = \mathbf{C}(1 + \mathbf{t}_1). \tag{13.32}$$

In the $\mathbf{t}_1$ matrix, only the lower-left block is nonzero, with ov elements. The Fock matrix can now be expressed in terms of these modified three-index integrals and the one-electron integrals, h_{rs}:

$$f_{rs} = h_{rs} + \sum_{i} \left(2 \sum_{Q} B_{rs}^{Q} B_{ii}^{Q} - \sum_{Q} B_{ri}^{Q} B_{is}^{Q} \right), \tag{13.33}$$

$$h_{rs} = \sum_{\mu} X_{\mu r} \left(\sum_{\nu} Y_{\nu s} h_{\mu\nu} \right). \tag{13.34}$$

When using conventional four-index integrals, folding t_1 into the Hamiltonian requires $\mathcal{O}(N^5)$ floating-point operations per iteration, where N is the dimension of the one-electron basis. By substituting DF or CD integrals for four-index integrals, the cost of this transformation is reduced to $\mathcal{O}(N^4)$.

13.4 Algorithm Details

13.4.1 Communication-Avoiding CCD Algorithm

The spin-free CCD equations, obtained by neglecting all terms containing single excitations in Eqs. (13.5)–(13.15), consist of 13 tensor contractions, 9 that scale as the sixth power of system size and 4 that scale as the fifth power of system size. It is well-known that these tensor contractions can be evaluated on a CPU using tuned BLAS libraries, which are capable of achieving up to 90% of the theoretical peak performance of a modern multicore CPU. We have demonstrated that a similar strategy, using the Nvidia CUBLAS library, works quite well for implementing the CCD equations on a GPU [3]. For systems with less than 250 active orbitals, an Nvidia C2050 GPU can accelerate a single iteration of the CCD method by a factor of 4–5 (relative to two 8-core Intel Xeon X5550 CPUs). This acceleration is consistent with the relative performance of DGEMM on the CPU and GPU.

The implementation in Ref. [3] represents the first example of the iterative portion of any CC method executed entirely on GPUs. By storing all tensors directly on the device, we eliminate host/device communication during the CC iterations, ensuring a best-case for performance acceleration. However, this algorithm is severely constrained by the modest total memory on the GPU. For small systems, no communication between host and device occurs once the ERI tensor (generated on the host by the GAMESS electronic structure package [37]) and the Fock matrix are copied to the device,

and convergence is monitored directly on the device (using cudaDnrm2[1]). For larger systems, where GPU memory cannot accomodate the v_{cd}^{ab} block of the ERI tensor, the associated contraction can be tiled and manageable blocks pushed to the device sequentially. This strategy precludes the aforementioned communication-free algorithm, but, for the systems studied, the overhead is nominal, suggesting that a low-storage algorithm in which all data is repeatedly copied to the device may be viable.

13.4.2 Low-Storage CCSD Algorithm

The spin-free CCSD method, as defined by Eqs. (13.5)–(13.15), requires 34 tensor contractions whose computational costs scale from the third to the sixth power of system size. We now abandon the notion of zero communication and adopt a strategy which repeatedly copies integrals and amplitudes to the device. These repeated transfers will incur some overhead that, fortunately, can be masked by simultaneously performing operations on both the GPU and CPU. The tensor contractions are distributed between CPU and GPU processors according to the computational cost to evaluate them. If a contraction scales as N^4 or less, it is initially classified as CPU work (N, here, is a measure of system size), and contractions scaling as N^6 are classified as GPU work. Terms that scale as N^5 are usually classified as CPU work, but, occasionally, the work is better suited for the GPU. For example, the required tensors may already be present in global memory on the device, or CPU memory operations (in the form of tensor permutations) may be significantly reduced by evaluating the diagram on the device. Whenever possible, tensor permutations should be performed on the device, as it possesses higher memory bandwidth than the host. We pipeline all data transfers, permutations, and contractions into a single stream, with host/device transfers occuring via cudaMemcpyAsync.[2]

Most contractions involve the CC amplitudes (or a similarly sized intermediate tensor), the residual of the CC equations (the right-hand sides of Eqs. (13.5) and (13.6)), and a block of the ERI tensor. For the doubles equations, we require that three arrays be stored on the GPU: (i) the doubles amplitudes, t_{ij}^{ab}, (ii) the residual of the doubles equations, and (iii) a general integral buffer. For the singles equations, we require that the singles amplitudes and residual be stored. In addition, in order to limit excessive host/device communications, one additional buffer of the size of the doubles amplitudes is allocated. The general integral buffer is allocated to be as large as possible, given the storage requirements for all other tensors. For this algorithm, this buffer must be at least large enough to accomodate data the size of the doubles amplitudes, o^2v^2. Thus, the minimum storage requirement for the GPU in this particular implementation is $\approx 4o^2v^2$ double precision numbers.

The minimum storage required for the general integral buffer on the GPU is o^2v^2. However, two blocks of the ERI tensor, v_{cd}^{ab} and v_{ci}^{ab} have v^4 and ov^3 elements, and because v is usually much larger than o, the general integral buffer may not accomodate them. Accordingly, contractions involving these tensors are tiled and performed sequentially. The low-storage CCSD algorithm described here supports tiling of all contractions involving tensors larger than o^2v^2. Four such contractions arise in this implementation of CCSD.

It should be noted here that a considerable reduction in the storage and floating point operations required for the evaluation of the largest tensor contraction, $\frac{1}{2}\sum_{ef} v_{ef}^{ab} c_{ij}^{ef}$, can be obtained by re-expressing the sum in terms of symmetric and antisymmetric components [38]. In total, a factor of 2 savings in storage and floating point operations can be gained *for this term* at the expense of a few additional tensor permutations. This low-storage CCSD algorithm and our DF-CCSD algorithm both utilize this factorization, while the algorithm of Ref. [3] does not. For the set of molecules studied in Ref. [3], the factorization decreases iteration times by roughly 10%; the effects are more significant as the size of the virtual space increases relative to the size of the occupied space.

[1] Technically cudaDnrm2 returns a value to CPU memory, but this communication is negligible.

[2] This strategy is obviously less effective with hardware that lack asynchronous memory capabilities.

Both the low-communication CCD and low-storage CCSD algorithms store all required blocks of the ERI tensor in main CPU memory and are thus not extensible to large systems. They nonetheless serve as useful explorations of possible strategies for porting coupled-cluster codes to heterogeneous computing environments. By eliminating I/O costs, host/device communication can be carefully analyzed to design algorithms that minimize their effects. These implementations also provide a best-case scenario for performance against which other algorithms can be measured.

13.4.3 Density-Fitted CCSD with a t_1-Transformed Hamiltonian

To move beyond the limited, proof-of-concept implementations described in Sections 13.4.1 and 13.4.2, we must consider algorithms that (i) do not store the full ERI tensor in main CPU memory and (ii) make few (or no) assumptions regarding the amount of data that can fit in GPU global memory. This latter stipulation requires blocking of essentially all tensor contractions performed on the device. The I/O costs associated with storing the full ERI tensor on disk are disruptive to the design of an efficient GPU-accelerated algorithm, so our most recent efforts emphasize the use of DF or CD approximations to the ERI tensor. Approximate three-index tensors also mitigate host/device communication costs.

Naively, one can make use of GPUs in a CCSD algorithm by performing all tensor contractions on the device using CUBLAS DGEMM calls. For each contraction, input/output buffers must be transferred to/from the device, and contractions must be tiled to account for the limited global memory on the GPU. This strategy suffers from two obvious deficiencies. First, we have ignored the overhead associated with data transfers. Second, valuable CPU cycles are wasted if the CPU is idle during GPU-driven tensor contractions. We can address this first deficiency, partially masking the overhead of data transfers, by interleaving communication and computation. A very simple blocked algorithm that interleaves cudaMemcpyAsync and cublasDgemm calls is outlined in Figure 13.1.

Figure 13.2a illustrates the performance of this blocked algorithm that interleaves communication and computation. For a matrix multiply involving square matrices with dimensions of 15,000, 6 Intel Core i7-3930K cores, 1 Fermi GPU, 2 Fermi GPUs, and 1 Kepler GPU achieve 142, 285, 553, and 859 GF (10^9 floating-point operations per second), respectively. As Figure 13.2b shows, using CUBLAS DGEMM in this way results in very modest accelerations over the Intel MKL implementation of DGEMM executed on Core i7-3930K processors; for $15{,}000 \times 15{,}000$ matrices, we observe only 2.0×, 3.9×, and 6.0× accelerations over MKL DGEMM when using 1 Fermi GPU, 2 Fermi GPUs, or 1 Kepler GPU, respectively. The ratio of performance of the interleaved algorithm to one making direct use of CUBLAS DGEMM without such considerations is illustrated in Figure 13.2c. For modest-sized tensor contractions, involving matrix dimensions on the order of 1000–2000, this simple interleaving strategy masks communication quite well, boosting performance over the most naive use of CUBLAS DGEMM by as much as 40%.

In a GPU-enabled DF/CD-CCSD implementation, we do not expect to observe accelerations as large as a factor of 4 or 6. First, not all tensor contractions that arise in CCSD are as regular as those examined in Figure 13.2. Second, many kernels in a CCSD computation do not benefit from the use of GPUs. For example, the DIIS convergence acceleration procedure is completely I/O bound. Reasonable accelerations can only be achieved with an algorithm that considers the strategies of the previous sections and distributes entire diagrams between GPU and CPU.

The leading contribution to the scaling of the DF/CD-CCSD algorithm involves the evaluation of the double-particle ladder diagram: $\sum_{cd} t_{ij}^{cd} \sum_Q B_{ac}^Q B_{bd}^Q$ (the term A_{ij}^{ab} earlier). This contraction scales as $\frac{1}{2} o^2 v^4$, and the construction of the four-external-index block of the ERI tensor scales as $v^4 N_{\text{aux}}$, where N_{aux} is the dimension of the auxiliary basis set in a DF-CCSD computation or the number of Cholesky vectors in a CD-CCSD computation. Using the symmetric and antisymmetric tensors of Ref. [38], we can reduce the cost of the construction of the integrals and the subsequent contraction with the CCSD amplitude by a factor of 2. Even with these techniques, the evaluation of this diagram dominates the

```
// OpenMP parallelization for multiple GPUs over M and N blocks
#pragma omp parallel for schedule (static) num_threads(num_gpus)
for M ∈ blocksM, N ∈ blocksN do
    // copy first tile of A and B to device
    cudaMemcpyAsync Ablock → gpuA_curr [sizeK*sizeM elements]
    cudaMemcpyAsync Bblock → gpuB_curr [sizeK*sizeN elements]
    for K ∈ blocksK do
        // two OpenMP threads for interleaving
        #pragma omp parallel num_threads(2)
        if thread == 0 then
            // DGEMM for current block using GPU stream 0
            cublasDgemm gpuA_curr·gpuB_curr → gpuC, O(sizeM*sizeN*sizeK)
        else
            // copy next tile (if it exists) using GPU stream 1
            cudaMemcpyAsync Ablock → gpuA_next [sizeK*sizeM elements]
            cudaMemcpyAsync Bblock → gpuB_next [sizeK*sizeN elements]
        end if
    end for
    // copy result back to host
    cudaMemcpyAsync gpuC → Cblock [sizeM*sizeN elements]
end for
```

Figure 13.1 *A simple interleaved DGEMM algorithm. For a matrix multiplication, $C_{mn} = \sum_k A_{mk} B_{kn}$, the m and n dimensions are blocked to parallelize the algorithm over multiple GPUs and to guarantee that all blocks will fit in GPU global memory. The sum dimension, k, is blocked in order to interleave computation and communication. From Ref. [5]*

cost of a DF/CD-CCSD algorithm; hence, our strategy is to evaluate A_{ij}^{ab} on the GPU while evaluating all other terms on the CPU. If the GPU finishes its work before the CPU, the algorithm automatically switches to one similar to that outlined in Figure 13.1 to evaluate the remaining diagrams. Likewise, if the CPU finishes its tasks before the GPU completes the evaluation A_{ij}^{ab}, the otherwise idle CPU cores assist in the evaluation of this diagram. The algorithm for evaluating A_{ij}^{ab} using GPU hardware is given in Figure 13.3.

13.5 Computational Details

13.5.1 Conventional CCD and CCSD

The spin-free CCSD equations presented in Eqs. (13.5)–(13.15) were implemented in the PSI3 electronic structure package [39]. All hybrid GPU/CPU computations were performed using the Dirac GPU testbed system at NERSC, a single node of which consists of two Intel Xeon X5550 CPUs and one Nvidia Tesla C2050 (Fermi) GPU. This combination of hardware is a good model for what will be a standard node on next-generation supercomputers, such as the Titan system at Oak Ridge National Laboratory. Computations using CC implementations in well-known electronic structure packages (PSI3, NWChem [40], and Molpro [41]) were performed using the same CPUs. For these experiments, both hybrid and pure-CPU computations were performed using a single node; GPU+CPU computations use a single Tesla C2050 GPU and two Intel Xeon CPUs, and CPU computations use two Intel Xeon CPUs. Both Molpro and NWChem make use of process-based parallelism through

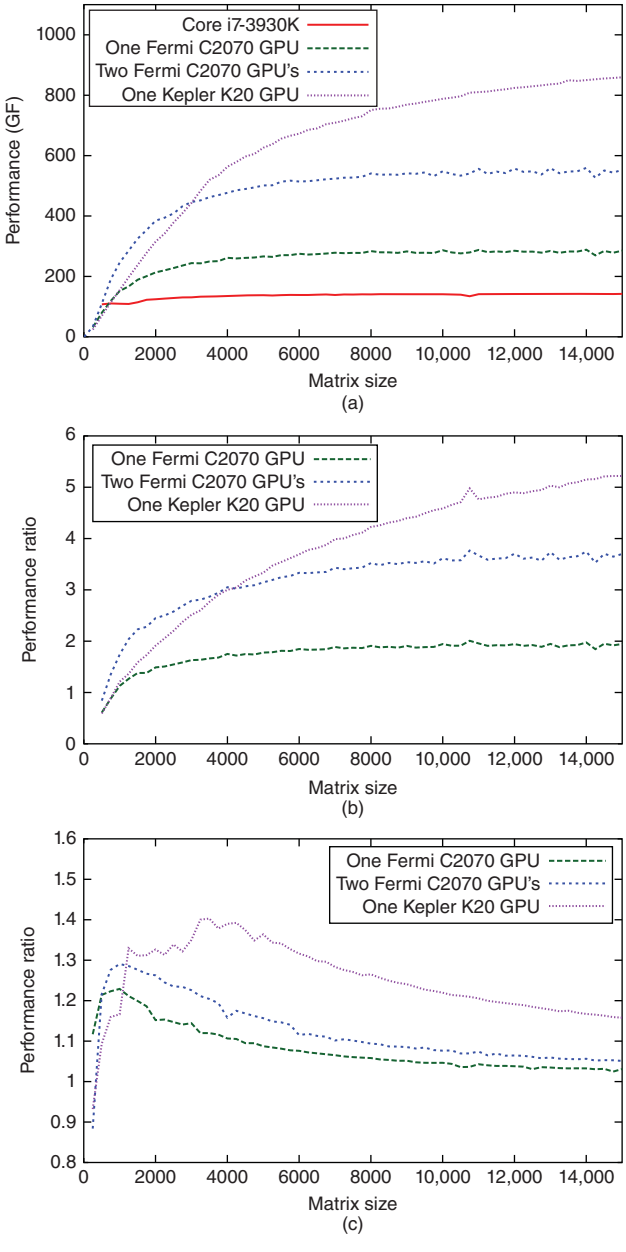


Figure 13.2 *The (a) performance (in GF) of the blocked DGEMM algorithm outlined in Figure 13.1. The performance of Fermi and Kepler GPUs are illustrated relative to that of the Intel Core i7 3930K CPU in (b). The performance boost achieved by interleaving communication and computation is shown in (c)*

```
// A CPU thread drives each GPU
if driving a GPU then
    gpu_done = false
    // Copy t2(±) to GPU.
    cudaMemcpy t2(+)→GPU [o(o + 1)v(v + 1)/4 elements]
    cudaMemcpy t2(-)→GPU [o(o + 1)v(v + 1)/4 elements]

    // OpenMP parallelization over multiple GPUs
    #pragma omp parallel for schedule(dynamic) num_threads(num_gpus)
    for a ∈ v do
        // Block b dimension so 3-index tensors fit in GPU global memory.
        // The maximum possible block size, N_b, is v − a.
        for B ∈ blocksB and b > a do
            // For the current a, copy all N_aux v integrals of B^Q_ac.
            cudaMemcpyAsync B^Q_ac → GPU [N_aux v elements]
            // For the current block, B, copy N_aux v N_b integrals of B^Q_bd.
            cudaMemcpyAsync B^Q_bd → GPU [N_aux v N_b elements]
            // Build subset of (ac|bd) tensor, V_bdc.
            cublasDgemm, O(v^2 N_b N_aux)
            // Build (±) integral tensors (on the device).
            V(±)_bcd = V_bdc ± V_bcd
            // Contract integrals with amplitudes (+).
            cublasDgemm O(o(o + 1)v(v + 1)N_b/4)
            // Copy residual, R, back to host and accumulate residual.
            cudaMemcpyAsync R → CPU [o ∗ (o + 1)/2 ∗ N_b elements]
            // Contract integrals with amplitudes (-).
            cublasDgemm O(o(o + 1)v(v + 1)N_b/4)
            // Copy residual, R, back to host and accumulate.
            cudaMemcpyAsync R → CPU [o ∗ (o + 1)/2 ∗ N_b elements]
        end for
    end for
    gpu_done = true
else
    // the remaining (num_gpu - num_cpu) threads evaluate other CC diagrams
    if gpu_done == false then
        CPU cores evaluate a diagram (or a block of a diagram)
    else
        break
    end if
end if
// if any work is remaining, use interleaved GPU DGEMM algorithm
```

Figure 13.3 *Pseudocode for the DF/CD-CCSD procedure. The ladder diagram, A_{ij}^{ab}, is evaluated on the GPU using the symmetric and antisymmetric tensors defined in Ref. [38] (here denoted by (±)). From Ref. [5]*

the Global Arrays toolkit [42, 43]. For all computations using these packages, seven of eight cores were dedicated to computations, while one was reserved as a communication helper thread. It has been shown that NWChem performs as well (and often significantly better) when one core is dedicated to communication [44]. The native PSI3 CCSD implementation and our own CPU code utilize shared-memory parallelism through threaded BLAS calls of the GotoBLAS2 library. For consistency, all computations with all packages were performed using C_1 point-group symmetry.

13.5.2 Density-Fitted CCSD

The computations using our density-fitted CCSD algorithm were performed on a single workstation consisting of a 6-core Intel Core i7-3930K (3.20 GHz) CPU with access to 64 GB RAM and either a single Nvidia Tesla K20 (Kelper) GPU or two Nvidia Tesla C2070 (Fermi) GPUs. The GPU-accelerated DF-CCSD algorithm was implemented in a development version of the Psi4 electronic structure package [45]. All computations were performed within the frozen core approximation. For an analysis of the performance of the underlying DF-CCSD CPU implementation, the reader is directed to Ref. [35]. Again, all computations were performed using C_1 point-group symmetry.

13.6 Results

13.6.1 Communication-Avoiding CCD

The performance of the low-communication GPU-CCD algorithm is compared to two well-known electronic structure packages in Table 13.1. We consider hydrocarbons with as many as 20 carbon atoms described by the modest 6-31G basis set. Timings correspond to only the iterative portions of the CCD algorithm. For the GPU implementation, the initial integral push to the device is excluded. Also, the CPU and GPU implementations of Ref. [3] do not involve any I/O-intensive procedures such as the DIIS convergence acceleration. For Molpro and NWChem, the times corresponding to integral generation and sorting were excluded. The C2050 GPU-CCD algorithm consistently outperforms all other implementations on a per iteration basis.

Table 13.1 *Comparison of CPU and GPU implementations of CCD*

			Ref. [3]		X5550	
Molecule	o	v	C2050	X5550	Molpro	NWChem
C_8H_{10}	21	63	0.3	1.3	2.3	5.1
$C_{10}H_8$	24	72	0.5	2.5	4.8	10.6
$C_{10}H_{12}$	26	78	0.8	3.5	7.1	16.2
$C_{12}H_{14}$	31	93	2.0	10.0	17.6	42.0
$C_{14}H_{10}$	33	99	2.7	13.9	29.9	59.5
$C_{14}H_{16}$	36	108	4.5	21.6	41.5	90.2
C_{20}	40	120	8.8[a]	40.3	103.0	166.3
$C_{16}H_{18}$	41	123	10.5[a]	50.2	83.3	190.8
$C_{18}H_{12}$	42	126	12.7[a,b]	50.3	111.8	218.4
$C_{18}H_{20}$	46	138	20.1[a,b]	86.6	157.4	372.1

Timings per CC iteration are given in seconds. The letters o and v represent the number of doubly occupied and virtual orbitals in each system, respectively.
[a] The matrix–matrix multiplication involving $(ac|bd)$ was tiled.
[b] Some two-electron integrals were pushed to the GPU every iteration.

Using a single C2050 Fermi GPU, we observe improvements in the CCD iteration times of 4.0–5.2× relative to the threaded (eight threads) CPU implementation. This acceleration is consistent with the relative performance of DGEMM using the GotoBLAS2 and CUBLAS implementations of DGEMM on the CPU and GPU, respectively. As stated earlier, the limited global memory of the GPU requires that, for larger calculations, the tensor contraction involving the v_{cd}^{ab} block of integrals be tiled. For $C_{18}H_{20}$, this block of integrals requires 2.7 GB storage. The time given in Table 13.1 includes this integral transfer and the GPU implementation still performs 4.3× better than the CPU code. An average CCD iteration is anywhere from 8 to 12× faster when using the C2050 algorithm, relative to that in the Molpro package. For $C_{18}H_{20}$, a single iteration of the C2050 algorithm requires 20.1 seconds, while an average CCD iteration in Molpro requires 2.5 minutes. The comparison to Molpro is not necessarily a fair one, and it does not accurately reflect the utility of GPU processors relative to multicore CPUs. Our communication-avoiding GPU-CCD algorithm involves zero I/O, and meaningful comparisons should only be drawn between it and our corresponding CPU algorithm. The accelerations we observe (4–5×) are consistent with the relative performance of DGEMM on GPU and CPU processors and represent an approximation to the upper-bound on performance that could be expected in a production-level iterative CC algorithm.

Figure 13.4 illustrates the double-precision floating-point performance for the CPU and GPU-CCD algorithms. The CPU algorithm achieves only 57 GF, while the GPU can achieve nearly 250 GF, which corresponds to 50% of the theoretical peak performance for the C2050 GPU. Performance as a function of system size is more varied for the GPU than the CPU, but we find that this is less of a problem for Fermi hardware after the release of CUDA 3.2 than with older generations of Tesla products and older versions of CUDA. In fact, in our original implementation of GPU-CCD (before the release of CUDA 3.2), performance was tightly coupled to system size, with large increases in performance when matrix sizes matched GPU warp sizes. At that time, significant performance increases could be obtained by padding the occupied and virtual spaces in an effort to yield warp-matched matrix dimensions. With the release of CUDA 3.2, however, padding seems to have been built directly into the DGEMM implementation; padding the occupied and virtual spaces has little positive effect.

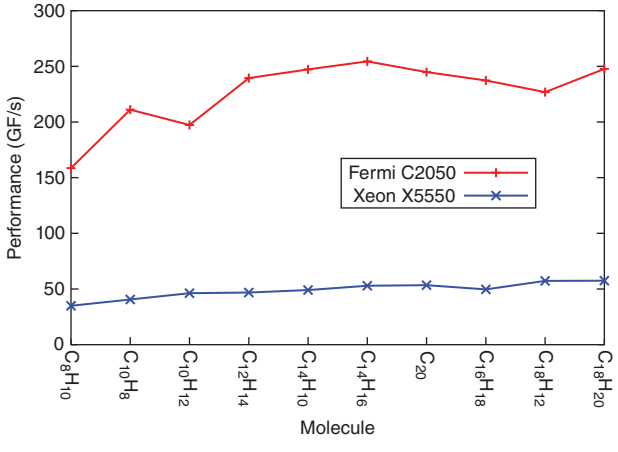


Figure 13.4 *Performance in GF (10^9 floating point operations per second) for different implementations of spin-free CCD. Results are given for both CPU and GPU hardware, and CPU BLAS routines utilize eight threads*

13.6.2 Low-Storage CCD and CCSD

Avoiding communication altogether is a short-sighted approach to developing extensible GPU CC software. The issue of limited global memory must eventually be addressed, and repeated memory transfers will become necessary. Fortunately, the data at the bottom of Table 13.1 suggest that this data motion might not completely negate the benefits of the GPU.

In this section, we illustrate the performance of a spin-free CCSD algorithm that stresses the importance of limited global memory over the desire to reduce communication. Essentially all data involved in the CC equations (the ERI tensor and CC amplitudes) are transfered at least once to the GPU during each iteration. We mask the overhead of these transfers by simultaneously performing operations on both CPU and GPU, and the larger number of small terms that arise in CCSD afford ample opportunity to mask the costs of any host/device communication. The performance of the hybrid GPU/CPU CCSD algorithm is compared on a per-iteration basis to the pure CPU algorithm and several well-known electronic structure packages in Table 13.2. We consider the same set of hydrocarbons presented in Table 13.1. The hybrid GPU/CPU algorithm consistently outperforms all other CCSD implementations on a per iteration basis. In particular, the hybrid algorithm is found to be 2.8–4.1× faster than the corresponding pure CPU algorithm, and anywhere from 4.8× to 10.6× faster per iteration than Molpro. Note that, in the small molecule limit, the hybrid CCSD algorithm does not perform nearly as well relative to the CPU codes as the low-communication GPU-CCD algorithm. While the hybrid algorithm is never slower than the CPU algorithm, the modest performance improvement for the smaller systems suggest that these systems simply do not have enough work for the GPU to do to fully mask the transfer overhead. However, in the large-molecule limit, the relative performance of the hybrid algorithm increases, and we observe accelerations that we expect from the relative performance of DGEMM on the CPU and GPU. Hence, for sufficiently large systems, transfer times can be effectively masked by overlapping CPU and GPU computations.

Most chemical applications that use CCSD require much larger basis sets than the 6-31G basis used in Table 13.2 for accurate results. As such, many electronic structure packages optimize their CCSD codes for very large basis sets where $v \gg o$. In Table 13.3, we present timings for several molecules in much larger basis sets. Consider methanol, CH_3OH, described by an augmented triple-zeta basis set. Here the virtual space is 25× larger than the occupied space, which is much more typical in a state-of-the-art CC application. The hybrid code is less than 2× faster than the pure CPU code and only marginally faster than Molpro. It is tempting to attribute the poor performance to the relative

Table 13.2 *Comparison of CPU[a] and GPU[b] implementations of CCSD*

								GPU speedup	
Molecule	o	v	GPU[b]	CPU[b]	Molpro	NWChem	PSI3	CPU	Molpro
C_8H_{10}	21	63	0.5	1.4	2.4	8.4	7.9	2.80	4.80
$C_{10}H_8$	24	72	0.8	2.4	5.1	16.8	17.9	3.00	6.38
$C_{10}H_{12}$	26	78	1.3	3.8	7.2	25.2	23.6	2.92	5.54
$C_{12}H_{14}$	31	93	3.0	10.1	19.0	64.4	54.2	3.37	6.33
$C_{14}H_{10}$	33	99	3.9	14.0	31.0	90.7	61.4	3.59	7.94
$C_{14}H_{16}$	36	108	5.7	21.5	43.1	129.2	103.4	3.77	7.56
C_{20}	40	120	9.6	38.0	102.0	233.9	162.6	3.96	10.63
$C_{16}H_{18}$	41	123	11.6	45.9	84.1	267.9	192.4	3.96	7.25
$C_{18}H_{12}$	42	126	12.9	50.9	116.2	304.5	216.4	3.95	9.01
$C_{18}H_{20}$	46	138	20.8	84.5	161.4	512.0	306.9	4.06	7.76

Timings per CC iteration are given in seconds. GPU speedup signifies the relative cost of CPU algorithms as compared to the C2050 algorithm.

[a] Using all eight cores of two Intel Xeon X5550 CPUs.

[b] Using a single Nvidia C2050 GPU and two Intel Xeon X5550 CPUs.

Table 13.3 *Comparison of CPU[a] and GPU[b] implementations of CCSD in large basis sets*

				Iteration time (seconds)			GPU speedup	
Molecule	Basis	o	v	GPU[b]	CPU[a]	Molpro	CPU	Molpro
CH_3OH	aug-cc-pVTZ	7	175	1.7	3.1	2.8	1.82	1.65
C_6H_6	aug-cc-pVDZ	15	171	4.2	10.9	17.4	2.60	4.14
$CH_3OSOOCH_3$	aug-cc-pVDZ	23	167	7.9	25.7	31.3	3.25	3.94
$C_{10}H_{12}$	cc-pVDZ	26	164	8.6	30.9	56.8	3.59	6.60

[a]Using all eight cores of two Intel Xeon X5550 CPUs.
[b]Using a single Nvidia C2050 GPU and two Intel Xeon X5550 CPUs.

sizes of the occupied and virtual spaces and thus the shape of the DGEMMs that arise in the CCSD algorithm. Many of the tensor contractions in Eqs. (13.5)–(13.15) are far from square, particularly the particle–particle ladder diagram. However, the timings for $C_{10}H_{12}$ tell a very different story. This molecule is present in both Tables 13.2 and 13.3, first with 78 virtual orbitals and then with 164 virtual orbitals. We see that more than doubling the size of the virtual space results in greater efficiency of the hybrid code relative to both the pure CPU code and to Molpro. The performance of the hybrid code is nearly independent of the relative sizes of the occupied and virtual spaces, and what actually determines performance is the absolute size of these spaces. Each space must be large enough that there is enough work to make efficient use of the GPU.

13.6.3 Density-Fitted CCSD

In the previous sections, we have described out efforts to establish a reasonable upper bound to the potential performance advantages of GPUs over CPUs when solving the coupled-cluster equations. Our proof-of-principle codes demonstrate that modest (4–5×) accelerations can be achieved by carefully managing data motion and overlapping communication and computation. In this section, we demonstrate that an efficient production-level GPU-accelerated CCSD implementation can be achieved by adopting approximate representations of the ERI tensor, in the form of either DF or CD. Our base algorithm is optimized for modern multicore CPUs, and it is competitive with other, well known implementations of CCSD (performance data, in terms of the accuracy of the DF/CD approximations and the efficiency of the algorithm, can be found in Ref. [35]). For a benzene trimer, represented by the aug-cc-pVDZ basis set, a single iteration of the DF-CCSD algorithm requires 2150 seconds when using all six cores of the Core i7-3930K processor. A single iteration of the CCSD algorithm in the Molpro [41] electronic structure package requires roughly 1.83× more wall time using the same resources. For this example, we use DF in the CCSD equations, and the auxiliary basis set is the aug-cc-pVDZ-RI basis set [46], optimized for density-fitted (or resolution of the identity, RI) second-order perturbation theory (RI-MP2). Hence, the present DF-CCSD implementation is a well-optimized base algorithm against which the GPU implementation can be fairly judged. Furthermore, unlike the algorithms described previously in this chapter, the present CPU and GPU codes both utilize DIIS convergence acceleration, and all timings presented include this I/O dominated procedure.

Figure 13.5 illustrates the acceleration observed for GPU-enabled DF-CCSD using one Fermi GPU, two Fermi GPUs, or a single Kepler GPU. This performance analysis suggests that there is little reason to use GPUs for very small systems (e.g., five water molecules); a 1.2× acceleration does not justify the cost of high-end compute-oriented graphics processors. However, for the largest systems studied here, a single Fermi GPU can double the speed of a DF-CCSD iteration, and two Fermi GPUs or a Kepler GPU can provide roughly 3× acceleration. These tests all utilize the cc-pVDZ basis set (and the cc-pVDZ-RI auxiliary basis set), and $(H_2O)_{20}$ has 460 active basis functions.

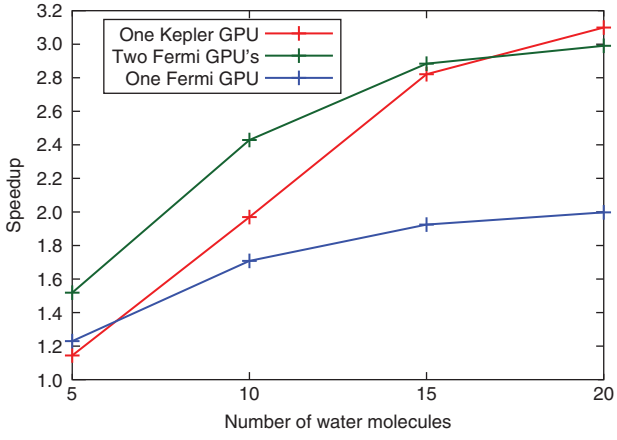


Figure 13.5 *Speedup in the average DF-CCSD iteration time for clusters of water molecules (represented by a cc-pVDZ basis set, using a cc-pVDZ-RI auxiliary basis set) using one Fermi C2070 GPU, two Fermi C2070 GPUs, or one Kepler K20c GPU. The speedup is defined relative to the average iteration time for DF-CCSD using all six cores of a Core i7-3930K processor*

Table 13.4 *Average iteration time (in seconds) for DF/CD-CCSD computations of adenine-thymine and a benzene trimer represented by the aug-cc-pVDZ basis and the uracil dimer represented by the aug-cc-pVTZ basis*[a]

	Adenine-thymine[b]			Benzene trimer[c]			Uracil dimer[d,e]		
	A_{ij}^{ab}	Total	Speedup	A_{ij}^{ab}	Total	Speedup	A_{ij}^{ab}	Total	Speedup
Core i7-3930K	1134	2156	–	1665	2719	–	8590	11924	–
One Fermi C2070	703	1054	2.05	1082	1298	2.09	6095	6741	1.77
Two Fermi C2070	349	747	2.89	546	958	2.84	4159	4791	2.49
Kepler K20c	457	789	2.73	648	995	2.73	3826	4724	2.52

Computations using GPUs also made use of the host Intel Core i7 CPU during the iterations.
[a]From Ref. [5].
[b]Using DF in the SCF (aug-cc-pVDZ-JK auxiliary basis) and CCSD (aug-cc-pVDZ-RI auxiliary basis) procedures.
[c]Using CD (10^{-4} error threshold) in the SCF and CCSD procedures.
[d]Using DF in the SCF (aug-cc-pVTZ-JK auxiliary basis) and CCSD (aug-cc-pVTZ-RI auxiliary basis) procedures.
[e]Using frozen natural orbitals with a conservative 10^{-6} occupancy threshold.

Table 13.4 provides average iteration times for the GPU-enabled DF/CD-CCSD algorithm described above using several different GPU/CPU configurations. For our test cases, we have chosen several nonbonded complexes: the hydrogen-bonded configuration of adenine-thymine [47], a uracil dimer [47], and a benzene trimer whose coordinates are taken from the crystal structure of Bacon *et al*. [48]. Adenine-thymine and the benzene trimer are represented by an aug-cc-pVDZ basis set and have 517 and 558 active orbitals, respectively. The uracil dimer is represented by the larger aug-cc-pVTZ basis set. We use frozen natural orbital (FNO) techniques to truncate the virtual space [49–54] (with a conservative occupation threshold of 10^{-6}); the resulting system has 822 active basis functions. For the molecules represented by the aug-cc-pVDZ basis set, we see that the use of a single Fermi C2070 GPU doubles the efficiency of the computation. The addition of a second Fermi only improves the total iteration speed by factors of 1.41 and 1.35 for adenine-thymine and the benzene trimer, respectively, but, interestingly, we see that the evaluation of the A_{ij}^{ab} diagram

scales almost perfectly going from one to two Fermi GPUs. The total iteration time does not improve by a factor of 2 because some time-consuming steps do not make use of available GPU resources. For example, these average iteration times include all tasks that arise in a standard CCSD algorithm, including the I/O-intensive DIIS convergence acceleration procedure. For the uracil dimer represented by the larger aug-cc-pVTZ basis set, we observe similar performance for each graphics processor. A single Fermi GPU provides a modest acceleration of 1.77×, while two Fermi GPUs or a Kepler provide similar accelerations of about 2.5× relative to the base, CPU-only implementation.

GPU global memory is and will likely remain quite limited as compared to that available on the host CPU. For example, the K20c and C2070 GPUs have only 4.6 and 5.25 GB of global memory, respectively (with ECC memory enabled). We have designed our implementation with these limitations in mind. The simple blocked DGEMM algorithm presented in Figure 13.1 is tiled such that GPU global memory can accomodate all input and output buffers. In the DF/CD-CCSD implementation, the evaluation of the A_{ij}^{ab} diagram should be similarly blocked. As shown in Figure 13.3, the present algorithm requires that $\frac{1}{2}o^2v^2$ double precision numbers fit on the device. This assumption is not unreasonable for systems with up to roughly 800 active basis functions (like the uracil dimer with aug-cc-pVTZ basis). However, at some point, this algorithm will fail because we cannot store $\frac{1}{2}o^2v^2$ double precision numbers on the GPU. To avoid this limitation, we could block the diagram over all unique ij pairs; depending on the loop structure, this choice would result in either (i) redundant construction of the $(vv|vv)$ block of the ERI tensor from three-index integrals or (ii) the repeated transfer of the coupled-cluster amplitudes to the device. Alternatively, we could choose a subset of ij pairs for which the evaluation of A_{ij}^{ab} is possible on the GPU and evaluate the remaining terms using CPU resources (or other GPUs). This blocking structure is a natural starting point for future distributed parallel GPU-accelerated DF/CD-CCSD implementations.

13.7 Conclusions

We have reviewed several strategies for porting iterative coupled-cluster methods to heterogenous computing environments. Our initial investigation was based on a minimal communication model in which the CCD equations were solved entirely on a GPU. From a chemical perspective, the implementation was of little use, as only very small systems could be treated. However, the implementation established a true upper-bound to the performance increases that one can reasonably expect when executing iterative coupled cluster on graphics processors; the expected acceleration is tied to the relative performance of DGEMM on the CPU and GPU. Next, we explored the effects of data transfers on the performance of a spin-free CCSD algorithm. Again, for modest systems, the relative GPU/CPU performance for the overall CCSD algorithm was very similar to that of the implementations of DGEMM upon which the algorithms were built. Importantly, we demonstrated that careful algorithm design, particularly with respect to the simultaneous use of both CPU and GPU processors, can mitigate the cost of large amounts of data motion across the PCI bus. Two very important (and somewhat obvious, in hindsight) conclusions can be drawn from these initial explorations: (i) work is best distributed between CPU and GPU in terms of entire diagrams and (ii) asynchronous memory transfers are absolutely necessary to effectively mask the cost associated with moving data between host and device. Finally, we presented a robust implementation of GPU CCSD that utilized either DF or CD approximations to factorize the ERI tensor. The DF/CD approximations substantially reduce the amount of data that must be transferred to the device each iteration and, coupled with a t_1-transformation of the Hamiltonian, greatly simplify the spin-free CCSD equations. Using two Fermi C2070 GPUs or a single Kepler K20c GPU, the CCSD equations could be solved roughly 3× more efficiently than when utilizing all six cores of an Intel Core i7 3930k CPU. The accelerations from GPUs were not as impressive as those observed for our initial, proof-of-concept algorithms, but

this reduction in performance is an unavoidable consequence of extending the implementation to the treatment of more than 800 active orbitals. A threefold acceleration may seem modest, but, given that CCSD computations can take days or weeks on a single workstation, the observed speedup is of real practical benefit. The present algorithm achieves this performance by (i) interleaving computation and communication in tensor contractions performed on the device and (ii) ensuring that no CPU cores are idle while the GPU evaluates the most computationally demanding diagrams.

Acknowledgments

AED acknowledges support from the National Science Foundation American Competitiveness in Chemistry Postdoctoral Fellowship (CHE-1137288) and the Computational Postdoctoral Fellowship program at Argonne National Laboratory. CDS acknowledges support from the National Science Foundation (ACI-1147843). This research used the "Dirac" GPU testbed system of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under contract no. DE-AC02-05CH11231. This research used resources of the Argonne Leadership Computing Facility (ALCF) and Laboratory Computing Resource Center at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract no. DE-AC02-06CH11357.

References

1. Macedonia, M. (2003) The GPU enters computing's mainstream. *Computer*, **36**, 106–108.
2. Nickolls, J. and Dally, W. (2010) The GPU computing era. *IEEE Micro*, **30**, 56–69.
3. DePrince, A.E. and Hammond, J.R. (2011) Coupled cluster theory on graphics processing units I. The coupled cluster doubles method. *J. Chem. Theory Comput.*, **7**, 1287–1295.
4. DePrince, A. and Hammond, J. (2011) Quantum chemical many-body theory on heterogeneous nodes. Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on, pp. 131–140.
5. DePrince, A.E., Kennedy, M.R., Sumpter, B.G. and Sherrill, C.D. (2014) Density-fitted singles and doubles coupled cluster on graphics processing units. *Mol. Phys.*, **112**, 844–852.
6. Purvis, G.D. and Bartlett, R.J. (1982) A full coupled-cluster singles and doubles model: the inclusion of disconnected triples. *J. Chem. Phys.*, **76**, 1910–1918.
7. Raghavachari, K., Trucks, G.W., Pople, J.A. and Head-Gordon, M. (1989) A fifth-order perturbation comparison of electron correlation theories. *Chem. Phys. Lett.*, **157**, 479–483.
8. Ma, W., Krishnamoorthy, S., Villa, O. and Kowalski, K. (2010) Acceleration of streamed tensor contraction expressions on GPGPU-based clusters. IEEE International Conference on Cluster Computing, pp. 207–216.
9. Ma, W., Krishnamoorthy, S., Villa, O. and Kowalski, K. (2011) GPU-based implementations of the noniterative regularized-CCSD(T) corrections: applications to strongly correlated systems. *J. Chem. Theory Comput.*, **7**, 1316–1327.
10. Asadchev, A. and Gordon, M.S. (2013) Fast and flexible coupled cluster implementation. *J. Chem. Theory Comput.*, **9**, 3385–3392.
11. Melicherčík, M., Demovič, L. and Michal Pitoňák, P.N. (2010) Acceleration of CCSD(T) computations using technology of graphical processing unit.
12. Pulay, P. (1980) Convergence acceleration of iterative sequences. The case of SCF iteration. *Chem. Phys. Lett.*, **73**, 393–398.
13. Scuseria, G.E., Lee, T.J. and Schaefer, H.F. (1986) Accelerating the convergence of the coupled-cluster approach. The use of the DIIS method. *Chem. Phys. Lett.*, **130**, 236–239.

14. Piecuch, P., Kucharski, S.A., Kowalski, K. and Musiał, M. (2002) Efficient computer implementation of the renormalized coupled-cluster methods: the R-CCSD[T], R-CCSD(T), CR-CCSD[T], and CR-CCSD(T) approaches. *Comput. Phys. Commun.*, **149**, 71–96.
15. Scuseria, G.E. and Schaefer, H.F. III (1989) Is coupled cluster singles and doubles (CCSD) more computationally intensive than quadratic configuration interaction (QCISD)? *J. Chem. Phys.*, **90**, 3700–3703.
16. Koch, H., Christiansen, O., Kobayashi, R., Jørgensen, P. and Helgaker, T. (1994) A direct atomic orbital driven implementation of the coupled-cluster singles and doubles (CCSD) model. *Chem. Phys. Lett.*, **228**, 233–238.
17. Whitten, J.L. (1973) Coulombic potential-energy integrals and approximations. *J. Chem. Phys.*, **58**, 4496–4501.
18. Dunlap, B.I., Connolly, J.W.D. and Sabin, J.R. (1979) On some approximations in applications of $X\alpha$ theory. *J. Chem. Phys.*, **71**, 3396–3402.
19. Feyereisen, M., Fitzgerald, G. and Komornicki, A. (1993) Use of approximate integrals in Ab initio theory. An application in MP2 calculations. *Chem. Phys. Lett.*, **208**, 359–363.
20. Vahtras, O., Almlöf, J. and Feyereisen, M.W. (1993) Integral approximations for LCAO-SCF calculations. *Chem. Phys. Lett.*, **213**, 514–518.
21. Rendell, A.P. and Lee, T.J. (1994) Coupled-cluster theory employing approximate integrals: an approach to avoid the input/output and storage bottlenecks. *J. Chem. Phys.*, **101**, 400–408.
22. Weigend, F. (2002) A fully direct RI-HF algorithm: implementation, optimized auxiliary basis sets, demonstration of accuracy and efficiency. *Phys. Chem. Chem. Phys.*, **4**, 4285–4291.
23. Sodt, A., Subotnik, J.E. and Head-Gordon, M. (2006) Linear scaling density fitting. *J. Chem. Phys.*, **125**, 194109.
24. Werner, H.-J., Manby, F.R. and Knowles, P.J. (2003) Fast linear scaling second-order Møller-Plesset perturbation theory (MP2) using local and density fitting approximations. *J. Chem. Phys.*, **118**, 8149–8160.
25. Beebe, N.H.F. and Linderberg, J. (1977) Simplifications in the generation and transformation of two-electron integrals in molecular calculations. *Int. J. Quantum Chem.*, **12**, 683–705.
26. Roeggen, I. and Wisloff-Nilssen, E. (1986) On the Beebe-Linderberg 2-electron integral approximation. *Chem. Phys. Lett.*, **132**, 154–160.
27. Koch, H., de Meras, A.S. and Pedersen, T.B. (2003) Reduced scaling in electronic structure calculations using Cholesky decompositions. *J. Chem. Phys.*, **118**, 9481–9484.
28. Aquilante, F., Pedersen, T.B. and Lindh, R. (2007) Low-cost evaluation of the exchange Fock matrix from Cholesky and density fitting representations of the electron repulsion integrals. *J. Chem. Phys.*, **126**, 194106.
29. Hohenstein, E.G., Parrish, R.M. and Martínez, T.J. (2012) Tensor hypercontraction density fitting. I. Quartic scaling second- and third-order Møller-Plesset perturbation theory. *J. Chem. Phys.*, **137**, 044103.
30. Parrish, R.M., Hohenstein, E.G., Martínez, T.J. and Sherrill, C.D. (2012) Tensor hypercontraction. II. Least-squares renormalization. *J. Chem. Phys.*, **137**, 224106.
31. Christiansen, O., Koch, H. and Jørgensen, P. (1995) The second-order approximate coupled cluster singles and doubles model CC2. *Chem. Phys. Lett.*, **243**, 409–418.
32. Hattig, C. and Weigend, F. (2000) CC2 excitation energy calculations on large molecules using the resolution of the identity approximation. *J. Chem. Phys.*, **113**, 5154–5161.
33. Pitonak, M., Aquilante, F., Hobza, P., Neogrady, P., Noga, J. and Urban, M. (2011) Parallelized implementation of the CCSD(T) method in MOLCAS using optimized virtual orbitals space and Cholesky decomposed two-electron integrals. *Collect. Czech. Chem. Commun.*, **76**, 713–742.
34. Boström, J., Pitoňák, M., Aquilante, F., Neogrády, P., Pedersen, T.B. and Lindh, R. (2012) Coupled cluster and Møller-Plesset perturbation theory calculations of noncovalent intermolecular

interactions using density fitting with auxiliary basis sets from Cholesky decompositions. *J. Chem. Theory Comput.*, **8**, 1921–1928.

35. DePrince, A.E. and Sherrill, C.D. (2013) Accuracy and efficiency of coupled-cluster theory using density fitting/Cholesky decomposition, frozen natural orbitals, and a t_1-transformed hamiltonian. *J. Chem. Theory Comput.*, **9**, 2687–2696.
36. Epifanovsky, E., Zuev, D., Feng, X., Khistyaev, K., Shao, Y. and Krylov, A.I. (2013) General implementation of the resolution-of-the-identity and Cholesky representations of electron repulsion integrals within coupled-cluster and equation-of-motion methods: theory and benchmarks. *J. Chem. Phys.*, **139**, 134105.
37. Schmidt, M.W., Baldridge, K.K., Boatz, J.A., Elbert, S.T., Gordon, M.S., Jensen, J.H., Koseki, S., Matsunaga, N., Nguyen, K.A., Su, S., Windus, T.L., Dupuis, M. and Montgomery, J.A. Jr. (1993) General atomic and molecular electronic structure system. *J. Comput. Chem.*, **14**, 1347–1363.
38. Scuseria, G.E., Janssen, C.L. and Schaefer, H.F. III (1988) An efficient reformulation of the closed-shell coupled cluster single and double excitation (CCSD) equations. *J. Chem. Phys.*, **89**, 7382.
39. Crawford, T.D., Sherrill, C.D., Valeev, E.F., Fermann, J.T., King, R.A., Leininger, M.L., Brown, S.T., Janssen, C.L., Seidl, E.T., Kenny, J.P. and Allen, W.D. (2007) PSI3: an open-source *Ab Initio* electronic structure package. *J. Comput. Chem.*, **28**, 1610–1616.
40. Bylaska, E.J., de Jong, W.A., Govind, N., Kowalski, K., Straatsma, T.P., Valiev, M., van Dam, H.J.J., Wang, D., Aprà, E., Windus, T.L., Hammond, J., Autschbach, J., Nichols, P., Hirata, S., Hackler, M.T., Zhao, Y., Fan, P.-D., Harrison, R.J., Dupuis, M., Smith, D.M.A., Nieplocha, J., Tipparaju, V., Krishnan, M., Vazquez-Mayagoitia, A., Wu, Q., Voorhis, T.V., Auer, A.A., Nooijen, M., Crosby, L.D., Brown, E., Cisneros, G., Fann, G.I., Früchtl, H., Garza, J., Hirao, K., Kendall, R., Nichols, J.A., Tsemekhman, K., Wolinski, K., Anchell, J., Bernholdt, D., Borowski, P., Clark, T., Clerc, D., Dachsel, H., Deegan, M., Dyall, K., Elwood, D., Glendening, E., Gutowski, M., Hess, A., Jaffe, J., Johnson, B., Ju, J., Kobayashi, R., Kutteh, R., Lin, Z., Littlefield, R., Long, X., Meng, B., Nakajima, T., Niu, S., Pollack, L., Rosing, M., Sandrone, G., Stave, M., Taylor, H., Thomas, G., van Lenthe, J., Wong, A. and Zhang, Z. (2010) *NWChem*, A Computational Chemistry Package for Parallel Computers, Version 6.0.
41. Werner, H.-J., Knowles, P.J., Manby, F.R., Schütz, M., Celani, P., Knizia, G., Korona, T., Lindh, R., Mitrushenkov, A., Rauhut, G., Adler, T.B., Amos, R.D., Bernhardsson, A., Berning, A., Cooper, D.L., Deegan, M.J.O., Dobbyn, A.J., Eckert, F., Goll, E., Hampel, C., Hesselmann, A., Hetzer, G., Hrenar, T., Jansen, G., Köppl, C., Liu, Y., Lloyd, A.W., Mata, R.A., May, A.J., McNicholas, S.J., Meyer, W., Mura, M.E., Nicklass, A., Palmieri, P., Pflüger, K., Pitzer, R., Reiher, M., Shiozaki, T., Stoll, H., Stone, A.J., Tarroni, R., Thorsteinsson, T., Wang, M. and Wolf, A. (2010) *MOLPRO*, version 2010.1, a package of Ab initio programs, see `http://www.molpro.net` (accessed 21 September 2015).
42. Nieplocha, J., Harrison, R.J. and Littlefield, R.J. Global arrays: a portable "shared-memory" programming model for distributed memory computers. Proceedings of the 1994 conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, Supercomputing '94, pp. 340–349.
43. Nieplocha, J., Harrison, R.J. and Littlefield, R.J. (1996) Global arrays: a nonuniform memory access programming model for high-performance computers. *J. Supercomput.*, **10**, 169–189.
44. Hammond, J.R., Krishnamoorthy, S., Shende, S., Romero, N.A. and Malony, A.D. (2011) Performance Characterization of Global Address Space Applications: a Case Study with NWChem.
45. Turney, J.M., Simmonett, A.C., Parrish, R.M., Hohenstein, E.G., Evangelista, F.A., Fermann, J.T., Mintz, B.J., Burns, L.A., Wilke, J.J., Abrams, M.L., Russ, N.J., Leininger, M.L., Janssen, C.L., Seidl, E.T., Allen, W.D., Schaefer, H.F., King, R.A., Valeev, E.F., Sherrill, C.D. and Crawford, T.D. (2012) Psi4: an open-source Ab initio electronic structure program. *Wiley Interdiscip. Rev. Comput. Mol. Sci.*, **2**, 556–565.

46. Weigend, F., Köhn, A. and Hättig, C. (2002) Efficient use of the correlation consistent basis sets in resolution of the identity MP2 calculations. *J. Chem. Phys.*, **116**, 3175–3183.
47. Jurečka, P., Šponer, J., Černý, J. and Hobza, P. (2006) Benchmark database of accurate (MP2 and CCSD(T) complete basis set limit) interaction energies of small model complexes, DNA base pairs, and amino acid pairs. *Phys. Chem. Chem. Phys.*, **8**, 1985–1993.
48. Bacon, G.E., Curry, N.A. and Wilson, S.A. (1964) A crystallographic study of solid benzene by neutron diffraction. *Proc. R. Soc. London, Ser. A*, **270**, 98–110.
49. Sosa, C., Geersten, J., Trucks, G.W., Barlett, R.J. and Franz, J.A. (1989) Selection of the reduced virtual space for correlated calculations - an application to the energy and dipole-moment of H_2O. *Chem. Phys. Lett.*, **159**, 148–154.
50. Klopper, W., Noga, J., Koch, H. and Helgaker, T. (1997) Multiple basis sets in calculations of triples corrections in coupled-cluster theory. *Theor. Chem. Acc.*, **97**, 164–176.
51. Taube, A.G. and Bartlett, R.J. (2005) Frozen natural orbitals: systematic basis set truncation for coupled-cluster theory. *Collect. Czech. Chem. Commun.*, **70**, 837–850.
52. Taube, A.G. and Bartlett, R.J. (2008) Frozen natural orbital coupled-cluster theory: forces and application to decomposition of nitroethane. *J. Chem. Phys.*, **128**, 164101.
53. Landau, A., Khistyaev, K., Dolgikh, S. and Krylov, A.I. (2010) Frozen natural orbitals for ionized states within equation-of-motion coupled-cluster formalism. *J. Chem. Phys.*, **132**, 014109.
54. DePrince, A.E. and Sherrill, C.D. (2013) Accurate noncovalent interaction energies using truncated basis sets based on frozen natural orbitals. *J. Chem. Theory Comput.*, **9**, 293–299.

14

Perturbative Coupled-Cluster Methods on Graphics Processing Units: Single- and Multi-Reference Formulations

Wenjing Ma[1], *Kiran Bhaskaran-Nair*[2], *Oreste Villa*[3], *Edoardo Aprà*[2], *Antonino Tumeo*[4], *Sriram Krishnamoorthy*[4] *and Karol Kowalski*[2]

[1]*Institute of Software, Chinese Academy of Sciences, Beijing, China*
[2]*William R. Wiley Environmental Molecular Sciences Laboratory, Battelle, Pacific Northwest National Laboratory, Richland, WA, USA*
[3]*Nvidia, Santa Clara, CA, USA*
[4]*Computational Sciences and Mathematics Division, Pacific Northwest National Laboratory, Richland, WA, USA*

In this chapter, we discuss the implementation of perturbative coupled-cluster methods on graphics processing units and their implementation within the NWChem quantum chemistry software package. Both single and multi-reference formulations are discussed with the background theory provided for both. This is followed by an overview of the NWChem software architecture and how coupled-cluster methods are handled on GPUs within the global arrays framework that the NWChem software uses. We talk about specific optimizations used to improve performance of the GPU implementation as well as the specifics of the hybrid CPU–GPU (central processing unit–graphics processing unit approach that we employ. Finally, we show performance for CCSD(T) and MRCCSD(T) on GPU cluster and HPC hardware.

Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

14.1 Introduction

Recent progress in computational algorithms allows high-order quantum chemistry methods to simulate realistic molecular systems and processes. A growing body of evidence indicates that the proper inclusion of electron correlation effects is necessary for an accurate description of the mechanisms underlying chemical reactivity, molecular properties, and interactions of light with matter. The availability of reliable methods for benchmarking medium-size systems provides an opportunity to propagate high-level accuracy across spatial scales through proper calibration of low-order methodologies and through coupling of high-accuracy methods with multiscale approaches. However, realistic systems still pose a significant challenge for higher order approaches due to the steep computational cost. For canonical many-body formulations, where no simplifications are made regarding the assumed form of the inter-electron interactions, the utilization of parallel computer architectures is indispensable in addressing their polynomial (in system-size) numerical scaling. Methodologies that combine high numerical complexity with appropriate data granularity are the best candidates for efficient utilization of modern large-scale computer architectures that can comprise hundreds of thousands of computational cores and complex heterogeneity in the form of accelerators. In this context, the steep numerical scaling of the coupled-cluster (CC) methods [1] has been alleviated through efficient parallel implementations (see Ref. [2] and references therein). Moreover, several implementations of the CC methods are capable of utilizing multiple levels of parallelism, which are related to the algebraic structure of the underlying equations describing correlation effects between electrons. An excellent example is provided by the multi-reference coupled-cluster (MRCC) methods, where equations corresponding to various references spanning the model space can be calculated using separate processor groups (PGs). In recent work we have referred to this coarse-grain parallelism as reference-level parallelism (RLP) [3]. Several numerical studies have demonstrated that RLP enables a significant increase in the size of systems that are tractable by MRCC methods.

The emergence of heterogeneous computing systems has had a tremendous impact on the landscape of high-performance scientific computing. In particular, the rise of GPUs for general computation, as discussed in Chapter 2, supports substantially higher computational intensity within lower cost and power budgets. This has led to significant interest in the porting and optimization of scientific applications for GPUs with numerous examples of successful development of GPU-based software in computational chemistry published in the literature [4–33] and discussed in the other chapters of this book. While the previous chapter dealt with iterative CC methods, the focus of this chapter is specifically on the optimization of noniterative CC implementations for GPUs.

As discussed in Chapter 2, the advent of the CUDA as well as OpenCL programming models and OpenACC directives has greatly simplified the design of applications on GPUs. These programming models expose GPU resources as a collection of thread blocks rather than as an image-processing pipeline. These thread blocks are then programmed using a C-like language. While this simplifies the initial design of an application targeting GPUs, maximal utilization of GPU resources continues to require significant optimization effort.

In this chapter, we discuss approaches to efficiently map CC methods to GPUs. This includes effective parallelization of the tensor expressions, mapping them to the thread blocks, effectively utilizing the memory shared among the threads, maximizing data reuse, and effectively scheduling the data transfer between the CPUs and GPUs. The optimizations are specifically tailored to exploit the characteristics of the tensor contraction expression and the architectural characteristics of modern GPUs. To effectively utilize all compute cores in a node, both CPU and GPU, we employ a hybrid scheduling algorithm that dynamically load-balances the parallel units of work between the CPU and the GPU.

14.2 Overview of Electronic Structure Methods

In this chapter, we will give a short overview of two basic CC formalisms: single-reference coupled-cluster (SRCC) and MRCC methods. We will also stress the connections between the algebraic forms of the equations and parallelization strategies.

14.2.1 Single-Reference Coupled-Cluster Formalisms

The CC theory [1, 34–37] is predicated on the assumption that there exists a judicious choice of a single Slater determinant $|\Phi\rangle$, referred to as a reference function, which is capable of providing a zeroth-order description of the ground-state electronic state described by the wave function $|\Psi\rangle$. Usually, for closed-shell systems these reference wave functions are chosen as Hartree–Fock (HF) determinants, although other choices have been discussed in the literature. The existence of a reference function in most cases implies that the many-body perturbation theory (MBPT) expansion based on the Møller–Plesset partitioning of the electronic Hamiltonian H is convergent. Therefore, as a simple consequence of the linked cluster theorem (LCT) [38], energy and the corresponding wave function can be represented in the form of connected and linked diagrams, respectively. The LCT also forms a foundation for the CC representation of the wave function in the form of the exponential ansatz:

$$|\Psi\rangle = e^T|\Phi\rangle, \tag{14.1}$$

where the cluster operator T is represented by connected diagrams only. A standard way of introducing "working" CC equations for the cluster operator is to introduce Eq. (14.1) into the Schrödinger equation, that is,

$$He^T|\Phi\rangle = Ee^T|\Phi\rangle, \tag{14.2}$$

and premultiply both sides of Eq. (14.2) by e^{-T}

$$e^{-T}He^T|\Phi\rangle = E|\Phi\rangle. \tag{14.3}$$

Using the Baker–Campbell–Hausdorff lemma

$$e^{-B}Ae^B = A + [A,B] + \frac{1}{2!}[[A,B],B] + \frac{1}{3!}[[[A,B],B],B] + \cdots, \tag{14.4}$$

one can show that Eq. (14.2) can be cast into the following form:

$$(He^T)_C|\Phi\rangle = E|\Phi\rangle, \tag{14.5}$$

where the subscript "C" designates connected diagrams connected diagrams of a given operator expression. By projecting Eq. (14.5) onto all possible excited configurations $|\Phi^{a_1\cdots a_n}_{i_1\cdots i_n}\rangle$ with respect to the reference determinant $|\Phi\rangle$, one can decouple the equations for the cluster amplitudes:

$$\langle\Phi^{a_1\cdots a_n}_{i_1\cdots i_n}|(He^T)_C|\Phi\rangle = 0 \tag{14.6}$$

from the equation for the energy, obtained by projecting Eq. (14.5) onto the reference function

$$E = \langle\Phi|(He^T)_C|\Phi\rangle. \tag{14.7}$$

The above equations have become the standard equations for determining cluster amplitudes and corresponding energies. One should notice that, first, the nonlinear equations for the T operator need to be solved iteratively before the energy can be calculated using the known cluster amplitudes.

In practical applications, the cluster operator is approximated by a many-body expansion truncated at a certain excitation level m_A ($m_A \ll N$, where N designates the number of correlated electrons in the system):

$$T = \sum_{n=1}^{m_A} T_n, \tag{14.8}$$

where T_n is a part of the cluster operator T, which produces n-tuple excitations when acting on the reference function. The structure of such approximations leads to a well-known "accuracy" hierarchy of the CC methods corresponding to the inclusion of higher rank excitations:

$$\text{CCD} < \text{CCSD} < \text{CCSDT} < \text{CCSDTQ} < \cdots < \text{FullCC}(\equiv \text{FCI}), \tag{14.9}$$

where CCD [1], CCSD [37], CCSDT [39–41], and CCSDTQ [42, 43] approaches are defined by $T \simeq T_2$, $T \simeq T_1 + T_2$, $T \simeq T_1 + T_2 + T_3$, and $T \simeq T_1 + T_2 + T_3 + T_4$, respectively. Although it has been shown that inclusion of higher order clusters provides more accurate estimates of the energy, this procedure quickly becomes numerically infeasible due to the cost of inclusion of higher rank clusters. The numerical complexity of CC approximations grows rapidly with the rank of cluster operators. For example, while CCD and CCSD approaches are characterized by $n_o^2 n_u^4$ numerical scaling (n_o and n_u refer to the number of occupied and unoccupied orbitals), for the CCSDT and CCSDTQ methods this scaling amounts to $n_o^3 n_u^5$ and $n_o^4 n_u^6$, respectively. Even though the cost of the CCSD methods seems to be relatively low compared to the numerical scaling of the CCSDTQ method, one should realize that performing calculations for a water pentamer will be $5^6 = 15,625$ times more expensive compared to the calculations for a water monomer (assuming that the same basis set is used). The growth in the numerical cost looks even more intimidating for higher order methods.

Unfortunately, in many calculations, especially in the area of thermochemistry, it quickly became clear that the accuracies of CCSDT are needed. An efficient way of addressing this issue is to consider the links between SRCC theory and MBPT expansion. MBPT techniques enable one to determine a hierarchical structure of particular correlation effects. For example, the Møller–Plesset perturbation theory [44] establishes the following structure of the cluster operator:

$$T_2 = T_2^{(1)} + T_2^{(2)} + T_2^{(3)} + \cdots \tag{14.10}$$

$$T_1 = T_1^{(2)} + T_1^{(3)} + T_1^{(4)} + \cdots \tag{14.11}$$

$$T_3 = T_3^{(2)} + T_3^{(3)} + T_3^{(4)} + \cdots \tag{14.12}$$

$$T_4 = T_4^{(3)} + T_4^{(4)} + T_4^{(5)} + \cdots \tag{14.13}$$

$$\cdots,$$

where the order of the perturbation expansion is denoted as the superscript. One can show that the triply excited clusters in the second order of perturbation theory can be expressed in terms of first-order doubly excited cluster amplitudes, for example,

$$T_3^{(2)}|\Phi\rangle = R_3^{(0)} V_N T_2^{(1)}|\Phi\rangle, \tag{14.14}$$

where V_N is the two-body part of the Hamiltonian in normal ordered form $H_N = H - \langle\Phi|H|\Phi\rangle = F_N + V_N$, and $R_3^{(0)}$ is a three-body resolvent operator

$$R_3^{(0)} = \sum_{i<j<k;a<b<c} \frac{|\Phi_{ijk}^{abc}\rangle\langle\Phi_{ijk}^{abc}|}{\epsilon_i + \epsilon_j + \epsilon_k - \epsilon_a - \epsilon_b - \epsilon_c}, \tag{14.15}$$

where ϵ's refer to the HF orbital energies. Using similar arguments the authors of Ref. [45] have introduced the corrections to the energy obtained in the CCSD calculations (E^{CCSD}):

$$E^{\mathrm{CCSD(T)}} = E^{\mathrm{CCSD}} + \langle\Phi|T_2^+ V_N R_3^{(0)} V T_2|\Phi\rangle + \langle\Phi|T_1^+ V_N R_3^{(0)} V T_2|\Phi\rangle \,, \tag{14.16}$$

which defines the so-called CCSD(T) approach. An important feature of the CCSD(T) approach is the fact that it combines elements of fourth and fifth order of the standard MBPT expansion containing triply excited intermediate states. The CCSD(T) approach can also be viewed as an extension of the CCSD[T] method [46]. Currently, the CCSD(T) method is the most frequently employed CC approach especially in studies of spectroscopic properties, geometry optimization, and chemical reactions.

The indisputable success of the CCSD(T) method in describing nondegenerate electronic states has sparked an interest toward extension of the perturbative methods accounting for the effect of triples toward more challenging cases where the wave function is no longer dominated by a single determinant. This is a typical situation encountered in bond-breaking/forming processes.

Significant progress in addressing these challenges has been achieved by developing perturbative techniques based on the partitioning of similarity-transformed Hamiltonians [47–49]. These ideas have spawned into several formulations of noniterative ground-state corrections [50–58]. Another class of methods tackling the problem of bond-breaking processes is anchored in the method of moments (MMCC) formalism [59], where the expansion for the exact energy is expressed in terms of nonvanishing moments corresponding to the approximate CC approaches. The algebraic structure of these expansions makes them a very efficient tool in designing noniterative approaches. Especially effective in recovering the correlation effects is the recently introduced variant of the MMCC method based on the use of left eigenvectors of the similarity-transformed Hamiltonian (see the CR-CC(2,3) and CR-CC(2,4) approaches introduced by Piecuch *et al.* [60, 61]).

Recently, a new formalism, which combines a new form of the moment expansion with regularization of the cluster operator, has emerged. In the so-called generating functional (GF) moment expansion [62] (or GF expansion for short), the exact energy (E) is expressed in terms of approximate CC energy ($E^{(A)}$), the moments of the CC equations ($\bar{M}_J$, J labels here excited configurations), and derivatives of the GF (W), which corresponds to a connected part of the overlap between exact and auxiliary wave functions in the exponential parametrization, defined by Σ and S cluster operators, respectively.

$$E = E^{(A)} + \sum_{J;J\neq 0} \bar{M}_J^{(A)} \left[\frac{\partial}{\partial S_J} W(\Sigma, S)\right] |_{S^{(A)}=T^{(A)};S^{(R)}=0,} \tag{14.17}$$

where

$$W(\Sigma, S) = \ln\langle\Phi|(e^{\Sigma^+} e^S)_C|\Phi\rangle. \tag{14.18}$$

The validity of the expansion (14.17) is limited only to regions where the correlation effects are characterized by rather small values of corresponding cluster amplitudes. In order to extend the GF expansion to strongly interacting systems, the regularization of cluster amplitudes is necessary. These factors have been included in the design of the regularized version of the CCSD(T) (Reg-CCSD(T)) [63], which is defined by a formula analogous to Eq. (14.16):

$$E^{\mathrm{reg-CCSD(T)}} = E^{\mathrm{CCSD}} + \langle\Phi|(\Sigma_{\mathrm{reg},2}^+ V_N) R_3^{(0)}(\omega^2) V T_2|\Phi\rangle + \langle\Phi|\Sigma_{\mathrm{reg},1}^+ V_N R_3^{(0)}(\omega^2) V T_2|\Phi\rangle, \tag{14.19}$$

where $\Sigma_{\mathrm{reg},1}$ and $\Sigma_{\mathrm{reg},2}$ are regularized cluster operators, ω^2 is a regularization parameter, and $R_3^{(0)}(\omega^2)$ is a ω^2-dependent three-body resolvent

$$R_3^{(0)}(\omega^2) = \sum_{i<j<k;a<b<c} \frac{|\Phi_{ijk}^{abc}\rangle\langle\Phi_{ijk}^{abc}|}{\epsilon_i + \epsilon_j + \epsilon_k - \epsilon_a - \epsilon_b - \epsilon_c - 3\omega^2} \,. \tag{14.20}$$

Similar regularization techniques based on the Tikhonov regularization have been used to alleviate problems encountered by the linear SRCC formulations in the quasi-degenerate regime [64]. For $\omega^2 = 0$, the Reg-CCSD(T) approach is identical to the original CCSD(T) formulation. It has been demonstrated that the regularization algorithm, Eq. (14.19), leads to significant improvements of the CCSD(T) results in the geometry regions corresponding to stretched nuclear geometries [63]. It should also be mentioned that the Reg-CCSD(T) and CCSD(T) approaches are characterized by the same numerical overhead proportional to $n_o^3 n_u^4$. As mentioned earlier, the expressions for the Reg-CCSD(T) and CCSD(T) methods have the same algebraic structure. Therefore, the GPU implementation for the noniterative Reg-CCSD(T) methods also provides the GPU implementation for the CCSD(T) approach ($\omega^2 = 0$).

14.2.2 Multi-Reference Coupled-Cluster Formulations

Many aspects of computational chemistry require methodologies capable of describing a delicate balance between static and dynamic correlation effects for systems and processes belonging to the so-called multi-reference chemistry. The common feature of systems falling into this class of problems is the fact that the corresponding wave functions, in contrast to the closed-shell systems, cannot be described by a single Slater determinant. This situation commonly occurs in bond-breaking processes, poly-radical species, transition-metal compounds, low-spin open-shell states, and reactions involving potential energy surface crossing. To describe these problems, one has to resort to multi-reference methods such as complete active space self-consistent field formalisms, various variants of multi-reference many-body perturbation theory (MRMBPT) methods [65–73] (such as the ubiquitous CASPT2 approach [74]), and multi-reference configuration interaction formulations (MRCI) [75], MRCC Methods [76–99], and canonical transformation theory [100].

The dawn of peta-scale computer architectures offers a unique opportunity to validate and apply accurate yet numerically expensive multi-reference theories to large molecular systems. The MRCC method is among the methodologies that can take advantage of this fact. The MRCC formalisms extend the applicability of the SRCC methods to quasi-degenerate situations by replacing the notion of a reference function $|\Phi\rangle$ used in the SRCC theories by the concept of the model space ($\mathcal{M}_0$) spanned (ls) by the most important Slater determinants $|\Phi_\mu\rangle$ ($\mu = 1, \cdots, M$)

$$\mathcal{M}_0 = ls\{\ |\Phi_\mu\rangle\ \}_{\mu=1}^{M} \tag{14.21}$$

required to describe a certain subset of electronic states. In a natural way, the MRCC formalisms lead to partitioning of correlation effects into static (associated with the many-body effects within the model space $\mathcal{M}_0$) and dynamic (associated with the many-body effects within the orthogonal complement of the model space $\mathcal{M}_0^{\perp}$) effects. These methods thus provide a way to properly treat correlation effects for electronic states characterized by strong correlation effects.

In this chapter, we will focus on the state-specific Hilbert space (HS) formulation of the MRCC theory, where the electronic wave function $|\Psi\rangle$ is expressed by the Jeziorski–Monkhorst Ansatz [87]

$$|\Psi\rangle = \sum_{\mu=1}^{M} c_\mu e^{T^{(\mu)}} |\Phi_\mu\rangle, \tag{14.22}$$

where the $T^{(\mu)}$ are the reference-specific cluster operators. For the complete model space (CMS), which is defined by determinants obtained by all possible distributions of the active electrons among the active spin orbitals, the intermediate normalization condition [87] requires that the cluster operators do not produce any excitations within the model space, that is,

$$\langle \Phi_\nu | T^{(\mu)} | \Phi_\mu \rangle = 0 \ \forall_{\mu,\nu}. \tag{14.23}$$

In the most rudimentary MRCC approximation with singles and doubles (MRCCSD), the cluster operators are represented as a sum of singly ($T_1^{(\mu)}$) and doubly ($T_2^{(\mu)}$) excited clusters:

$$T^{(\mu)} \simeq T_1^{(\mu)} + T_2^{(\mu)}, \tag{14.24}$$

where each component $T_i^{(\mu)}$ is defined by the cluster amplitudes $t_{a(\mu)\cdots}^{i(\mu)\cdots}(\mu)$

$$T_1^{(\mu)} = \sum_{i(\mu),a(\mu)} t_{a(\mu)}^{i(\mu)}(\mu) X_{a(\mu)}^{+} X_{i(\mu)}, \tag{14.25}$$

$$T_2^{(\mu)} = \sum_{i(\mu)<j(\mu),a(\mu)<b(\mu)} t_{a(\mu)b(\mu)}^{i(\mu)j(\mu)}(\mu) X_{a(\mu)}^{+} X_{b(\mu)}^{+} X_{j(\mu)} X_{i(\mu)}. \tag{14.26}$$

In the above summations, $i(\mu), j(\mu)$, $(a(\mu), b(\mu))$ indices correspond to occupied (unoccupied) spin orbitals in reference $|\Phi_\mu\rangle$. As a consequence of the intermediate normalization condition, the summations in Eqs. (14.25) and (14.26) exclude the case when all indices correspond to active spin orbitals. In the state-specific formulations, the working equations for the cluster amplitudes (or the sufficiency conditions) are obtained by substituting the Jeziorski–Monkhorst Ansatz (14.22) into the Schrödinger equation

$$H \sum_{\mu}^{M} c_\mu e^{T^{(\mu)}} |\Phi_\mu\rangle = E \sum_{\mu=1}^{M} c_\mu e^{T^{(\mu)}} |\Phi_\mu\rangle. \tag{14.27}$$

Because of the known overcompleteness problem of the state-specific approaches, several types of sufficiency conditions have been discussed in the literature [95–99]. The BW-MRCCSD and Mk-MRCCSD amplitude equations take the following form:

$$(E - H_{\mu\mu}^{\text{eff}}) \langle \Phi_\theta^{(\mu)} | e^{T^{(\mu)}} |\Phi_\mu\rangle - \langle \Phi_\theta^{(\mu)} | H_N(\mu) e^{T^{(\mu)}} |\Phi_\mu\rangle_{C+DC,L} = 0 \;\forall_\mu \;, \quad \text{(BW-MRCCSD)}, \tag{14.28}$$

$$\langle \Phi_\theta^{(\mu)} | (H e^{T^{(\mu)}})_C |\Phi_\mu\rangle c_\mu + \sum_{\nu\neq\mu} \langle \Phi_\theta^{(\mu)} | e^{-T^{(\mu)}} e^{T^{(\nu)}} |\Phi_\mu\rangle H_{\mu\nu}^{\text{eff}} c_\nu = 0 \;\forall_\mu, \quad \text{(Mk-MRCCSD)}, \tag{14.29}$$

where the cluster operators are given by Eq. (14.24), and $\langle \Phi_\theta^{(\mu)}|$ are excited configurations corresponding to the excitations used to define cluster operators $T^{(\mu)}$. The subscript $C + DC, L$ designates all connected diagrams and all linked, but disconnected diagrams. In contrast to the Mk-MRCC approach, the BW-MRCC formalism contains disconnected diagrams. The energies and c_μ coefficients are obtained by diagonalizing the effective Hamiltonian matrix (as eigenvalue and components of corresponding eigenvector), which for CMS is defined by the matrix elements $H_{\nu\mu}^{\text{eff}}$:

$$H_{\nu\mu}^{\text{eff}} = \langle \Phi_\nu | (H e^{T^{(\mu)}})_C |\Phi_\mu\rangle. \tag{14.30}$$

The sufficiency conditions can be cast in the following algebraic form:

$$\mathbf{R}^{(\mu)} = \mathbf{F}^{(\mu)}(T^{(\mu)}) + \mathbf{G}^{(\mu)}(T^{(1)}, \cdots, T^{(\mu)}, \cdots, T^{(M)}) = \mathbf{0} \quad \forall_{\mu=1,\cdots,M}, \tag{14.31}$$

where the $\mathbf{F}^{(\mu)}(T^{(\mu)})$ part represents the direct terms ($\langle \Phi_\theta^{(\mu)} | (H e^{T^{(\mu)}})_C |\Phi_\mu\rangle$) and the $\mathbf{G}^{(\mu)}(T^{(1)}, \cdots, T^{(\mu)}, \cdots, T^{(M)})$ parts represent the coupling terms. While the direct terms depend only on the cluster operator corresponding to a given reference, the coupling term may involve all possible cluster operators. This can be directly seen in the Mk-MRCCSD approach [99], while in the BW-MRCCSD this dependence is implicit through the energy obtained from the diagonalization of the effective Hamiltonian.

Among several ways of correcting MRCCSD energies (for details see Ref. [101] and references therein), the simplest approach is based on adding diagrams accounting for the effect of triples to the diagonal elements of the MRCCSD effective Hamiltonian matrix ($\mathbf{H}^{\mathrm{eff}}(CCSD)$) in a way analogous to that of the single reference CCSD(T) approach, that is,

$$H^{\mathrm{eff}}_{\nu\mu}(\mathrm{T}) = H^{\mathrm{eff}}_{\nu\mu}(\mathrm{CCSD}) + \delta_{\nu\mu}\delta_{\mu\mu}(T_3^{(\mu)}), \tag{14.32}$$

$$\delta_{\mu\mu}(T_3^{(\mu)}) = E^{[4]}_{\mathrm{T}}(\mu) + E^{[5]}_{\mathrm{ST}}(\mu) + E^{[4]}_{\mathrm{ST}}(\mu), \tag{14.33}$$

where $\delta_{\nu\mu}$ represents Kronecker δ function. The $E^{[4]}_{\mathrm{T}}(\mu)$, $E^{[5]}_{\mathrm{ST}}(\mu)$, and $E^{[4]}_{\mathrm{ST}}(\mu)$ terms represent fourth- and fifth-order contributions, given by the expressions

$$E^{[4]}_{\mathrm{T}}(\mu) = \frac{1}{36}\sum_{abcijk}\langle(\Phi_\mu)^{abc}_{ijk}|V_N(\mu)T_2^{(\mu)}|\Phi_\mu\rangle_C t^{abc}_{ijk}(\mu), \tag{14.34}$$

$$E^{[5]}_{\mathrm{ST}}(\mu) = \sum_{ai} s^a_i(\mu)t^a_i(\mu), \tag{14.35}$$

$$E^{[4]}_{\mathrm{ST}}(\mu) = \frac{1}{4}\sum_{abcijk} f_{kc}(\mu)t^{ab}_{ij}(\mu)t^{abc}_{ijk}(\mu), \tag{14.36}$$

where the $s^a_i(\mu)$ intermediate is defined as

$$s^a_i(\mu) = \frac{1}{4}\sum_{bcjk}\langle bc\|jk\rangle t^{abc}_{ijk}(\mu). \tag{14.37}$$

The form of these corrections is the same for both the BW-MRCCSD and Mk-MRCCSD effective Hamiltonians. Although several forms of the triply excited cluster amplitudes $t^{abc}_{ijk}(\mu)$ have been Discussed [102–106], we will employ the simplest choice defined by the formula

$$t^{abc}_{ijk}(\mu) = \frac{\langle(\Phi_\mu)^{abc}_{ijk}|V_N(\mu)T_2^{(\mu)}|\Phi_\mu\rangle_C}{D^{abc}_{ijk}(\mu)}, \tag{14.38}$$

which is analogous to the form of the T_3 operator utilized by the CCSD(T) method. In all our implementations, the $\mathbf{H}^{\mathrm{eff}}(CCSD)$ operator includes up to two-body terms. The total cost of this procedure (further referred to as the MRCCSD(T) approach) scales as $M \times N^7$, which poses a significant computational challenge.

In the following sections, we provide details of the GPU implementation of the SRCC and MRCC noniterative methods in the NWChem suite of codes [107].

14.3 NWChem Software Architecture

Two main concepts influenced the original design of NWChem conceived in 1993:

- scalability to a large number of processing elements on massively parallel computers;
- modular structure forming the foundation to implement new theoretical methods.

These two defining features distinguish the NWChem development, which is focused on massively parallel computers, from more traditional development techniques widely used in several successful computational chemistry packages.

The modular structure of the code architecture is achieved by using an object-oriented approach within the realm of the Fortran programming language. For a more detailed description of this topic,

we refer the reader to the publications that describe in detail the various components of the NWChem architecture [107, 108].

In order to achieve parallel scalability, the NWChem software development adopts, for the bulk of its communication, the Global Arrays (GAs) toolkit [109]. The GA toolkit is a library that was designed for parallelizing codes whose main quantities are large and dense arrays. GA forms an abstraction layer for the scientific programmer by distributing the arrays among the memory of the processing elements of a distributed-memory parallel computer and providing a series of operations for easily manipulating the elements of the arrays. While a more traditional message-passing approach would require synchronization of sender and receiver to perform tasks such as array transformation, the NWChem code uses one-sided GA operations for this purpose. GA operations can be classified into two categories: collective and local. Collective calls require all processes to participate, while local operations may be called by each process independently. Fetching (`ga_get`) or updating (`ga_put`) are the most common local operations. Commonly used linear algebra operations belong to the collective category (e.g., matrix multiply, eigensolvers, etc.). GA provides language bindings for Fortran and C/C++. The library is meant to be compatible with MPI and uses MPI itself for some of its functionality (e.g., process creation, some collective calls). The efficiency of GA on a given computer architecture relies heavily on the aggregate remote memory copy interface (ARMCI) library. ARMCI fulfills the role of the GA primary communications layer.

14.4 GPU Implementation

Evaluation of the CCSD(T) and MRCCSD(T) corrections is dominated by on-the-fly calculation of the $\langle \Phi^{abc}_{ijk} | V_N T_2 | \Phi \rangle$ (CCSD(T)) and $\langle (\Phi_\mu)^{abc}_{ijk} | V_N(\mu) T_2^{(\mu)} \Phi_\mu \rangle$ (MRCCSD(T)) projections, which assume identical algebraic forms. In both cases the related computational cost is proportional to N^7, which poses a significant challenge in calculations of large molecular systems. The details of the CCSD(T) and MRCCSD(T) GPU implementations are discussed in detail in Refs [26, 27]. Here we describe the general tenets of these developments. The N^7 scaling term in the CCSD(T) approach stems from the following terms:

$$\begin{aligned}\langle \Phi^{abc}_{ijk} | V_N T_2 | \Phi \rangle = {} & v^{ij}_{ma} t^{mk}_{bc} - v^{ij}_{mb} t^{mk}_{ac} + v^{ij}_{mc} t^{mk}_{ab} - v^{ik}_{ma} t^{mj}_{bc} + v^{ik}_{mb} t^{mj}_{ac} - v^{ik}_{mc} t^{mj}_{ab} \\ & + v^{jk}_{ma} t^{mi}_{bc} - v^{jk}_{mb} t^{mi}_{ac} + v^{jk}_{mc} t^{mi}_{ab} - v^{ei}_{ab} t^{jk}_{ec} + v^{ei}_{ac} t^{jk}_{eb} - v^{ei}_{bc} t^{jk}_{ea} \\ & + v^{ej}_{ab} t^{ik}_{ec} - v^{ej}_{ac} t^{ik}_{eb} + v^{ej}_{bc} t^{ik}_{ea} - v^{ek}_{ab} t^{ij}_{ec} + v^{ek}_{ac} t^{ij}_{eb} - v^{ek}_{bc} t^{ij}_{ea}, \\ & (i < j < k, a < b < c), \end{aligned} \tag{14.39}$$

where t^{ab}_{ij} and v^{pq}_{rs} represent doubly excited cluster amplitudes and antisymmetric two-electron integrals. In order to provide granularity for the parallel tensor contraction engine (TCE) [110] generated codes, the whole spin-orbital domain is partitioned into smaller pieces called *tiles*, which contain several spin-orbitals of the same spatial and spin symmetry. The maximum number of elements in the tile is often referred to as the `tilesize`. This partitioning induces partitioning or block-structure of all tensors used in the CC calculations. In the parallel implementation of the (T)-part, each core takes care of a different subset of projections defined by tiles $[i]$, $[j]$, $[k]$, $[a]$, $[b]$, $[c]$, that is, each core generates on the fly the set of $\langle \Phi^{abc}_{ijk} | V_N T_2 | \Phi \rangle$ projections with $i \in [i]$, $j \in [j]$, $k \in [k]$, $a \in [a]$, $b \in [b]$, $c \in [c]$. These projections are stored on the six-dimensional matrices $P3$ ($\langle \Phi^{abc}_{ijk} | V_N T_2 | \Phi \rangle$). For example, $P3$ is defined as the following matrix:

$$P3 \equiv P3(\dim[a], \dim[b], \dim[c], \dim[i], \dim[j], \dim[k]) \,, \tag{14.40}$$

where dim[i], … ,dim[c] are the dimensions of the corresponding tiles. Therefore, the local memory requirement for storing $P3$ matrices is defined by `tilesize`^{6}. If `tilesize` equals 20, this is equivalent to 0.48 GB (in a recently developed algorithm for the (T)-part of TCE, these tensors can be "sliced" along the first two dimensions, which leads to a less intensive use of the local memory, and one can effectively use a larger `tilesize` in the (T) calculations). Since the total floating-point operation (FLOP) count assigned to each core associated with forming $P3$ tensors is proportional to `tilesize`$^6 * n_u$, this type of calculation is ideally suited to take advantage of GPU accelerators. The whole process is split into a number of smaller tasks, where the summation goes over indices from a single tile; for example

$$P3(a,b,c,i,j,k) -= \sum_{e \in [e]} V(a,b,e,i) * T2(j,k,e,c) \ , \\ (i \in [i], j \in [j], k \in [k], a \in [a], b \in [b], c \in [c]), \tag{14.41}$$

where $V(a,b,e,i)$ and $T2(j,k,e,c)$ are two-electron integrals and doubly excited amplitudes tensors. For this elementary task, the FLOP count is equal to `tilesize`^{7}, which for `tilesize`=20 corresponds to 1.2 GB. For this reason, the utilization of the GPU accelerators can lead to considerable speedups in the case of large numerical load, which is created by the use of larger tiles.

Our GPU implementation exploits the Nvidia CUDA [111], which includes both a programming model and a virtual architecture model for the execution of general-purpose computation on GPUs. As discussed in earlier chapters, in CUDA a programmer describes its code in a parallel kernel, organized in groups of threads. Threads are grouped in multidimensional grid (up to three dimensions) thread blocks. Thread blocks are then grouped in a bidimensional grid. A grid of thread blocks corresponds to a kernel. The programmer, through host code, moves the data to the GPU memory space, executes the kernel on the data, and copies back the results. Various evolutions of the CUDA runtime have enabled peer-to-peer data movement from one GPU to another, even across the nodes of a cluster, without the intervention of the host. As discussed extensively in earlier chapters, when developing the CUDA code, a programmer must follow several guidelines to map its kernel on the target GPU architecture and maximize its performance. Failing to do so may result in lower than expected performance.

In this section, we provide a brief refresher overview of the Kepler architecture, which is at the core of the Tesla K20 GPU boards integrated in the Titan supercomputer at ORNL National Laboratory. We then present the baseline GPU kernel implementation for the tensor contractions and the set of architecture-specific optimizations. The CUDA code and the related optimizations for the tensor contractions are automatically generated through a domain-specific language (DSL) approach, which allows easy reuse of the same acceleration approaches for the various types of contractions that can be found in the CCSD(T) and MRCCSD(T) methods.

Since heterogeneous clusters, such as the Titan supercomputer, include both GPUs and CPUs, there is an opportunity to exploit one or the other or, ideally in seeking maximum performance, to attempt to utilize both simultaneously. Our implementation objective is to exploit all the available processing elements in order to maximize achievable performance even if this complicates the implementation. In the last part of this section, we explain how we implemented a dynamic load balancer that enables heterogeneous processing at the level of the whole cluster by exploiting the GA toolkit [109].

14.4.1 Kepler Architecture

For the GPU implementation, we targeted the Kepler architecture [112], which is the basis of the Tesla K20X [113] boards integrated in Titan (and also the basis of the K20, K40, and K80 model of Nvidia GPUs). The Tesla K20X is based on the GK110 processor and architecturally is a significant departure

from the previous generation designs (Fermi and Tesla), and mainly focuses on providing higher performance per watt with respect to its predecessors. The basic building blocks of the processor are the streaming multiprocessors, named SMXes. In the K20X, an SMX includes 192 single-precision and 64 double-precision arithmetic logic units (also called CUDA cores), 32 special-function units (SFUs, units that perform complex operations such as transcendental and trigonometric operations), and 32 load/store units for memory operations. The arithmetic is fully IEEE 754/2008 compliant. A K20X SMX includes 64 kB of on-chip memory, which can be configured as 16 kB shared memory (i.e., directly addressable memory) with 48 kB Level 1 cache, 32 kB shared memory with 32 kB Level 1 cache, or as 48 kB shared memory with 16 kB Level 1 cache. In addition, there is a 48 kB cache for read-only data. This is an enlarged and more flexible version of the texture cache exposed in previous generation architectures, now opened to all the load operations from the SM. It has the added benefit to support full-speed unaligned memory access patterns. The K20X has a Level 2 cache of 1536 kB. The SMX schedules groups of 32 parallel threads, called warps. The 32 threads execute the same instruction, following a single-instruction, multiple-threads (SIMT) approach. An SMX features four warp schedulers, which allow issuing and executing four warps in parallel. Each scheduler provides two instruction dispatch units: this allows it to execute, during each cycle, two independent instructions of a warp in parallel. Different from previous architectures, the scheduler can simultaneously issue 32- and 64-bit instructions. If threads in a warp incur in a branch, and some of the threads take different directions, they generate divergence, slowing down the execution, because the warp scheduler fetches the same instruction for the whole warp. So, in a branch, the warp scheduler must fetch the instructions for all the different directions even if they are executed only by some of the threads. An SMX includes a total of 65,536 registers and can keep up to 2048 threads active. The new instruction set architecture (ISA) of Kepler enables each thread to use up to 255 registers; obviously, a higher number of registers used per thread means that fewer threads can be kept simultaneously active. The ISA also implements native shuffle and efficient atomic instructions. The HPC Tesla versions of Kepler architecture GPUs enable several interesting features in the CUDA programming model. Dynamic parallelism enables the GPU to generate new work (kernels), without involving the CPU. Hyper-Q increases the number of work queues between the host and the work distributor logic in the GPU, allowing multiple CUDA streams, multiple processes on the host (e.g., different MPI processes), and multiple threads in a process to issue work simultaneously. A full GPU chip includes multiple SMXes. For the Tesla K20X, there are 14 SMXes [113]. The SMXes are connected to the on-board memory through six 64-bit-wide memory controllers, providing a total bus width of 384 bits. As discussed in Chapter 2, in CUDA terminology the on-board memory is called *global memory* because it is shared by all the SMXes. The Tesla K20X GPU has a clock of 732 MHz (the clock domain is the same for the whole chip) and connects to 6 GB of GDDR5 memory with a data rate of 5.2 GHz, for a peak theoretical bandwidth of 250 GB/second.

Although Kepler has more relaxed constraints than previous architectures, there still are some rules that a programmer should follow to maximize memory bandwidth utilization. Kepler requires that memory accesses of threads from the same warps reside in the same 128-byte chunk, which corresponds to the size of a cache line. This requirement is also termed memory access coalescing, because the memory operations need to be aggregated together with a specific schema. The reason is that the data are brought in the cache anyway after the first access. In a departure from previous architectures, memory accesses of logically consecutive threads do not need to access consecutive memory locations in the same order, and can interleave. The shared memory, instead, has 32 banks of 64 bits each, and the only requirement for not generating bank conflicts is that the accesses of the threads of the warps are inside the same 64-bit word aligned segment, for both 32- and 64-bit accesses. In any case, bank conflicts have impacts orders of magnitude smaller on the performance with respect to earlier architectures.

14.4.2 Baseline Implementation

Our GPU implementation is based on a DSL code generation approach. Our approach provides a new TCE for NWChem that enables automatic generation of CUDA code starting from the high-level expression of the tensor contractions in mathematical form. The engine performs the analysis and generates code integrating the optimizations that allow better exploitation of the GPU architecture. This approach makes the optimizations more general and reusable for other methods that exploit (generalized) tensor contractions beside the CC method.

Our baseline approach for implementing tensor contractions on GPUs, which maps computations within a thread block, is similar to the approaches used for matrix–matrix multiplication. However, it also includes optimizations for memory management and common sub-expression refactoring.

The whole application invokes the same sequential tensor contractions a number of times. This requires executing each time expensive memory allocation and deallocation in CUDA. To reduce the interactions with the CUDA runtime and increase the performance, we developed a simple but effective memory manager, which reuses previously allocated, but not currently used, memory locations for newly issued requests. The kernel that executes the tensor contractions requires the dimensions of the tensors and the pointers to the buffers in GPU memory as inputs. The host passes all the buffer with the data of a tensor as a linear array, thus it is necessary to compute the strides to access the different dimensions of the tensors. Because the computation of the strides is redundant and constantly reused by all the threads to compute the offsets of each element accessed, they are computed in the host and passed as arguments to the kernel.

All the thread blocks and threads share the same values of the kernel function arguments. Threads differentiate their work by exploiting their indices (locations in the thread block and in the grid of thread blocks). As previously explained, however, the number of dimensions for thread blocks and grids is 3. While this is sufficient for standard matrix multiplications, it is too limited with respect to the number of dimensions of tensors. Thus, threads need to identify values for the dimension by exploiting modulo and division operations on their own indices. To minimize the number of operations, the encoding of tensor dimensions is split between the number of grid dimensions available. Nevertheless, the size of the problem can significantly vary and is unknown at compile time. Because the number of tensor contractions is very high, tailoring thread block sizes for each expression would be unmanageable. For these reasons, the size of the thread blocks is fixed at 16 along all dimensions.

The computation inside a thread block follows the same principle of the matrix multiplication, but with additional dimensions. The kernel chooses a contracted index, and one dimension each from two tensors to perform a matrix multiplication at the thread-block level. The threads in each thread block cooperate in moving data between the GPU memory and the shared memory. The overall kernel implementation exploits the full dimensionality of the thread block to reduce the index computation overhead. Each thread computes an element of the output array, thereby maximizing reuse. Each thread also moves, at most, only a single element to the shared memory, minimizing the data movement costs. For each of the input tensors, the kernel tiles the dimension that has the fastest varying index that is not a common index. This enables different threads to access adjacent elements of the input arrays, thus allowing coalescing of single-element memory operations.

14.4.3 Kernel Optimizations

On top of this baseline implementation, which exploits the basic approach of the CUDA programming model, we implemented several optimizations to better adapt the generated kernel to Kepler generation GPU architectures. These optimizations are enabled by the automatic generation of the code from the high- level DSL.

The first optimization involves the ordering of the indices of the tensor contractions. When indices of tensor contractions in the original high-level representation always occur in the same order, they can be replaced by a single index whose size is the product of the previous indices. We call this optimization *index combining*. Performing this optimization as early as possible is very convenient, because it improves thread-block utilization and reduces the index computation overhead in the kernel. Its benefits are obviously dependent on the actual tensor contractions, but in general they appear to be very useful for the CC calculations.

The second optimization is related to the fact that the baseline approach only exploits tiling in one dimension for each input array. When the tiled dimension matches the thread block size, or if the tiled dimension is sufficiently large, they approach optimum performance. However, dimensions of tensors involved in CC calculations vary significantly due to different types of sparsity and symmetry. In many cases, the tiled dimensions are small, and do not match the thread-block configuration (e.g., dimension size of 17 with thread block size of 16), significantly reducing the thread-block and warp utilization. For these reasons, our code generator is able to perform an optimization named *dimension flattening*. In the baseline implementation, all the threads in a thread block contribute to elements of the output array that differ only for two indices, one for each input array. With the dimension-flattened code, the indices of the output array are grouped into those from each of the input array. Each group is flattened into a single linear dimension, and tiled according to the thread-block dimensions. Each thread contributes to a distinct value of the whole dimension-flattened array, possibly resulting in threads from the same thread blocks operating on elements of the output array that differ in more than two indices. Figure 14.1 shows this optimization, highlighting the

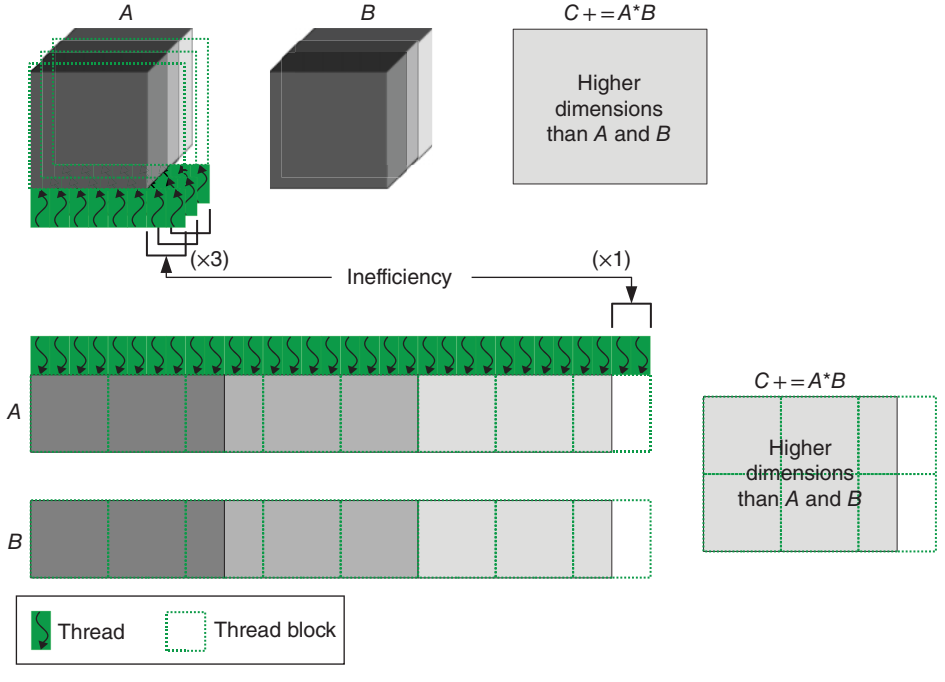


Figure 14.1 *Dimension-flattening optimization. The solid lines correspond to different two-dimensional regions. The dotted lines correspond to the mapping of the data blocks to the thread blocks. Tensors A and B are flattened into two-dimensional arrays, increasing utilization of thread blocks. Each thread block not necessarily works only on values from a single dimension of the original tensor*

reduction of the inefficiencies. Index-combining is oblivious to subsequent transformation. However, dimension- flattening incorporates within a single operation both tiling and an implicit index permutation operation. Thus, the kernel needs to re-create the original indices during its execution. Conversely, the kernel does not need to reproduce the indices coalesced through index-combining. We ensure that data movement in the shared memory is the same as the baseline version, except for the conditionals, through careful encoding of the indices. Each thread still loads a single element of the input arrays. The behavior is obtained by encoding the flattened group indices into the dimensions of the thread-block and grid configurations through the gridDim and blockDim constructs. Decoding is performed by exploiting the multidimensional thread identifiers (threadIdx) and the size (number of threads) for each dimension of the thread block (blockDim). The threads that are on the same dimension of a thread block obtain the same index of the group for that dimension, thus they can share the data movement cost. Dimension-flattening also maintains the higher locality achieved in the baseline implementation by tiling on one of the fastest varying indices. This is obtained by decoding the indices beginning with the fastest varying index, and ensuring that adjacent threads in a thread block process potentially adjacent elements of input arrays, facilitating coalescing of memory accesses.

Another optimization, enabled by the larger register files of the latest GPU architectures, is *register tiling*. As previously explained, each thread computes one element of the output tensor. Register tiling is implemented by modifying the baseline kernel as follows: In the host function that invokes the kernel, the grid size is reduced by a factor of 4 for each dimension. The thread-block size, instead, remains the same. This means that each thread reads 4 times the data from the input arrays, and writes 16 times the data into the output array. The overall result is a reduction in the number of memory accesses of the kernel to the input arrays. At each iteration, four rows and four columns from the two input tensors need to be loaded in the shared memory. This corresponds to a total of 64 rows and 64 columns of output, and requires the calculation of four index sets for the first and four index sets for the second input tensors. Each thread then loads four elements from each corresponding row and four elements from each corresponding column in the shared memory into registers. Each thread accesses each of the four elements with a stride of 16 (block size in one dimension) and then performs the 16 multiplications, accumulating the results into 16 double-precision registers. The results are then accumulated in the output array in the device memory.

The higher reuse of data in shared memory and the exploitation of registers with the register tiling optimization make the accumulation of the results in the output arrays the most constraining part for accesses to the global memory. The index calculation in the baseline implementation favors the input tensors and tries to coalesce as much as possible their accesses. The code optimized with register tiling, instead, also exploits a *modified index calculation order*. The modified order ensures that adjacent threads operate on adjacent sets of output elements, providing coalesced memory accesses.

Since each thread in the code optimized with register tiling computes 16 elements instead of 1, the thread must check the boundaries for each of the 16 writes to the output array. Every time a value is written, the thread must check whether the location it is going to write to, addressed by its own thread index plus the stride, falls inside the boundaries of the destination array. It must do this four times for each of the dimensions of the thread block. This checking leads to a large number of branches. To reduce the overhead, the boundaries are checked in reverse order with respect to the strides (*reversed condition checking*). First, the larger stride is checked, and in such a case all the four offsets are calculated. If not, it progressively reduces the checking to smaller strides, only calculating the required offsets. This makes it possible to perform four checkings (thus causing conditional branches) only in the worst case. In the best case, the kernel executes only one check, because if the first condition is met, subsequent conditional statements are not executed.

14.4.4 Data-Transfer Optimizations

We exploited the CUDA support for asynchronous CPU–GPU data transfer. In particular, the transfer of the output tensor from the GPU to the CPU was overlapped with the computation of another portion of the output tensor. Asynchronous memory copy operations supported by CUDA favor contiguous data transfers. We therefore identify the outermost dimension of the output tensor as the streaming index. The computation of the output tensor is thus pipelined, with computation of a segment overlapped with the data transfer for another segment. This effectively results in a three-stage pipeline: transfer of a segment of the output tensor from CPU to GPU; computation of the updates to the output tensor segment; and transfer of the updated tensor from GPU to CPU. We also evaluated an equivalent three-stage pipeline that replaces the CPU-to-GPU transfer with a stage on the CPU that takes the output tensor segment computed on the GPU and updates the CPU copy. The two input tensors in these scenarios were much smaller than the output tensor, allowing us to copy them to GPU memory before the pipelined execution begins. The appropriate pipelining strategy is chosen based on the capabilities of the GPU in supporting bidirectional data transfer versus the relative computational cost of the CPU accumulation operation. Note that the various pipelined implementations together with the data transfer operations were generated from the DSL, allowing us to quickly adapt the scheme employed as GPUs evolve.

We observed that the contributions to each segment of the output tensor in the CC triples calculation can be stored entirely on the GPU. We therefore designed an improved version that allocates memory to store a segment of the output tensor in GPU memory and computes all contributions to it across several kernel invocations. Once the tensor segment has been fully computed, the energy contribution can be computed in the GPU. After all such contributions have been computed, the energy scalar is transferred to the CPU. This scheme addresses the increasing disparity between the GPU computational capacity and the CPU–GPU data transfer rate, which dramatically improves performance when the total execution time is bound by the data transfer costs, as is the case on modern GPUs.

14.4.5 CPU–GPU Hybrid Architecture

Modern parallel computing platforms combine GPUs and multicore CPUs. One can use any combination of CPU and GPU resources on such systems, but to obtain the maximum possible achievable performance, it is generally necessary to efficiently utilize all processing cores, both CPU and GPU. To this end, we employ a dynamic load-balancing scheme in which the computation of each block of the output tensor is treated as a task. All contributions to an output tensor segment, a single task involving one or more kernel calls, are computed by the same processing element. This processing element could either be a CPU core or a GPU. The tasks are dynamically assigned to each processing element using a dynamic load balancer across all CPUs and GPUs in the parallel system. Given that GPUs cannot manage inter-process communication, one CPU core is dedicated to managing the communication and dynamic load balancing on behalf of the GPU. The increased computational power of a GPU as compared to a single CPU core enables the former to execute tasks at a much faster rate. The dynamic load balancing employed allows such differing execution rates while ensuring that all processing units are busy computing various parts of the contraction. Figure 14.2 shows the execution steps of the CPU–GPU hybrid implementation.

14.5 Performance

In this section, we discuss the performance of the CPU–GPU implementations of the noniterative CCSD(T) (Reg-CCSD(T)) and MRCCSD(T) formalisms on the example of medium-size molecular systems.

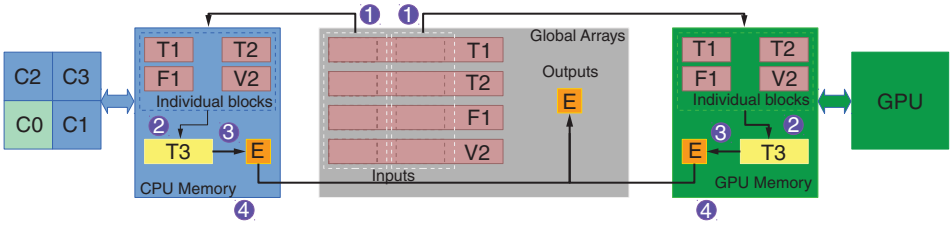


Figure 14.2 *CPU–GPU hybrid implementation: execution steps involved in the noniterative triples correction. T1, T2, and T3 tensors corresponds to the T_1, T_2, and $T_3^{(2)}$ amplitudes (SRCC) or to the reference-specific $T_1^{(\mu)}$, $T_2^{(\mu)}$, and perturbative $T_3^{(\mu)}$ amplitudes (MRCC). The F1 and V2 tensors corresponds to one- and two-electron integrals, respectively. The steps are as follows: step 1, copy input blocks from Global Arrays to the CPU or GPU local memory; step 2, contract input blocks into intermediate tensor block; step 3, reduce intermediate tensor to compute energy correction contribution; step 4, reduce final energy correction across all CPUs and GPUs. One of the CPU cores (Core 0 in the image) manages communication and load balancing toward the GPU, and also performs some basic sequential operations that would be more expensive on the GPU. (See insert for colour representation of this figure)*

14.5.1 CCSD(T) Approach

The first GPU implementation of the CCSD(T) (Reg-CCSD(T)) approach (see [26]) was tested and optimized on two clusters, one with Tesla T10 (C1060) GPUs and the other with Fermi T20 (C2050) cards. The T10 cluster consisted of 64 nodes, while the T20 cluster contained 16 nodes. Each node on the cluster with Tesla T10 GPUs had two Quad-Core Intel Xeon X5560 CPUs, with a frequency of 2.80 GHz, and 8 MB L2 cache. Two nodes shared one Tesla S1070 box, implying that every node had two Tesla T10 GPUs. Each node on the Fermi cluster was equipped with two Quad-Core Intel Xeon E5520 CPUs, with a frequency of 2.27 GHz. Each node had a single GPU. PCI Express 2.0 was used for I/O between the host and the device on both systems. These numerical experiments showed the role of data granularity in taking advantage of heterogeneous computer architectures. In order to test the impact of the `tilesize` on the performance of our GPU implementation, we performed tests for the uracil molecule in the composite 6-31G [114] (localized on hydrogen atoms) and 6-31G** [114] (localized on the carbon atoms) basis set where the spatial symmetry was not invoked. This situation commonly occurs in calculations for large systems without symmetry where `tilesize` can be sufficiently large. The experiments were run on 30 nodes, with two processes on each node. The speedup of the GPU over the CPU version varied from 3 to 8.75. For larger `tilesize`, which implies more FLOPS per process (proportional to $(\texttt{tilesize})^7$), the speedup of GPU was more obvious. While for small tile sizes ($\texttt{tilesize} = 10$) the GPU speedup was rather modest (around 3), for larger tiles ($\texttt{tilesize} = 21$) the speedup was much better (around 8.75). We should expect that a further increase in the tile size should result in a further improvement in the GPU speedup. To verify the generality of this observation, we also evaluated the impact of `tilesize` on the performance of our GPU implementation for the dodecane molecule . We observed speedups improving with `tilesize`, reaching more than a factor of 8 with a `tilesize` of 20.

As an illustrative example of test runs on modern GPU-supported architectures, we discuss tests performed on Titan's hybrid-architecture Cray XK7 system with a theoretical peak performance exceeding 27,000 trillion calculations per second (27 petaflops). It contains both advanced 16-core AMD Opteron CPUs and Nvidia Kepler (K20X) GPUs. As a test case, we chose a pentacene molecule ($C_{22}H_{14}$) described by the cc-pVDZ basis set [115]. All tests were performed using C_1 symmetry. In two representative calculations, 96 nodes were used. In the first run, we used eight CPUs per node and no GPU. In the second Run, we used seven CPUs per node and one GPU. The energies and timings for these runs are shown in Table 14.1. A comparison of timings from Table 14.1 shows that

Table 14.1 *Time comparison for CPU and CPU+GPU runs of the noniterative part of the CCSD(T) approach for the pentacene molecule in cc-pVDZ basis set*

Comput. configuration	CCSD(T) energy (Hartree)	Time (seconds)
Eight CPU per node	−844.4003995	9240.3
Seven CPU + one GPU per node	−844.4003995	1630.7

Tests were performed on the Titan Cray XK7 system at ORNL.

the 7 CPU + 1 GPU run is around 5.6×faster than the run utilizing eight CPUs per node. Based on the earlier analysis, the speedups should be even more favorable for larger `tilesizes`.

14.5.2 MRCCSD(T) Approaches

The development of the GPU implementations of the MRCCSD(T) approaches has been integrated with the recently explored MRCC computational algorithms based on the utilization of the RLP and PGs (for details see Refs [3, 27, 101, 116]). By PG (G_i), one means a partitioning of the processor domain (D) into smaller pieces, which can be symbolically expressed as

$$D = \bigcup_{i=1,\cdots,I} G_i, \tag{14.42}$$

where I is the total number of the PGs. One also assumes that the number of processors in each group (S_i) is the same and is equal to S, that is

$$S_i = S = \frac{N_p}{I} \quad (i = 1, \cdots, I). \tag{14.43}$$

In the above equation, N_p stands for the total number of processors, which is a multiple of the PG number I. The key idea is to distribute the formation of reference-specific MRCC equations over various PGs.

This approach employs two-level parallelism: (i) RLP, where each set of reference-specific equations (or their aggregate) is calculated on separate PGs, and (ii) task-level parallelism used to calculate a given set of reference-specific equations. In the simplest case, the work organization chart (symbolically designated by W) corresponds to the situation when a single PG is delegated to calculate a single set of reference-specific equations ($\mathbf{R}^{(\mu)}$) composed of the direct ($\mathbf{F}^{(\mu)}$) and coupling ($\mathbf{G}^{(\mu)}$) terms. In this case, the number of PGs coincides with the size of the model space (i.e., $I = M$). A more general situation corresponds to the case when each PG G_i forms several ($n_r(i)$) residual vectors $\mathbf{R}^{(\mu)}$. This can be symbolically denoted as

$$W = \bigcup_{i=1,\cdots,I} W_i(n_r(i)), \tag{14.44}$$

where W_i refers to the workload on the corresponding PG G_i. In order to provide best load balancing between the workloads on each PG, it is natural to assume that

$$n_r = n_r(i) = \frac{M}{I} \quad (i = 1, \cdots, I). \tag{14.45}$$

The MRCCSD(T) calculations are composed of two major steps (see Figure 14.3). First, one solves the iterative MRCCSD equations using two-level parallelism. In the second step, the diagonal noniterative triples corrections, Eqs. (14.32) and (14.33), are formed using a similar scheme; that is, each correction can be formed using separate PGs. Additionally, task-level parallelism is enhanced by the utilization of the GPU cores, which contributes to the third-level parallelism.

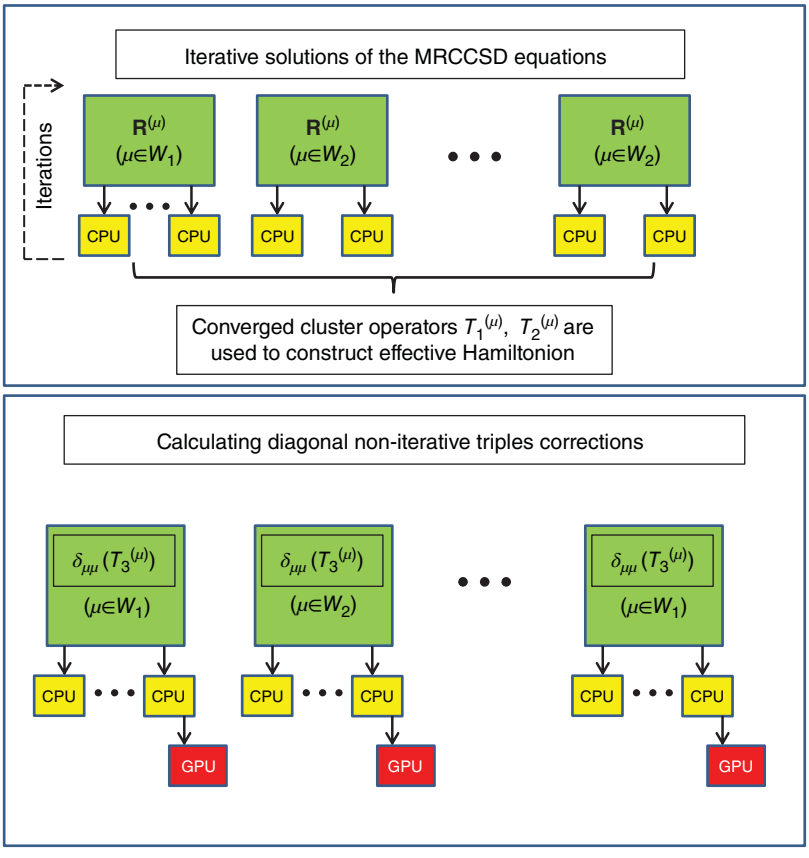


Figure 14.3 *Schematic representation of the GPU-enhanced reference-level parallelism. Separate processor groups are delegated to calculate the reference-specific parts of the iterative MRCCSD equations and reference-specific noniterative corrections due to triples. (See insert for colour representation of this figure)*

The MRCCSD(T) performance tests have been performed for the dodecane molecule with the simple model space defined by two active electrons distributed over two active orbitals. In order to evaluate the relative benefits of using CPUs versus GPUs, we considered several CPU–GPU configurations. Memory requirements limit the number of CPUs we can use on each node to 8. Therefore, we evaluated the performance of computing MRCCSD(T) when using two, four, and eight cores per node. For the hybrid CPU–GPU execution, one of the cores drives the GPU rather than performing calculations. All experiments were performed on up to 24 nodes, which is the largest number of nodes that contain the Tesla M2090 GPUs in the cluster. While the implementation allows us to exploit multiple GPUs per node, this cluster configuration does not enable such an evaluation, limiting our tests to one GPU per node. We expect our implementation to scale reasonably well on multi-GPU configurations given the reduction in communication traffic stemming from the RLP .

In Figure 14.4, we show the speedup achieved by the various CPU–GPU configurations with respect to execution times when run on two CPU cores and no GPUs. We expect that increasing the number of CPUs utilized increases the speedup relative to serial execution. In addition, deploying GPUs speeds up the calculations further. Likewise, increasing the block sizes with a given CPU–GPU

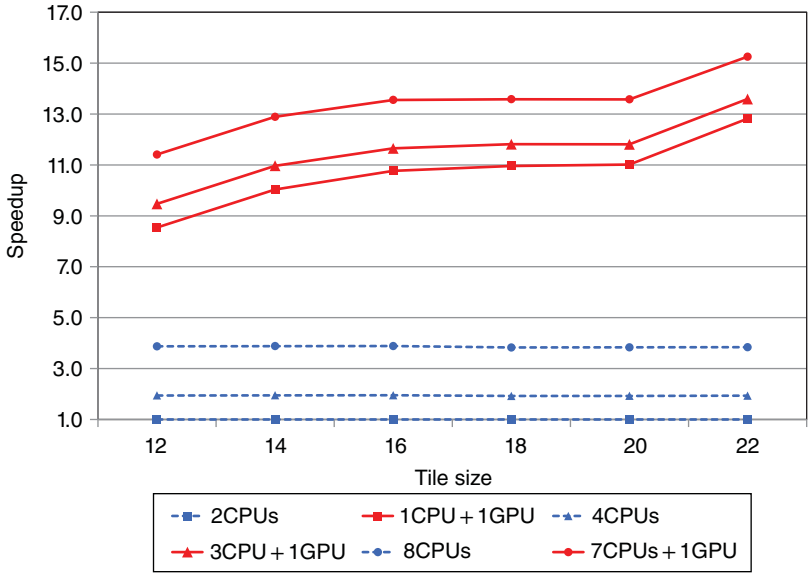


Figure 14.4 *Speedup of the noniterative MRCCSD(T) calculations for various configurations for block sizes varying from 12 through 22 with respect to the execution times using two CPUs and no GPUs*

node configuration also speeds up the calculation. Indeed, we see that doubling the number of CPU cores expectedly results in close to double the performance. This speedup is also independent of block sizes, demonstrating that the sequential CPU execution achieves the same floating-point performance independent of the block size. Unlike execution on the CPU cores, the speedups achieved when using GPUs depends on the block sizes. In particular, when employing one CPU and one GPU, increasing the block size from 12 to 22 provides us a speedup factor of 1.5 without any additional hardware resources. This is due to the improved floating-point performance achieved by the GPU when using larger block sizes. Using one CPU core and one GPU instead of two CPU cores results in a performance improvement factor of 8.5 when using a block size of 12. Comparing the execution time on two CPUs with that of one CPU core and one GPU, we can determine that a single GPU achieves a speedup of 16 over a single CPU core. Given this factor, we would predict, in the absence of superlinear effects, a speedup of 11.5 when employing seven CPU cores and one GPU. We, in fact, achieve a speedup of 11.4. Similarly, with a block size of 22, we determine the speedup achieved by a single GPU as compared to a single CPU core to be 24.6. When using seven CPU cores and one GPU, this should result in a speedup of 15.8. We observe a speedup of 15.3. This close match between the anticipated and observed speedups shows that additional cores are effectively utilized as they are added to the execution.

14.6 Outlook

The availability of the GPU accelerators has the potential to transform the area of molecular simulations employing high-level methods for accurate description of the electron correlation effects. The main reason for this is related to the very high numerical footprint of these approaches. This fact makes the applicability of CC methods inextricably linked to the progress in hardware development and advances in programming models. The development of efficient parallel implementations of the

SRCC and MRCC methods capable of taking advantage of heterogeneous architectures remains one of the most important factors in enabling these expensive methodologies for large molecular systems. The joint utilization of novel parallel tools such as PGs and the possibilities offered by GPU accelerators provides a venue to significantly extend the area of application of noniterative CCSD(T) and MRCCSD(T) methods. Our tests clearly indicate the role of GPUs in accelerating the most numerically expensive part of the CCSD(T) and MRCCSD(T) calculations where the corrections due to triples are calculated. We demonstrated that our parallel implementation effectively load-balances the work between the CPUs and GPUs, adapting to their differing computational capabilities. Our tests also showed that our implementations achieve close to anticipated speedups with increasing core counts and demonstrated that the computational resources are effectively utilized. The developed capabilities have already enabled us to perform accurate simulations for systems and processes which until recently were considered to be too numerically challenging. These include the MRCC calculations for complicated excited states characterized by collective multi-electron excitations, low-spin open-shell electronic states, and combustion processes. As far as future development of high-level *ab initio* methods is concerned, we envision further development of novel algorithms utilizing GPU technology especially in the theoretical/computational areas associated with various tensor decomposition techniques, which may lead to an unprecedented paradigm change in the applicability of CC formalisms.

Acknowledgments

This work has been supported by the Extreme Scale Computing Initiative (K.B.-N., S.K., O.V., H.J.J.v.D., K.K.), a Laboratory Directed Research and Development Program at Pacific Northwest National Laboratory. A large portion of the research was performed using PNNL Institutional Computing at Pacific Northwest National Laboratory and EMSL, a national scientific user facility sponsored by the Department of Energy's Office of Biological and Environmental Research and located at Pacific Northwest National Laboratory. The Pacific Northwest National Laboratory is operated for the U.S. Department of Energy by the Battelle Memorial Institute under contract no. DEAC0676RLO-1830. Large-scale BW-MRCCSD calculations have been performed using a 2012 ASCR Leadership Computing Challenge (ALCC) award (K.B.-N., S.K., E.A., K.K.) allocation at Oak Ridge Leadership Computing Facility (OLCF).

References

1. Čížek, J. (1966) On the correlation problem in atomic and molecular systems. Calculation of wavefunction components in ursell-type expansion using quantum-field theoretical methods. *J. Chem. Phys.*, **45**, 4256–4266.
2. de Jong, W.A., Bylaska, E., Govind, N., Janssen, C.L., Kowalski, K., Mueller, T., Nielsen, I.M.B., van Dam, H.J.J., Veryazov, V. and Lindh, R. (2010) Utilizing high performance computing for chemistry: parallel computational chemistry. *Phys. Chem. Chem. Phys.*, **12**, 6896–6920.
3. Brabec, J., Pittner, J., van Dam, H.J.J., Apra, E. and Kowalski, K. (2012) Parallel implementation of multireference coupled-cluster theories based on the reference-level parallelism. *J. Chem. Theory Comput.*, **8**, 487–497.
4. Götz, A.W., Woelfle, T. and Walker, R.C. (2010) Quantum chemistry on graphics processing units, in *Annual Reports in Computational Chemistry*, Annual Reports in Computational Chemistry, vol. **6** (ed. RA Wheeler, Elsevier, pp. 21–35.
5. Xu, D., Williamson, M.J. and Walker, R.C. (2010) Advancements in molecular dynamics simulations of biomolecules on graphical processing units, in *Annual Reports in Computational*

Chemistry, Annual Reports in Computational Chemistry, vol. **6** (ed. RA Wheeler), Elsevier, pp. 3–19.

6. Anderson, A.G., Goddard, W.A. III and Schroeder, P. (2007) Quantum Monte Carlo on graphical processing units. *Comput. Phys. Commun.*, **177**, 298–306.
7. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krueger, J., Lefohn, A.E. and Purcell, T.J. (2007) A survey of general-purpose computation on graphics hardware. *Comput. Graphics Forum*, **26**, 80–113.
8. Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G. and Schulten, K. (2007) Accelerating molecular modeling applications with graphics processors. *J. Comput. Chem.*, **28**, 2618–2640.
9. Hardy, D.J., Stone, J.E. and Schulten, K. (2009) Multilevel summation of electrostatic potentials using graphics processing units. *Parallel Comput.*, **35**, 164–177.
10. Stone, J.E., Hardy, D.J., Ufimtsev, I.S. and Schulten, K. (2010) GPU-accelerated molecular modeling coming of age. *J. Mol. Graphics Modell.*, **29**, 116–125.
11. Salomon-Ferrer, R., Götz, A.W., Poole, D., Le Grand, S. and Walker, R.C. (2013) Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. Explicit Solvent Particle Mesh Ewald. *J. Chem. Theory Comput.*, **9**, 3878–3888.
12. Le Grand, S., Götz, A.W. and Walker, R.C. (2013) SPFP: speed without compromise-A mixed precision model for GPU accelerated molecular dynamics simulations. *Comput. Phys. Commun.*, **184**, 374–380.
13. Götz, A.W., Williamson, M.J., Xu, D., Poole, D., Le Grand, S. and Walker, R.C. (2012) Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. Generalized born. *J. Chem. Theory Comput.*, **8**, 1542–1555.
14. Yasuda, K. (2008) Two-electron integral evaluation on the graphics processor unit. *J. Comput. Chem.*, **29**, 334–342.
15. Yasuda, K. (2008) Accelerating density functional calculations with graphics processing unit. *J. Chem. Theory Comput.*, **4**, 1230–1236.
16. Ufimtsev, I.S. and Martinez, T.J. (2008) Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation. *J. Chem. Theory Comput.*, **4**, 222–231.
17. Ufimtsev, I.S. and Martinez, T.J. (2009) Quantum chemistry on graphical processing units. 2. Direct self-consistent-field implementation. *J. Chem. Theory Comput.*, **5**, 1004–1015.
18. Ufimtsev, I.S. and Martinez, T.J. (2009) Quantum chemistry on graphical processing units. 3. Analytical energy gradients, geometry optimization, and first principles molecular dynamics. *J. Chem. Theory Comput.*, **5**, 2619–2628.
19. Isborn, C.M., Luehr, N., Ufimtsev, I.S. and Martinez, T.J. (2011) Excited-state electronic structure with configuration interaction singles and Tamm-Dancoff time-dependent density functional theory on graphical processing units. *J. Chem. Theory Comput.*, **7**, 1814–1823.
20. Titov, A.V., Ufimtsev, I.S., Luehr, N. and Martinez, T.J. (2013) Generating efficient quantum chemistry codes for novel architectures. *J. Chem. Theory Comput.*, **9**, 213–221.
21. Vogt, L., Olivares-Amaya, R., Kermes, S., Shao, Y., Amador-Bedolla, C. and Aspuru-Guzik, A. (2008) Accelerating resolution-of-the-identity second-order Moller-Plesset quantum chemistry calculations with graphical processing units. *J. Phys. Chem. A*, **112**, 2049–2057, Conference in Honor of Professor William A Lester on his 70th Birthday, University California, Berkeley, CA, MAR, 2007.
22. Friedrichs, M.S., Eastman, P., Vaidyanathan, V., Houston, M., Legrand, S., Beberg, A.L., Ensign, D.L., Bruns, C.M. and Pande, V.S. (2009) Accelerating molecular dynamic simulation on graphics processing units. *J. Comput. Chem.*, **30**, 864–872.
23. van Meel, J.A., Arnold, A., Frenkel, D., Zwart, S.F.P. and Belleman, R.G. (2008) Harvesting graphics power for MD simulations. *Mol. Simul.*, **34**, 259–266.

24. Eastman, P. and Pande, V.S. (2010) Efficient nonbonded interactions for molecular dynamics on a graphics processing unit. *J. Comput. Chem.*, **31**, 1268–1272.
25. DePrince, A.E. III and Hammond, J.R. (2011) Coupled cluster theory on graphics processing units I. The coupled cluster doubles method. *J. Chem. Theory Comput.*, **7**, 1287–1295.
26. Ma, W., Krishnamoorthy, S., Villa, O. and Kowalski, K. (2011) GPU-based implementations of the noniterative regularized-CCSD(T) corrections: applications to strongly correlated systems. *J. Chem. Theory Comput.*, **7**, 1316–1327.
27. Bhaskaran-Nair, K., Ma, W., Krishnamoorthy, S., Villa, O., van Dam, H.J.J., Apra, E. and Kowalski, K. (2013) Noniterative multireference coupled cluster methods on heterogeneous CPU-GPU systems. *J. Chem. Theory Comput.*, **9**, 1949–1957.
28. Asadchev, A., Allada, V., Felder, J., Bode, B.M., Gordon, M.S. and Windus, T.L. (2010) Uncontracted Rys quadrature implementation of up to G functions on graphical processing units. *J. Chem. Theory Comput.*, **6**, 696–704.
29. Asadchev, A. and Gordon, M.S. (2012) New multithreaded hybrid CPU/GPU approach to Hartree-Fock. *J. Chem. Theory Comput.*, **8**, 4166–4176.
30. Asadchev, A. and Gordon, M.S. (2013) Fast and flexible coupled cluster implementation. *J. Chem. Theory Comput.*, **9**, 3385–3392.
31. Miao, Y. and Merz, K.M. Jr. (2013) Acceleration of electron repulsion integral evaluation on graphics processing units via use of recurrence relations. *J. Chem. Theory Comput.*, **9**, 965–976.
32. Wu, X., Koslowski, A. and Thiel, W. (2012) Semiempirical quantum chemical calculations accelerated on a hybrid multicore CPU-GPU computing platform. *J. Chem. Theory Comput.*, **8**, 2272–2281.
33. Wilkinson, K. and Skylaris, C.-K. (2013) Porting ONETEP to graphical processing unit-based coprocessors. 1. FFT box operations. *J. Comput. Chem.*, **34**, 2446–2459.
34. Coester, F. (1958) Bound states of a many-particle system. *Nucl. Phys.*, **7**, 421–424.
35. Coester, F. and Kümmel, H. (1960) Short-range correlations in nuclear wave functions. *Nucl. Phys.*, **17**, 477–485.
36. Paldus, J., Shavitt, I. and Čížek, J. (1972) Correlation problems in atomic and molecular systems.4. Extended coupled-pair many-electron theory and its application to BH3 molecule. *Phys. Rev. A*, **5**, 50–67.
37. Purvis, G. and Bartlett, R. (1982) A full coupled-cluster singles and doubles model - the inclusion of disconnected triples. *J. Chem. Phys.*, **76**, 1910–1918.
38. Goldstone, J. (1957) Derivation of the Brueckner many-body theory. *Proc. R. Soc. London, Ser. A Math. Phys. Sci.*, **239**, 267–279.
39. Noga, J. and Bartlett, R. (1987) The full CCSDT model for molecular electronic-structure. *J. Chem. Phys.*, **86**, 7041–7050.
40. Noga, J. and Bartlett, R. (1988) Erratum: the full CCSDT model for molecular electronic-structure. *J. Chem. Phys.*, **89**, 3041.
41. Scuseria, G. and Schaefer, H. (1988) A new implementation of the full CCSDT model for molecular electronic-structure. *Chem. Phys. Lett.*, **152**, 382–386.
42. Kucharski, S. and Bartlett, R. (1991) Recursive intermediate factorization and complete computational linearization of the coupled-cluster single, double, triple, and quadruple excitation equations. *Theor. Chim. Acta*, **80**, 387–405.
43. Oliphant, N. and Adamowicz, L. (1991) Coupled-cluster method truncated at quadruples. *J. Chem. Phys.*, **95**, 6645–6651.
44. Møller, C. and Plesset, M. (1934) Note on an approximation treatment for many-electron systems. *Phys. Rev.*, **46**, 0618–0622.
45. Raghavachari, K., Trucks, G., Pople, J. and Head-Gordon, M. (1989) A 5th-order perturbation comparison of electron correlation theories. *Chem. Phys. Lett.*, **157**, 479–483.

46. Urban, M., Noga, J., Cole, S. and Bartlett, R. (1985) Towards a full CCSDT model for electron correlation. *J. Chem. Phys.*, **83**, 4041–4046.
47. Stanton, J. and Gauss, J. (1995) Perturbative treatment of the similarity transformed Hamiltonian in equation-of-motion coupled-cluster approximations. *J. Chem. Phys.*, **103**, 1064–1076.
48. Stanton, J. and Gauss, J. (1996) A simple correction to final state energies of doublet radicals described by equation-of-motion coupled cluster theory in the singles and doubles approximation. *Theor. Chim. Acta*, **93**, 303–313.
49. Stanton, J. (1997) Why CCSD(T) works: a different perspective. *Chem. Phys. Lett.*, **281**, 130–134.
50. Crawford, T. and Stanton, J. (1998) Investigation of an asymmetric triple-excitation correction for coupled-cluster energies. *Int. J. Quantum Chem.*, **70**, 601–611.
51. Kucharski, S. and Bartlett, R. (1998) Noniterative energy corrections through fifth-order to the coupled cluster singles and doubles method. *J. Chem. Phys.*, **108**, 5243–5254.
52. Gwaltney, S. and Head-Gordon, M. (2000) A second-order correction to singles and doubles coupled-cluster methods based on a perturbative expansion of a similarity-transformed Hamiltonian. *Chem. Phys. Lett.*, **323**, 21–28.
53. Gwaltney, S., Sherrill, C., Head-Gordon, M. and Krylov, A. (2000) Second-order perturbation corrections to singles and doubles coupled-cluster methods: general theory and application to the valence optimized doubles model. *J. Chem. Phys.*, **113**, 3548–3560.
54. Hirata, S., Nooijen, M., Grabowski, I. and Bartlett, R. (2001) Perturbative corrections to coupled-cluster and equation-of-motion coupled-cluster energies: a determinantal analysis. *J. Chem. Phys.*, **114**, 3919–3928.
55. Bomble, Y., Stanton, J., Kallày, M. and Gauss, J. (2005) Coupled-cluster methods including noniterative corrections for quadruple excitations. *J. Chem. Phys.*, **123**, 054101.
56. Kallày, M. and Gauss, J. (2005) Approximate treatment of higher excitations in coupled-cluster theory. *J. Chem. Phys.*, **123**, 214105.
57. Taube, A.G. and Bartlett, R.J. (2008) Improving upon CCSD(T): Lambda CCSD(T). I. Potential energy surfaces. *J. Chem. Phys.*, **128**, 044110.
58. Taube, A.G. and Bartlett, R.J. (2008) Improving upon CCSD(T): Lambda CCSD(T). II. Stationary formulation and derivatives. *J. Chem. Phys.*, **128**, 044111.
59. Kowalski, K. and Piecuch, P. (2000) The method of moments of coupled-cluster equations and the renormalized CCSD[T], CCSD(T), CCSD(TQ), and CCSDT(Q) approaches. *J. Chem. Phys.*, **113**, 18–35.
60. Piecuch, P. and Wloch, M. (2005) Renormalized coupled-cluster methods exploiting left eigenstates of the similarity-transformed Hamiltonian. *J. Chem. Phys.*, **123**, 224105.
61. Piecuch, P., Wloch, M., Gour, J. and Kinal, A. (2006) Single-reference, size-extensive, non-iterative coupled-cluster approaches to bond breaking and biradicals. *Chem. Phys. Lett.*, **418**, 467–474.
62. Kowalski, K. and Fan, P.-D. (2009) Generating functionals based formulation of the method of moments of coupled cluster equations. *J. Chem. Phys.*, **130**, 084112.
63. Kowalski, K. and Valiev, M. (2009) Extensive regularization of the coupled cluster methods based on the generating functional formalism: application to gas-phase benchmarks and to the S(N)2 reaction of CHCl3 and OH- in water. *J. Chem. Phys.*, **131**, 234107.
64. Taube, A.G. and Bartlett, R.J. (2009) Rethinking linearized coupled-cluster theory. *J. Chem. Phys.*, **130**, 144112.
65. Brandow, B. (1967) Linked-cluster expansions for nuclear many-body problem. *Rev. Mod. Phys.*, **39**, 771–828.
66. Wolinski, K. and Pulay, P. (1989) Generalized Moller-plesset perturbation-theory - 2nd order results for 2-configuration, open-shell excited singlet, and doublet wave-functions. *J. Chem. Phys.*, **90**, 3647–3659.

67. Zarrabian, S. and Paldus, J. (1990) Applicability of multireference many-body perturbation-theory to the determination of potential-energy surfaces - a model study. *Int. J. Quantum Chem.*, **38**, 761–778.
68. Hirao, K. (1992) Multireference Møller-plesset method. *Chem. Phys. Lett.*, **190**, 374–380.
69. Nakano, H. (1993) Quasi-degenerate perturbation-theory with multiconfigurational self-consistent-field reference functions. *J. Chem. Phys.*, **99**, 7983–7992.
70. Kozlowski, P. and Davidson, E. (1994) Considerations in constructing a multireference 2nd-order perturbation-theory. *J. Chem. Phys.*, **100**, 3672–3682.
71. Finley, J., Chaudhuri, R. and Freed, K. (1995) Applications of multireference perturbation-theory to potential-energy surfaces by optimal partitioning of H: intruder states avoidance and convergence enhancement. *J. Chem. Phys.*, **103**, 4990–5010.
72. Chaudhuri, R., Freed, K., Hose, G., Piecuch, P., Kowalski, K., Wloch, M., Chattopadhyay, S., Mukherjee, D., Rolik, Z., Szabados, A., Toth, G. and Surjan, P. (2005) Comparison of low-order multireference many-body perturbation theories. *J. Chem. Phys.*, **122**, 134105.
73. Hoffmann, M.R., Datta, D., Das, S., Mukherjee, D., Szabados, A., Rolik, Z. and Surjan, P.R. (2009) Comparative study of multireference perturbative theories for ground and excited states. *J. Chem. Phys.*, **131**, 204104.
74. Andersson, K., Malmqvist, P. and Roos, B. (1992) 2nd-order perturbation-theory with a complete active space self-consistent field reference function. *J. Chem. Phys.*, **96**, 1218–1226.
75. Werner, H. and Knowles, P. (1988) An efficient internally contracted multiconfiguration reference configuration-interaction method. *J. Chem. Phys.*, **89**, 5803–5814.
76. Mukherjee, D., Moitra, R. and Mukhopadhyay, A. (1975) Correlation problem in open-shell atoms and molecules - non-perturbative linked cluster formulation. *Mol. Phys.*, **30**, 1861–1888.
77. Lindgren, I. (1978) Coupled-cluster approach to the many-body perturbation-theory for open-shell systems. *Int. J. Quantum. Chem.*, **12**, 33–58.
78. Mukherjee, D., Moitra, R. and Mukhopadhyay, A. (1977) Applications of a non-perturbative many-body formalism to general open-shell atomic and molecular problems - calculation of ground and lowest $\pi - \pi\star$ singlet and triplet energies and 1st ionization-potential of trans-butadiene. *Mol. Phys.*, **33**, 955–969.
79. Mukherjee, D., Moitra, R. and Mukhopadhyay, A. (1977) Core-valence separation and use of non-orthogonal basic sets in non-perturbative open-shell many-body formalism. *Indian J. Pure Appl. Phys.*, **15**, 623–628.
80. Landau, A., Eliav, E. and Kaldor, U. (1999) Intermediate Hamiltonian Fock-space coupled-cluster method. *Chem. Phys. Lett.*, **313**, 399–403.
81. Haque, A. and Kaldor, U. (1985) Open-shell coupled-cluster theory applied to atomic and molecular-systems. *Chem. Phys. Lett.*, **117**, 347–351.
82. Rittby, M., Pal, S. and Bartlett, R. (1989) Multireference coupled-cluster method - ionization-potentials and excitation-energies for ketene and diazomethane. *J. Chem. Phys.*, **90**, 3214–3220.
83. Stolarczyk, L. and Monkhorst, H. (1985) Coupled-cluster method in Fock space .1. General formalism. *Phys. Rev. A*, **32**, 725–742.
84. Jeziorski, B. and Paldus, J. (1989) Valence universal exponential ansatz and the cluster structure of multireference configuration-interaction wave-function. *J. Chem. Phys.*, **90**, 2714–2731.
85. Meissner, L. (1998) Fock-space coupled-cluster method in the intermediate Hamiltonian formulation: model with singles and doubles. *J. Chem. Phys.*, **108**, 9227–9235.
86. Meissner, L. (2012) Various formulations of the Fock-space coupled-cluster method: advantages and disadvantages in their practical implementations. *Chem. Phys.*, **401**, 136–145.
87. Jeziorski, B. and Monkhorst, H. (1981) Coupled-cluster method for multideterminantal reference states. *Phys. Rev. A*, **24**, 1668–1681.

88. Meissner, L., Jankowski, K. and Wasilewski, J. (1988) A coupled-cluster method for quasidegenerate states. *Int. J. Quantum Chem.*, **34**, 535–557.
89. Paldus, J., Piecuch, P., Pylypow, L. and Jeziorski, B. (1993) Application of hilbert-space coupled-cluster theory to simple (H2)2 model systems - planar models. *Phys. Rev. A*, **47**, 2738–2782.
90. Kucharski, S. and Bartlett, R. (1991) Hilbert-space multireference coupled-cluster methods .1. The single and double excitation model. *J. Chem. Phys.*, **95**, 8227–8238.
91. Balková, A., Kucharski, S., Meissner, L. and Bartlett, R. (1991) A hilbert-space multireference coupled-cluster study of the H-4 model system. *Theor. Chim. Acta*, **80**, 335–348.
92. Meissner, L. and Bartlett, R. (1989) The general-model space effective Hamiltonian in order-for-order expansion. *J. Chem. Phys.*, **91**, 4800–4808.
93. Meissner, L. and Bartlett, R. (1990) A general model-space coupled-cluster method using a hilbert-space approach. *J. Chem. Phys.*, **92**, 561–567.
94. Li, X. and Paldus, J. (2003) General-model-space state-universal coupled-cluster theory: connectivity conditions and explicit equations. *J. Chem. Phys.*, **119**, 5320–5333.
95. Masik, J. and Hubac, I. (1997) Multireference Brillouin-Wigner coupled-cluster theory. Single-root approach, in *Advances in Quantum Chemistry: Quantum Systems in Chemistry and Physics, Part I, European Union*, Advances in Quantum Chemistry, vol. **31**, pp. 75–104, 2nd European Workshop on Quantum Systems in Chemistry and Physics, Jesus Coll, London, England, Apr 06-09, 1997.
96. Pittner, J. (2003) Continuous transition between Brillouin-Wigner and Rayleigh-Schrodinger perturbation theory, generalized Bloch equation, and Hilbert space multireference coupled cluster. *J. Chem. Phys.*, **118**, 10876–10889.
97. Mahapatra, U., Datta, B. and Mukherjee, D. (1998) A state-specific multi-reference coupled cluster formalism with molecular applications. *Mol. Phys.*, **94**, 157–171.
98. Mahapatra, U. Datta, B., Bandyopadhyay, B. and Mukherjee, D. State-specific multi-reference coupled cluster formulations: two paradigms, in *Advances in Quantum Chemistry: Modern Trends in Atomic Physics*, Advances in Quantum Chemistry, vol. **30**, Swedish Academy of Sciences; Goteborg University, pp. 163–193.
99. Evangelista, F.A., Allen, W.D. and Schaefer, H.F. III (2007) Coupling term derivation and general implementation of state-specific multireference coupled cluster theories. *J. Chem. Phys.*, **127**, 024102.
100. Yanai, T. and Chan, G. (2006) Canonical transformation theory for multireference problems. *J. Chem. Phys.*, **124**, 194106.
101. Bhaskaran-Nair, K., Brabec, J., Apra, E., van Dam, H.J.J., Pittner, J. and Kowalski, K. (2012) Implementation of the multireference Brillouin-Wigner and Mukherjee's coupled cluster methods with non-iterative triple excitations utilizing reference-level parallelism. *J. Chem. Phys.*, **137**, 094112.
102. Bhaskaran-Nair, K., Demel, O. and Pittner, J. (2010) Multireference Mukherjee's coupled cluster method with triexcitations in the linked formulation: efficient implementation and applications. *J. Chem. Phys.*, **132**, 154105.
103. Demel, O. and Pittner, J. (2006) Multireference Brillouin-Wigner coupled clusters method with noniterative perturbative connected triples. *J. Chem. Phys.*, **124**, 144112.
104. Bhaskaran-Nair, K., Demel, O. and Pittner, J. (2008) Multireference state-specific Mukherjee's coupled cluster method with noniterative triexcitations. *J. Chem. Phys.*, **129**, 184105.
105. Bhaskaran-Nair, K., Demel, O., Smydke, J. and Pittner, J. (2011) Multireference state-specific Mukherjee's coupled cluster method with noniterative triexcitations using uncoupled approximation. *J. Chem. Phys.*, **134**, 154106.
106. Evangelista, F.A., Prochnow, E., Gauss, J. and Schaefer, H.F. III (2010) Perturbative triples corrections in state-specific multireference coupled cluster theory. *J. Chem. Phys.*, **132**, 074107.

107. Valiev, M., Bylaska, E.J., Govind, N., Kowalski, K., Straatsma, T.P., Van Dam, H.J.J., Wang, D., Nieplocha, J., Apra, E., Windus, T.L. and de Jong, W. (2010) NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations. *Comput. Phys. Commun.*, **181**, 1477–1489.
108. Kendall, R.A., Aprà, E., Bernholdt, D.E., Bylaska, E.J., Dupuis, M., Fann, G.I., Harrison, R.J., Ju, J., Nichols, J.A., Nieplocha, J., Straatsma, T.P., Windus, T.L. and Wong, A.T. (2000) High performance computational chemistry: an overview of NWChem a distributed parallel application. *Comput. Phys. Commun.*, **128**, 260–283.
109. Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H. and Apra, E. (2006) Advances, applications and performance of the global arrays shared memory programming toolkit. *Int. J. High Perform. Comput. Appl.*, **20**, 203–231.
110. Hirata, S. (2003) Tensor contraction engine: abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *J. Phys. Chem. A*, **107**, 9887–9897.
111. NVIDIA Corporation (2012) NVIDIA CUDA C Programming Guide, Version 5.0.
112. NVIDIA Corporation (2012) NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Whitepaper. Available at: `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf` (accessed 22 September 2015).
113. NVIDIA Corporation (2013) NVIDIA Tesla GPU accelerators, Available at: `http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf` (accessed 22 September 2015).
114. Hehre, W., Ditchfield, R. and Pople, J. (1972) Self-consistent molecular-orbital methods. 12. Further extensions of gaussian-type basis sets for use in molecular-orbital studies of organic-molecules. *J. Chem. Phys.*, **56**, 2257–2261.
115. Dunning, T. (1989) Gaussian-basis sets for use in correlated molecular calculations. 1. The atoms boron through neon and hydrogen. *J. Chem. Phys.*, **90**, 1007–1023.
116. Kowalski, K., Bhaskaran-Nair, K., Brabec, J. and Pittner, J. (2013) Coupled cluster theories for strongly correlated molecular systems, in *Strongly Correlated Systems: Numerical Methods*, Springer Series in Solid-State Sciences, vol. **176**, Springer-Verlag, pp. 237–271, see `http://www.springer.com/us/book/9783642351051`.

Scientific Index

Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

Technical Index

Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

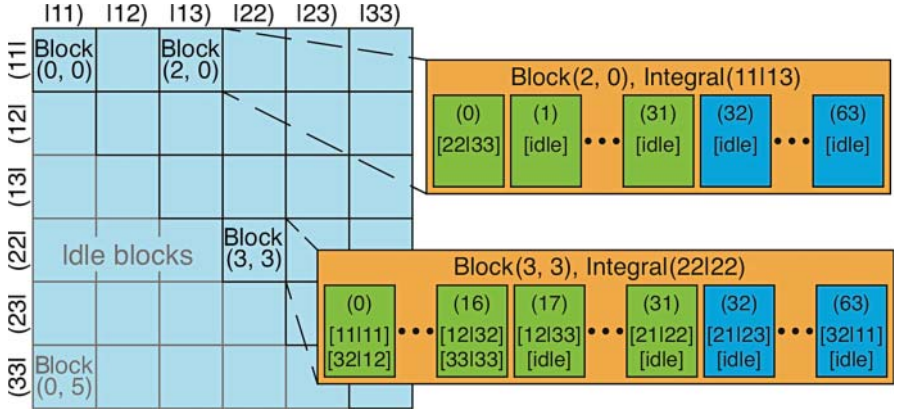


Figure 4.1 *Schematic of one-block one-contracted Integral (1B1CI) mapping. Cyan squares on left represent contracted ERIs each mapped to the labeled CUDA block of 64 threads. Orange squares show mapping of primitive ERIs to CUDA threads (green and blue boxes, colored according to CUDA warp) for two representative integrals, the first a contraction over a single primitive ERI and the second involving* $3^4 = 81$ *primitive contributions*

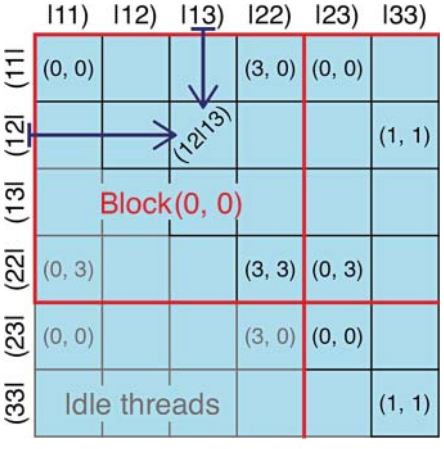


Figure 4.2 *Schematic of one-thread one-contracted Integral (1T1CI) mapping. Cyan squares represent contracted ERIs and CUDA threads. Thread indices are shown in parentheses. Each CUDA block (red outlines) computes 16 ERIs, with each thread accumulating the primitives of an independent contraction, in a local register*

Electronic Structure Calculations on Graphics Processing Units:
From Quantum Chemistry to Condensed Matter Physics, First Edition.
Edited by Ross C. Walker and Andreas W. Götz.

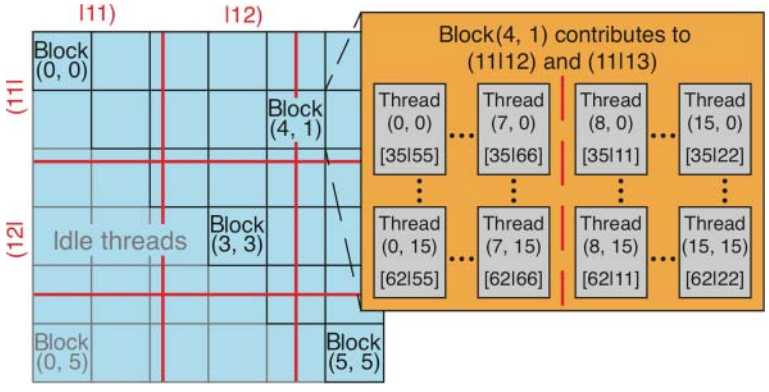


Figure 4.3 *Schematic of one-thread one-primitive integral (1T1PI) mapping. Cyan squares represent two-dimensional tiles of* 16×16 *primitive ERIs, each of which is assigned to a* 16×16 *CUDA block as labeled. Red lines indicate divisions between contracted ERIs. The orange box shows assignment of primitive ERIs to threads (gray squares) within a block that contains contributions to multiple contractions*

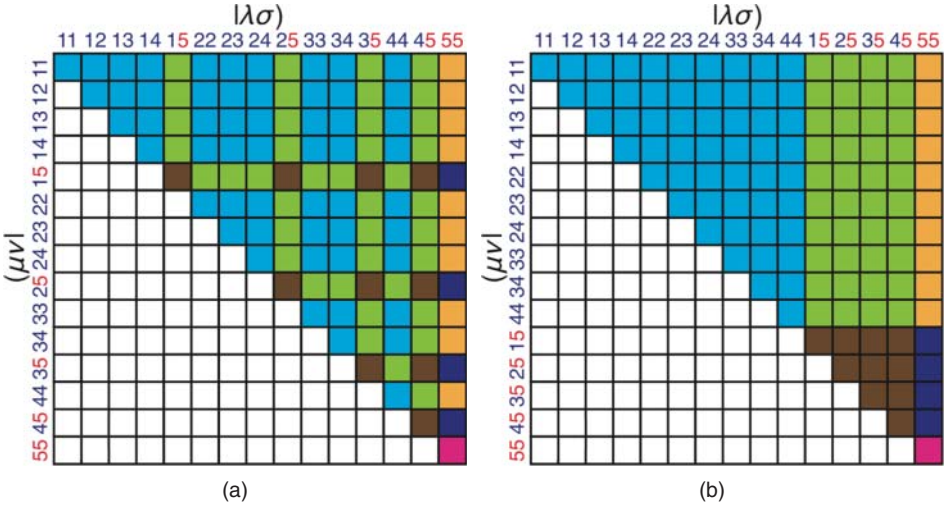


Figure 4.4 *ERI grids colored by angular momentum class for a system containing four s-shells and one p-shell. Each square represents all ERIs for a shell quartet. (a) Grid when bra and ket pairs are ordered by simple loops over shells. (b) ERI grid for same system with bra and ket pairs sorted by angular momentum, ss, then sp, then pp. Each integral class now handles a contiguous chunk of the total ERI grid*

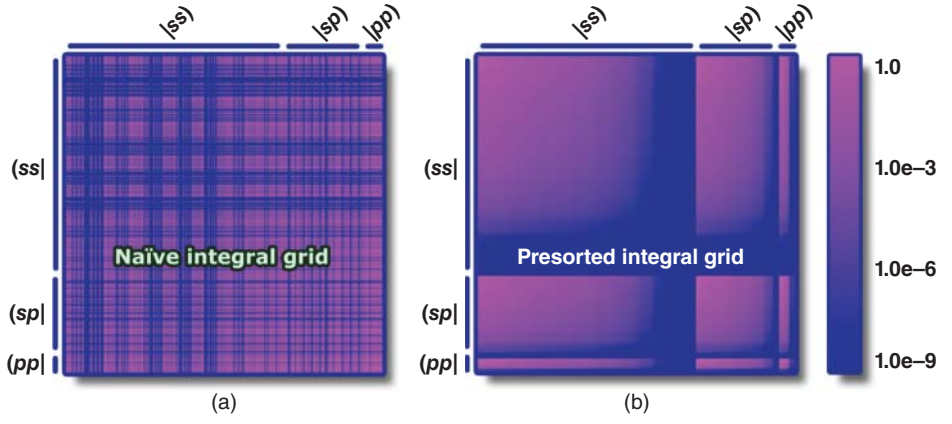


Figure 4.5 *Organization of ERIs for Coulomb formation. Rows and columns correspond to primitive bra and ket pairs, respectively. Each ERI is colored according to the magnitude of its Schwarz bound. Data are derived from calculation on ethane molecule. Figure (a) obtained by arbitrary ordering of pairs within each angular momentum class and suffers from load imbalance because large and small integrals are computed in neighboring cells, and (b) that sorts bra and ket primitives by Schwarz contribution within each momentum class, providing an efficient structure for parallel evaluation*

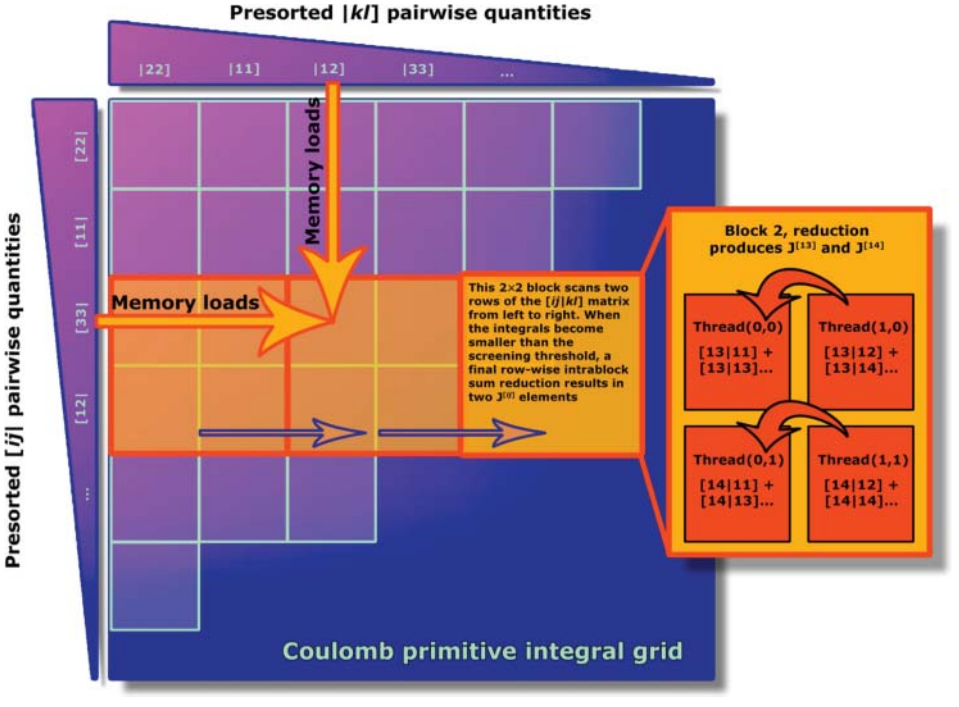


Figure 4.6 *Schematic representation of a J-Engine kernel for one angular momentum class, for example, (ss|ss). Cyan squares represent significant ERI contributions. Sorted bra and ket vectors are represented by triangles to the left and above the grid. The path of a 2×2 block as it sweeps across the grid is shown in orange. The final reduction across rows of the block is illustrated within the inset to the right*

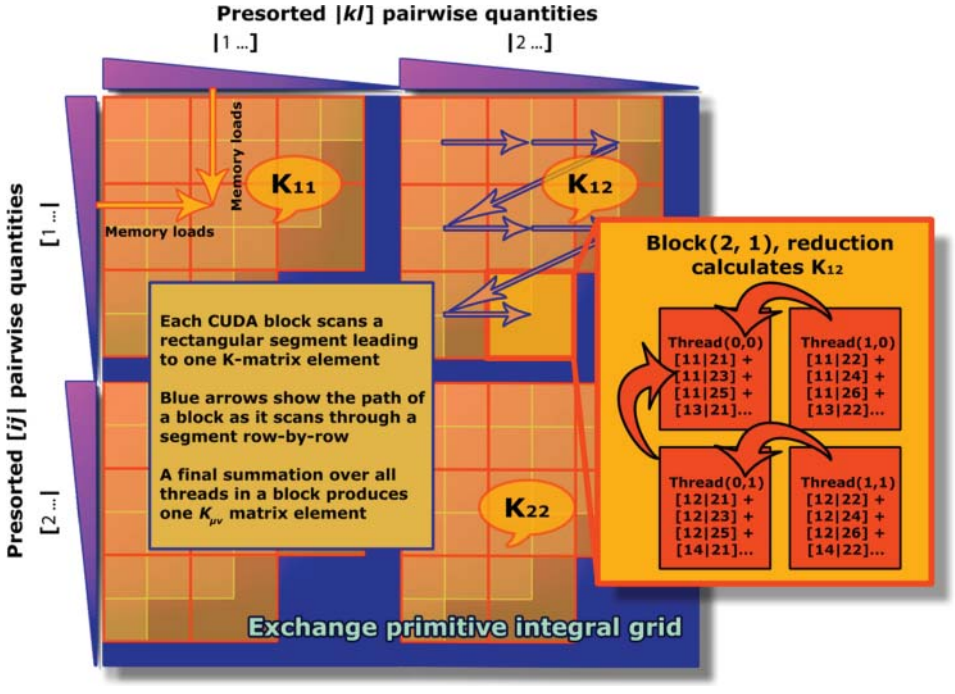


Figure 4.7 *Schematic of a K-Engine kernel. Bra and ket PQ arrays are represented by triangles to the left and above the grid. The pairs are grouped by ν and λ index and then sorted by bound. The paths of four blocks are shown in orange, with the zigzag pattern illustrated by arrows in the top right. The final reduction of an exchange element within a 2 × 2 block is shown to the right*

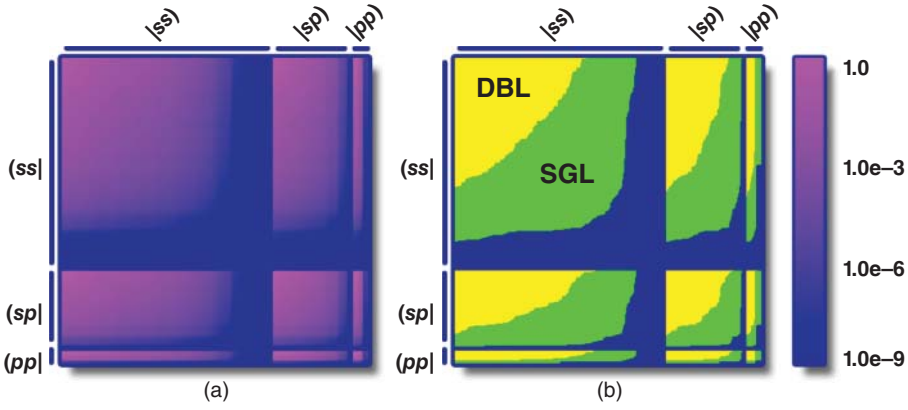


Figure 4.8 *Organization of double- and single-precision workloads within Coulomb ERI grids. As in Figure 4.5, rows and columns correspond to primitive bra and ket pairs. (a) Each ERI is colored according to the magnitude of its Schwarz bound. (b) ERIs are colored by required precision. Yellow ERIs require double precision, while those in green may be evaluated in single precision. Blue ERIs are neglected entirely*

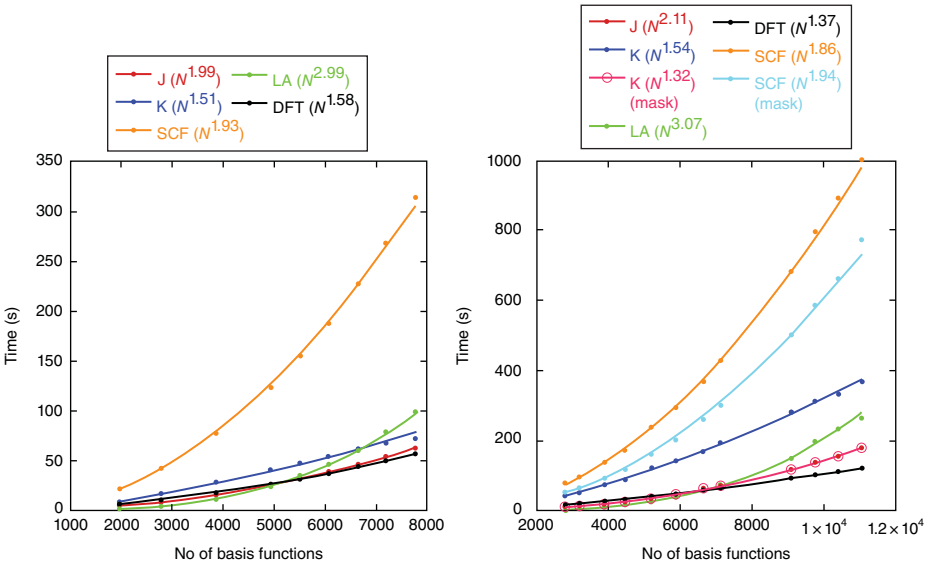


Figure 4.11 *First SCF iteration timings in seconds for (a) linear alkenes and (b) cubic water clusters. Total times are further broken down into J-Engine, K-Engine, distance-masked K-Engine, linear algebra (LA), and DFT exchange–correlation contributions. For water clusters, total SCF times are shown for both the naïve and distance-masked (mask) K-Engine. All calculations were performed using a single Tesla M2090 GPU and the 6-31G basis set. Power fits show scaling with increasing system size, and the exponent for each fit is provided in the legend*

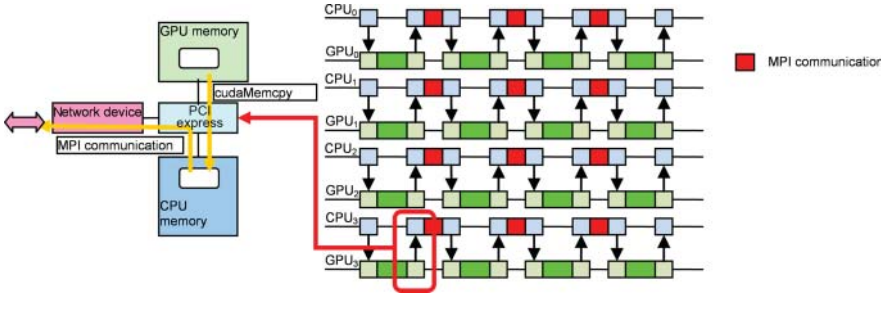


Figure 7.2 *Using multiple process accelerated by GPUs communicating with MPI*

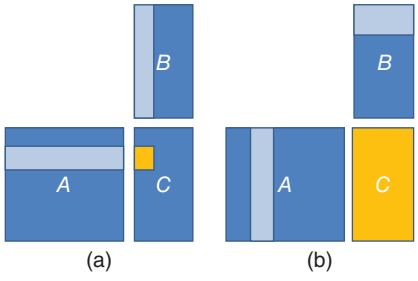


Figure 8.7 *Inner-product (a) and outer-product (b) form of matrix multiplication. The yellow areas in C indicate elements that can be computed independently by accessing the highlighted areas of A and B*

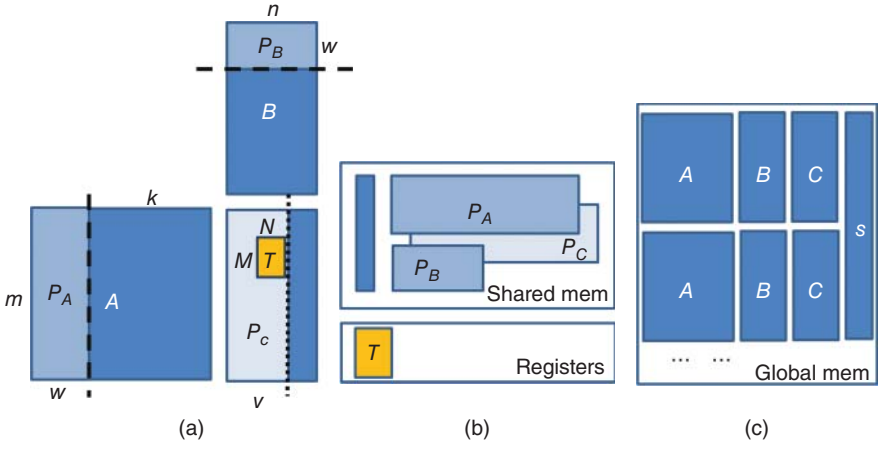


Figure 8.8 *(a) Parameterization of the $m \times n \times k$-matrix product $C = C + AB$. Each thread computes an $M \times N$ tile (T) of the result matrix C. In order to accommodate matrix sizes larger than the available shared memory, matrices are processed in slabs (P_A, P_B), with an input slab width w. In order to optimize the data output, the matrices (P_C) are written back using the output slab width v. (b) Close to the SM, registers are used to store the C matrix tile, while slabs of A, B, and C are stored in shared memory. (c) GPU memory stores all panel data, including the various blocks of A, B, C, and the stack buffers S*

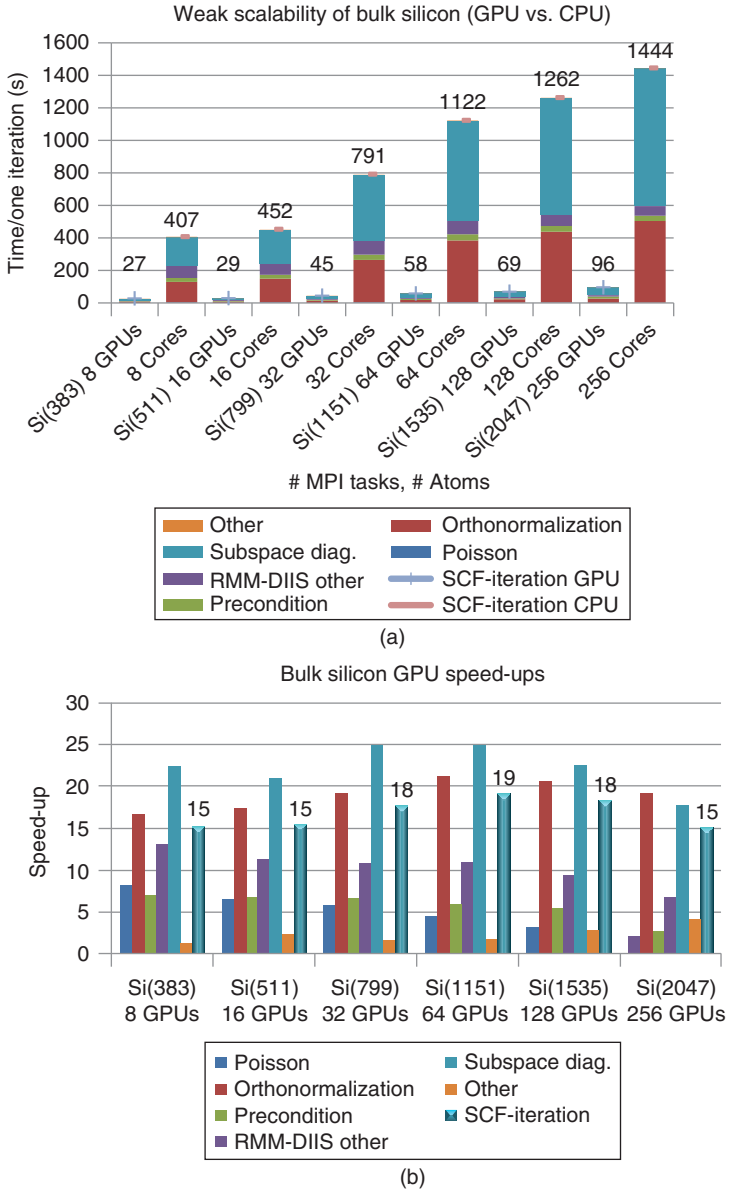


Figure 9.3 *(a) Weak scaling performance of the CPU and GPU versions of the program using bulk Si systems. (b) The achieved speedups with GPU acceleration. The GPU runs used one CPU core per GPU*

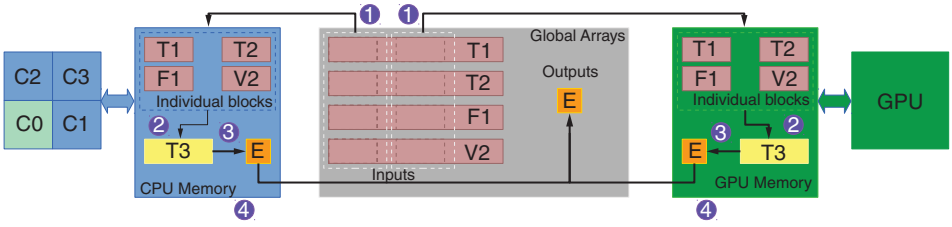


Figure 14.2 *CPU–GPU hybrid implementation: execution steps involved in the noniterative triples correction. T1, T2, and T3 tensors corresponds to the T_1, T_2, and $T_3^{(2)}$ amplitudes (SRCC) or to the reference-specific $T_1^{(\mu)}$, $T_2^{(\mu)}$, and perturbative $T_3^{(\mu)}$ amplitudes (MRCC). The F1 and V2 tensors corresponds to one- and two-electron integrals, respectively. The steps are as follows: step 1, copy input blocks from Global Arrays to the CPU or GPU local memory; step 2, contract input blocks into intermediate tensor block; step 3, reduce intermediate tensor to compute energy correction contribution; step 4, reduce final energy correction across all CPUs and GPUs. One of the CPU cores (Core 0 in the image) manages communication and load balancing toward the GPU, and also performs some basic sequential operations that would be more expensive on the GPU*

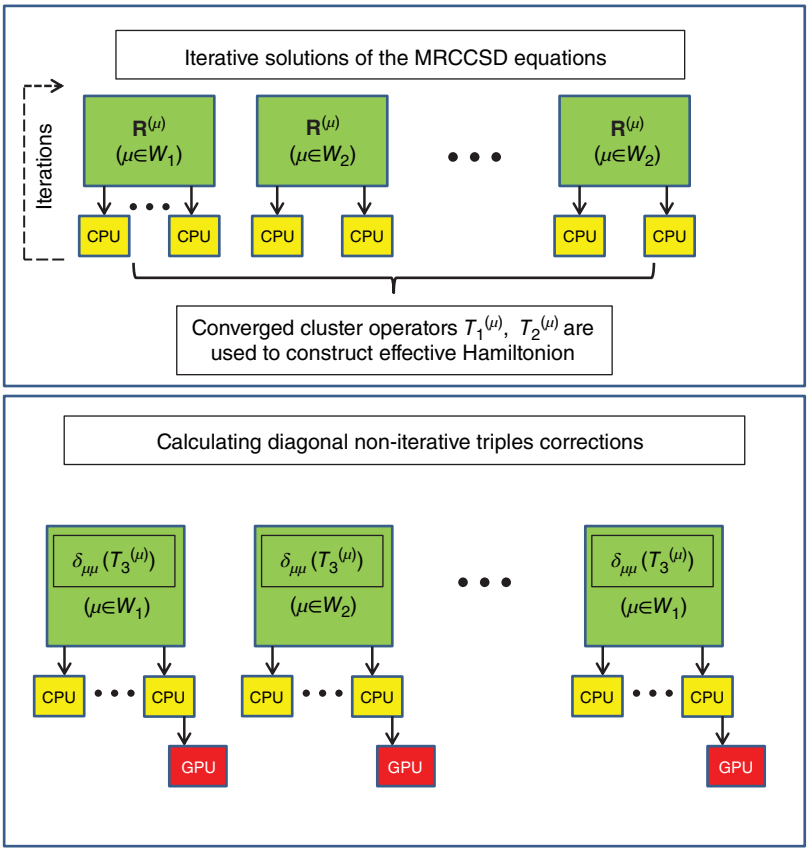


Figure 14.3 *Schematic representation of the GPU-enhanced reference-level parallelism. Separate processor groups are delegated to calculate the reference-specific parts of the iterative MRCCSD equations and reference-specific noniterative corrections due to triples*

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.