Introduction to OpenCL

Introduction to OpenCL

- OpenCL - **Open C**omputing **L**anguage
  - Open, royalty-free standard
  - Initially proposed by Apple
  - Specification maintained by the Khronos Group
  - Developed by a number of companies
  - Specification: set of requirements to be satisfied ⇒ must be implemented to use it
  - Device agnostic
- Framework for parallel programming across heterogeneous platforms consisting of:
  - CPUs, GPUs and other processors (FPGA, ...)
- Similar: Nvidia's CUDA

Main Idea

- **Main Idea of OpenCL:** Replace loops with data-parallel functions (**kernels**) that execute at each point in a problem domain

Traditional vector addition loop in C

```
void vec_add(int N,
                  const float *a,
                  const float *b,
                        float *c)
{
  int i;
  for(i=0; i<N; i++)
    c[i] = a[i] + b[i];
}
```

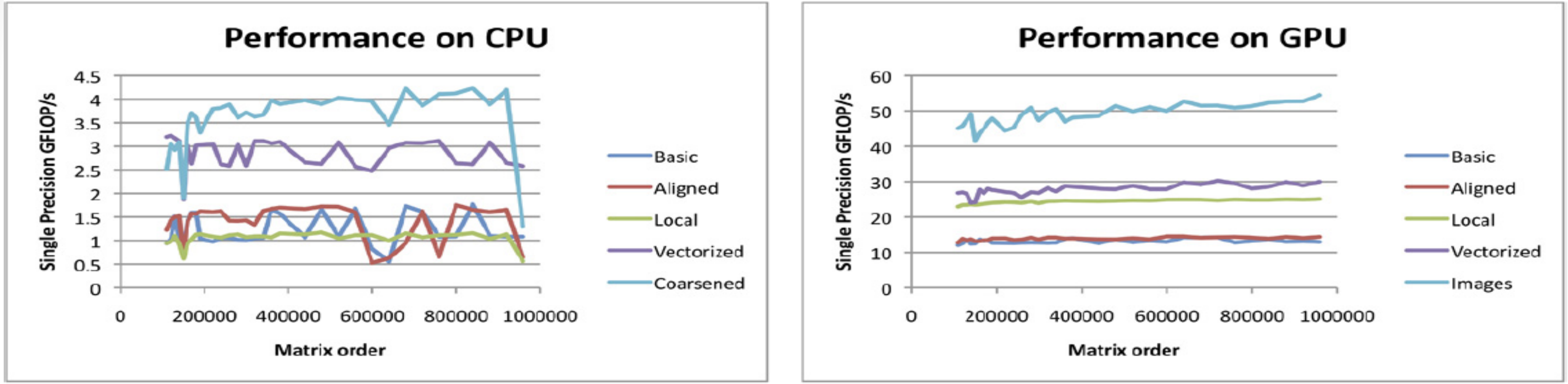
⇒

Vector addition OpenCL kernel

```
__kernel void vec_add(
        __global const float *a,
        __global const float *b,
        __global       float *c)
{
  int gid = get_global_id(0);
  c[gid] = a[gid] + b[gid];
}
```

- Code comparison - note differences:
  - Loop over N elements ⇒ N kernel instances execute in parallel
  - Qualifiers: __kernel, __global
  - Each kernel instance has a global identification number

Motivation

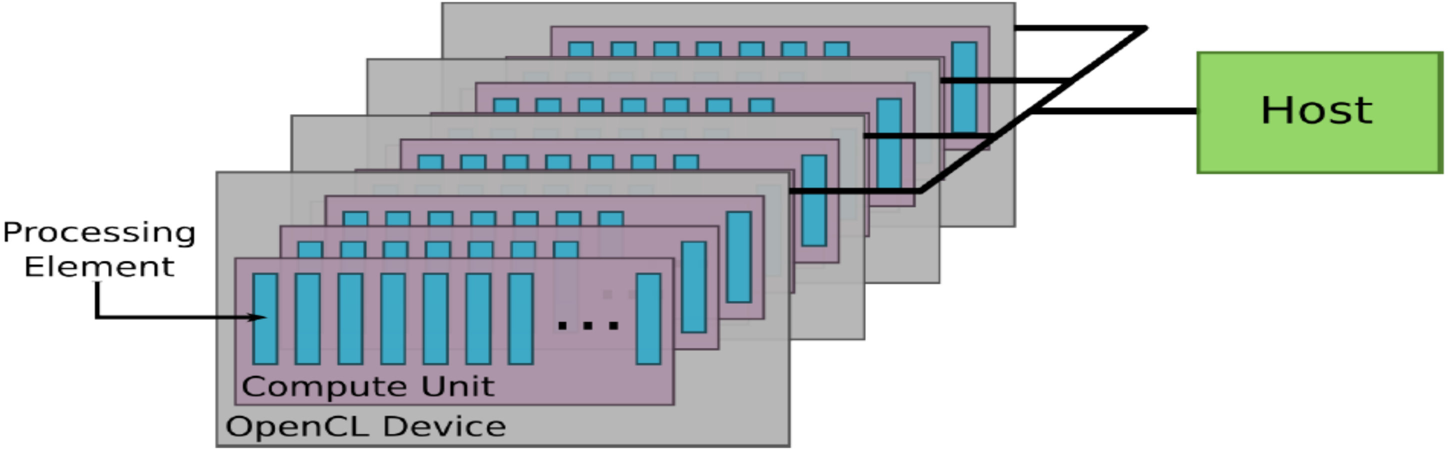


- AMD OpenCL Optimization Case Study: Diagonal Sparse Matrix Vector Multiplication
  - AMD Phenom II X4 965 CPU (quad core)
  - ATI Radeon HD 5870 GPU
  - Unoptimized CPU performance: 1 SP GFLOP/s
  - Optimized CPU performance reaches: 4 SP GFLOP/s
  - Optimized GPU performance reaches: 50 SP GFLOP/s

OpenCL Models

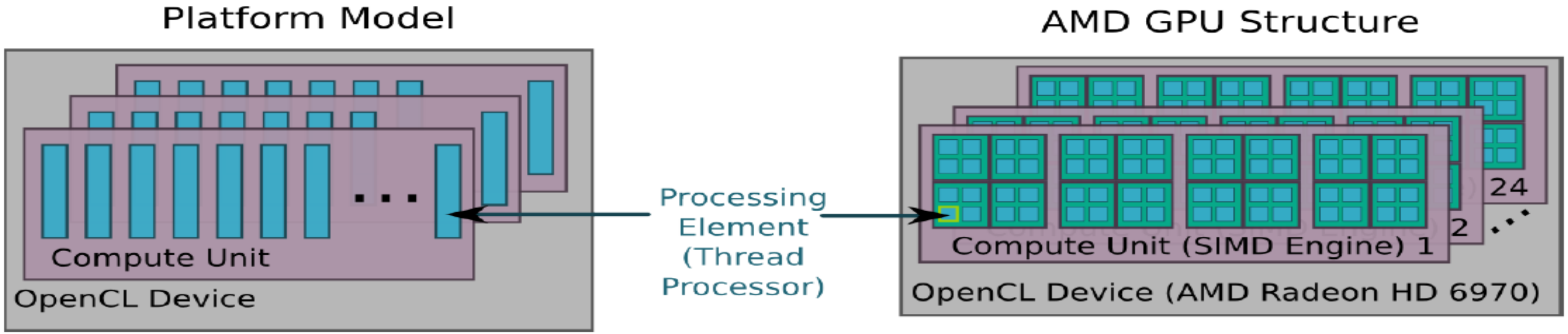
- Platform Model
- Execution Model
- Programming Model
- Memory Model

Platform Model



- Platform: one **Host** + one or more **OpenCL Devices**
    - **OpenCL Device:** divided into one or more **compute units**
        - **Compute unit:** divided into one or more **processing elements**
- Platform model designed to present a uniform view of many different kinds of parallel processors

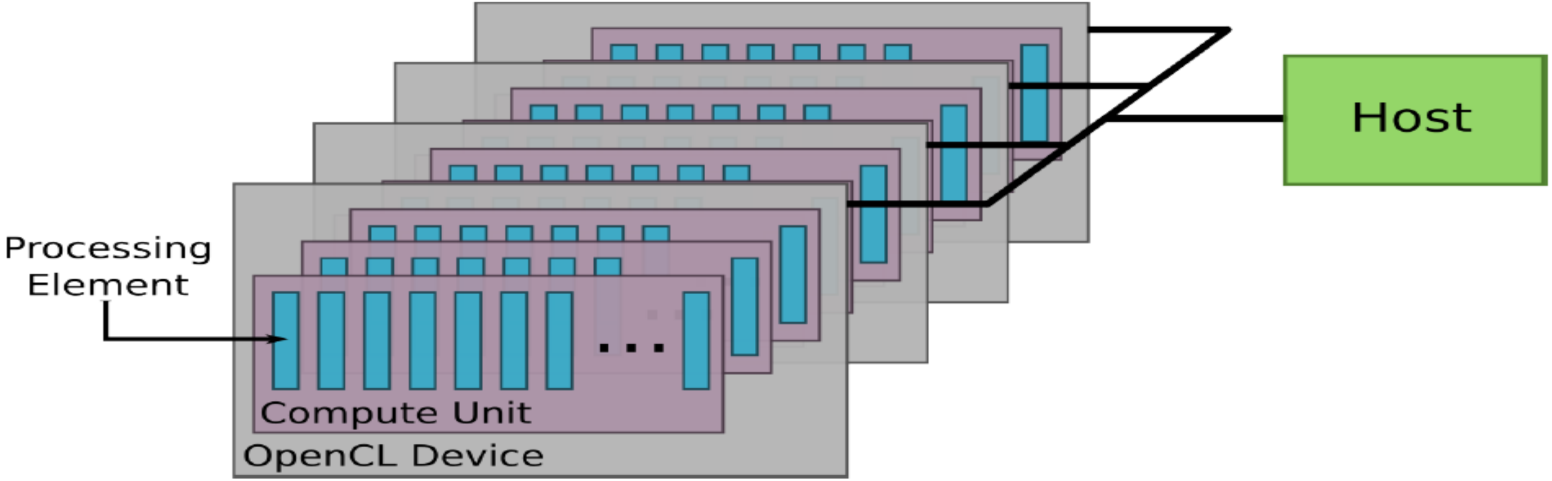
Platform Model Mapped onto AMD GPU


Platform Model

AMD GPU Structure

- OpenCL Device Example: AMD Radeon HD 6970
  - 24 compute units (SIMD engines or processors)
    - SIMD - Single Instruction, Multiple Data
    - High level of parallelism within a processor
  - 64 processing elements per compute unit
    = 1536 total processing elements

Execution Model

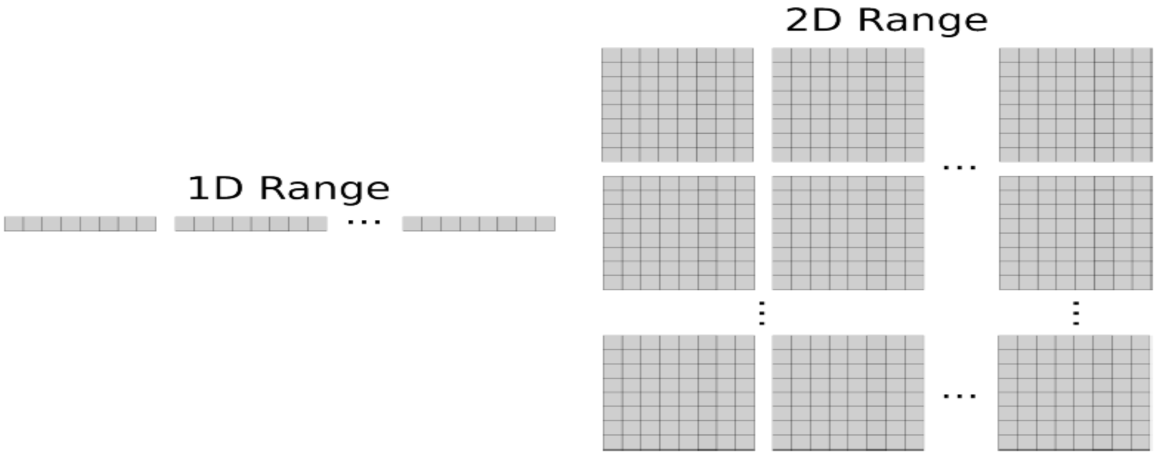


- OpenCL Application
  - Host Code
    - Written in C/C++
    - Executes on host
    - Submits work to OpenCL device(s)
  - Device Code
    - Written in OpenCL C
    - Executes on device(s)

Programming Model

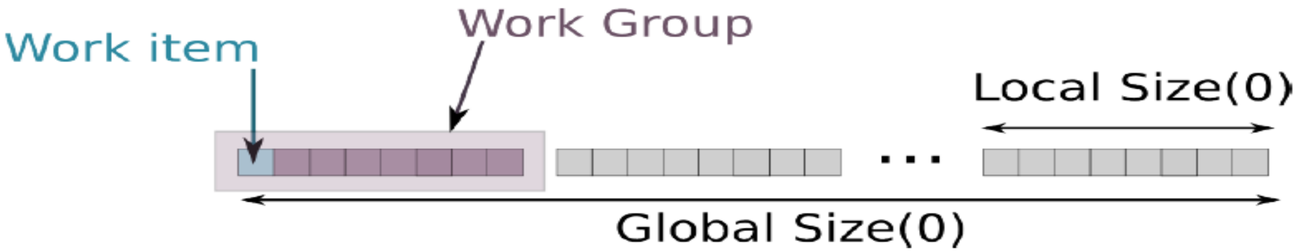
- Data-parallel programming:
    - Set of instructions are applied in parallel to each point in some abstract domain of indices.
        - On a SIMD processor, data parallelism achieved by performing the same task on many different pieces of data in parallel
        - Compare to MPI, where different processors perform the same task in parallel
    - Example: 8x8 Matrix addition
        - MPI with a 2-processor system: CPU A could add all elements from top half of matrices, CPU B could add all elements from bottom half - each CPU performs 32 additions serially
        - OpenCL on AMD GPU: Each of the 64 processing elements on the SIMD processor performs 1 addition
- Task-parallel programming:
    - Multiple parallel tasks

Programming Model: Data-Parallelism



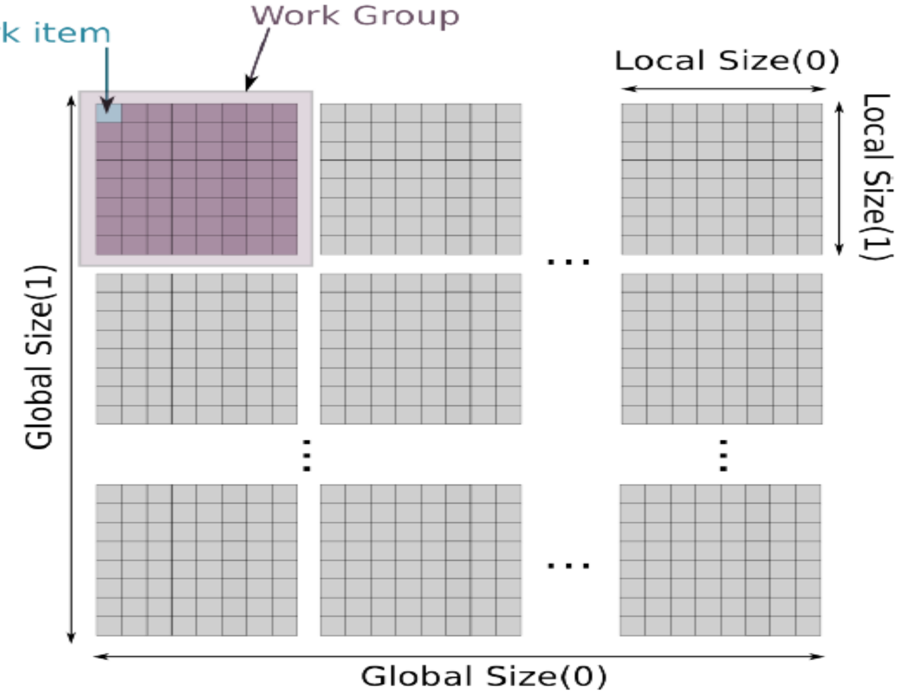
- Define N-Dimensional computation domain (N = 1,2 or 3)

Data-Parallelism with 1D Index Space



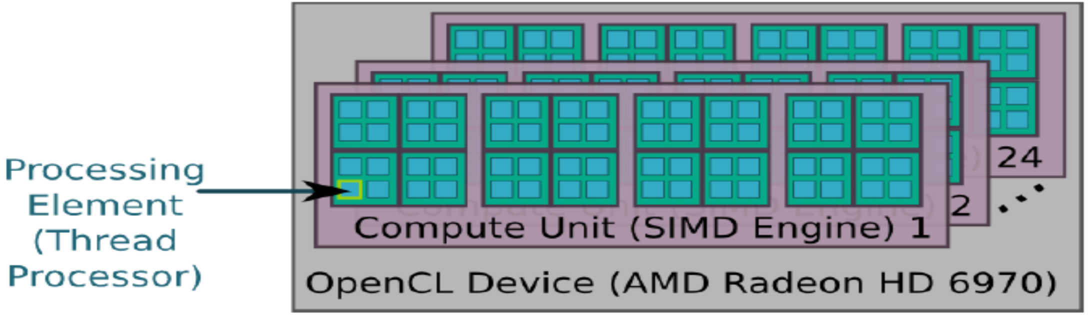
- When a kernel is submitted for execution, an index space is defined
- A kernel instance (**work item**, CUDA: thread) executes for each point in index space
- Each work item executes the same code but the path taken and data operated upon can vary per work item
- Work items organized into **work groups** (CUDA: thread blocks)
    - Assigned a unique work group ID
    - Work group synchronization

Data-Parallelism with 2D Index Space



- Example: processing a 1024 × 1024 image:
  Global Size(0) = Global Size(1) = 1024
  1 kernel execution per pixel ⇒ 1,048,576 total kernel executions

AMD GPU: Work Item Processing



- All processing elements within SIMD engine execute same instruction
- **Wavefront**: block of work-items that are executed together
- Wavefronts execute N work items in parallel, where N is specific to the GPU
    - For AMD Radeon HD 6970, $N = 64$ as there are 64 processing elements per SIMD engine
    - Consequence on branching

Memory Model

Private Memory (CUDA: local)

- Private to a work item, not visible to other work items

Local Memory (CUDA: shared)

- Shared within a work group

Constant Memory

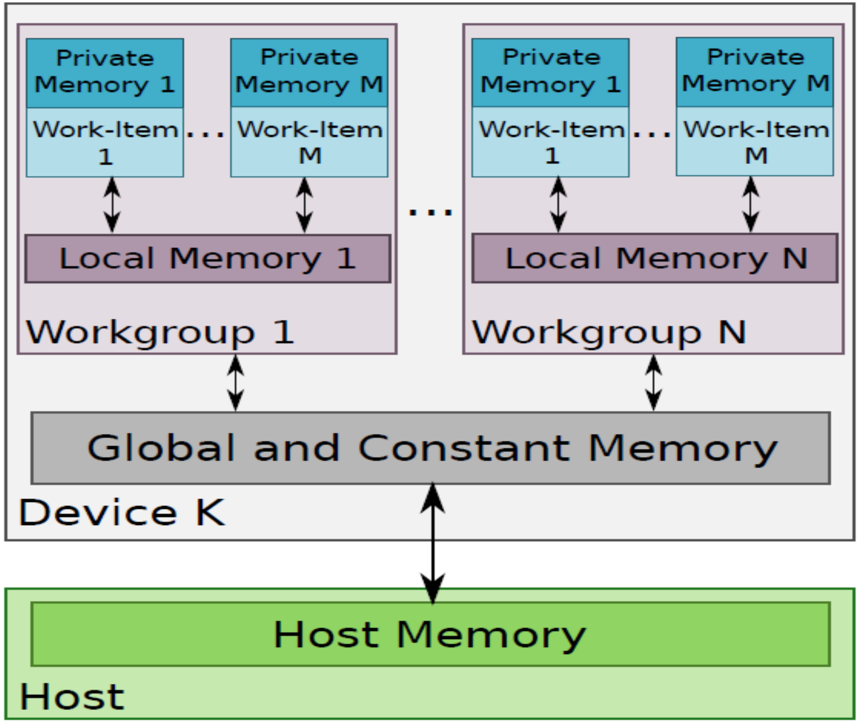
- Visible to all workgroups, read-only

Global Memory

- Accessible to all work items and the host

Host Memory

- Host-accessible



OpenCL Framework

- Platform Layer
  - Allows host to discover OpenCL devices and create contexts
- Runtime
  - Allows host to manipulate contexts (memory management, command execution..)
- OpenCL C Programming Language
  - Supports a subset of the ISO C99 language with extensions for parallelism
  - Device memory hierarchy ⇒ Address space qualifiers (__global, __local..)
  - Extensions for parallelism - support for work items (get_global_id), work groups (get_group_id, get_local_id), synchronization

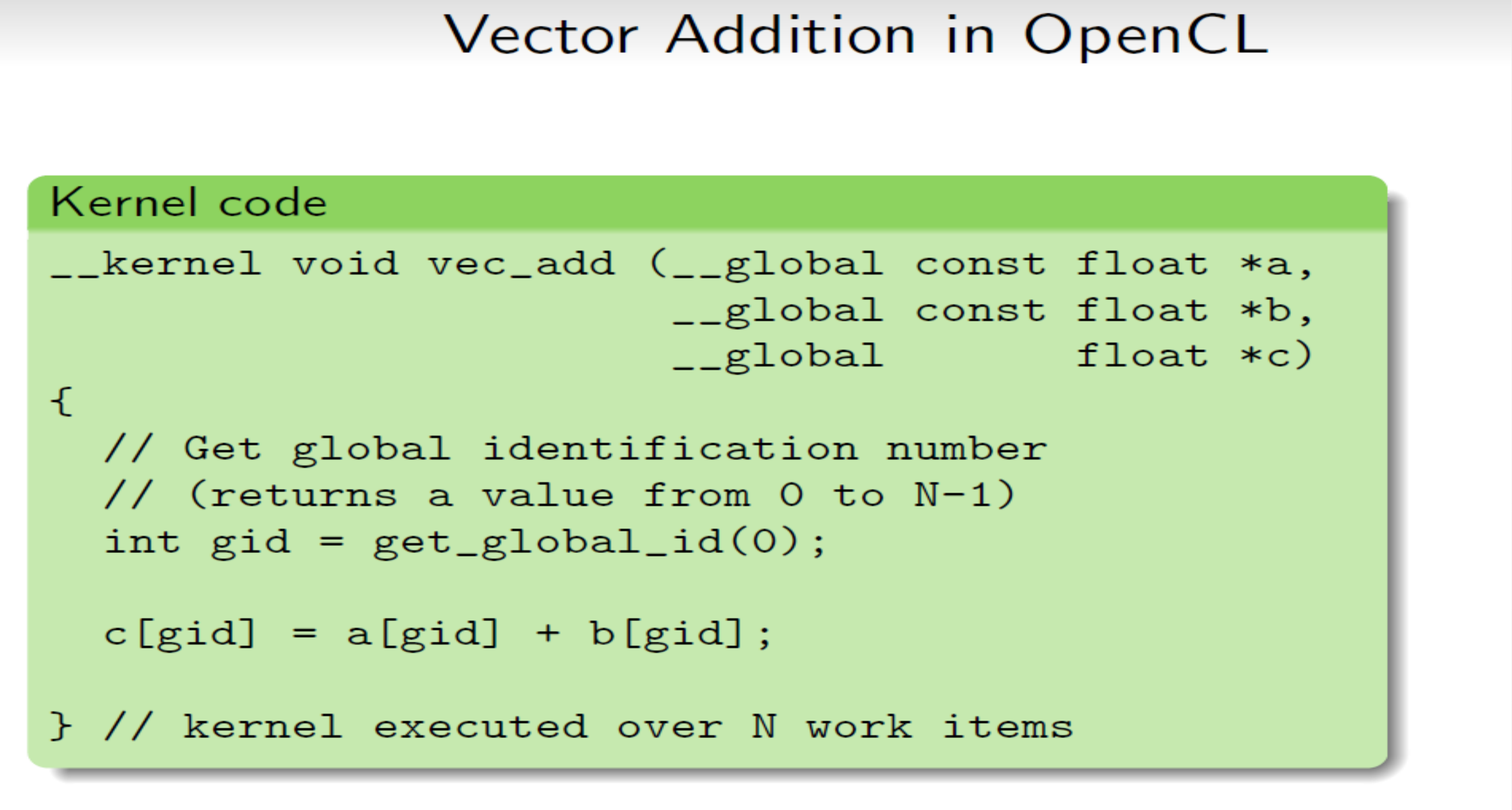
Vector Addition Example

- Simple example:

Vector Addition in C

```
void vector_add_c(const float *a,
                  const float *b,
                  float *c,
                  int N)
{
  int i;
  for(i=0; i<N; i++)
    c[i] = a[i] + b[i];
}
```

- For the OpenCL solution to this problem, there are two parts:
  - Kernel code
  - Host code



Vector Addition in OpenCL

Kernel code

```
__kernel void vec_add (__global const float *a,
                       __global const float *b,
                       __global       float *c)
{
  // Get global identification number
  // (returns a value from 0 to N-1)
  int gid = get_global_id(0);

  c[gid] = a[gid] + b[gid];

} // kernel executed over N work items
```

Vector Addition in OpenCL

Inline Kernel Code

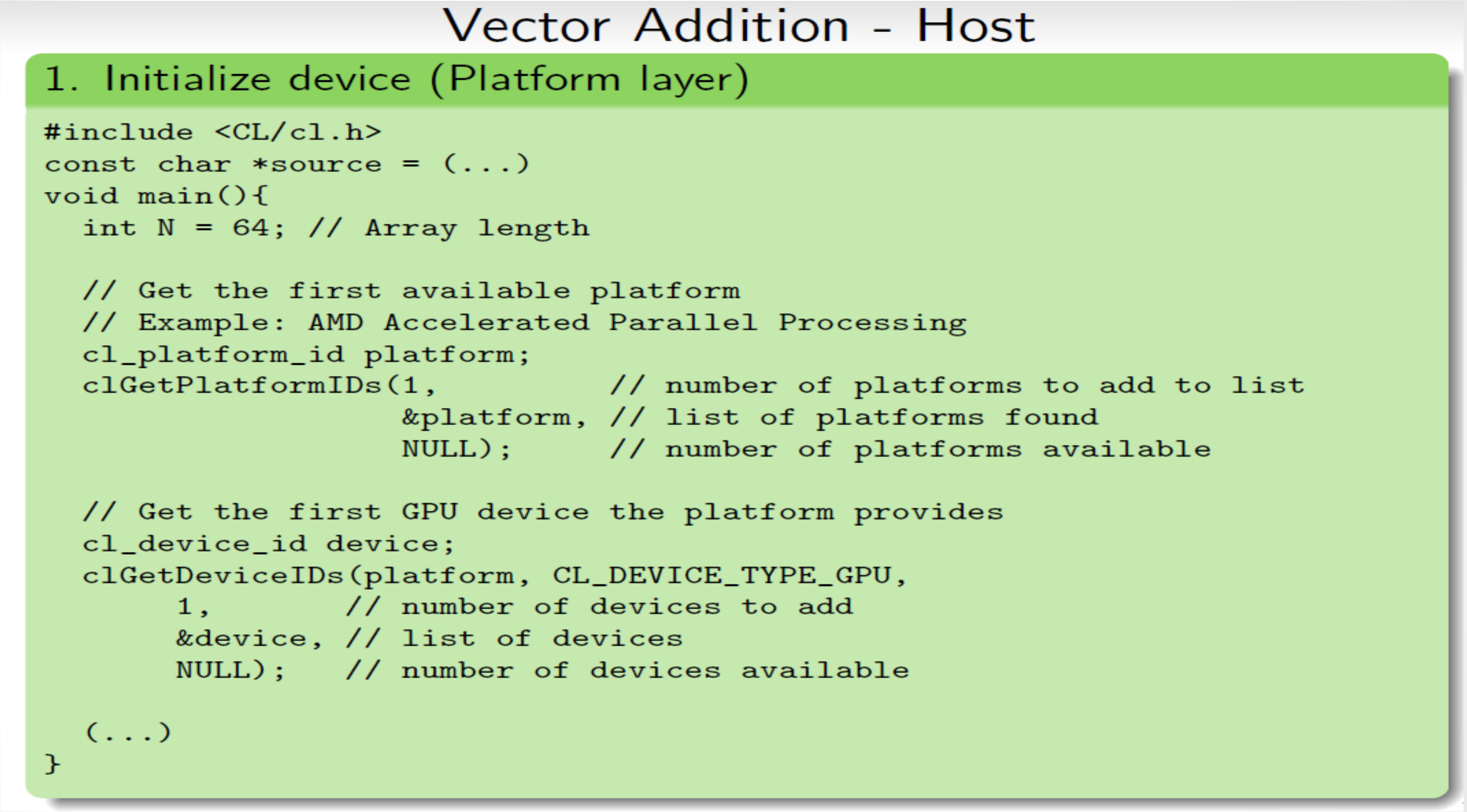
```
#include <CL/cl.h> // OpenCL header file

// OpenCL kernel source code included inline in host source code:
const char *source =
"__kernel void vec_add (__global const float *a,  \n"
"                       __global const float *b,  \n"
"                       __global       float *c)  \n"
"{                                                \n"
"  int gid = get_global_id(0);                    \n"
"  c[gid] = a[gid] + b[gid];                      \n"
"}                                                \n";

void main{}{
  (...)
}
```

Vector Addition in OpenCL

- Host program sets up the environment for the OpenCL program, creates and manages kernels
- 5 steps in a basic Host program
    1. **Initialize device (Platform layer)**
    2. Build program (Compiler)
    3. Create buffers (Runtime layer)
    4. Set arguments, enqueue kernel (Runtime layer)
    5. Read back results (Runtime layer)



Vector Addition - Host

1. Initialize device (Platform layer)

```
#include <CL/cl.h>
const char *source = (...)
void main(){
  int N = 64; // Array length

  // Get the first available platform
  // Example: AMD Accelerated Parallel Processing
  cl_platform_id platform;
  clGetPlatformIDs(1,         // number of platforms to add to list
                   &platform, // list of platforms found
                   NULL);     // number of platforms available

  // Get the first GPU device the platform provides
  cl_device_id device;
  clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU,
       1,       // number of devices to add
       &device, // list of devices
       NULL);   // number of devices available

  (...)
}
```

Vector Addition - Host

1. Initialize device (Platform layer)

```
void main(){
  (...)
  // Contexts are used by the runtime for managing program
  // objects, memory, and command queues

  // Create a context and command queue on that device
  cl_context context = clCreateContext(
       0,           // optional (context properties)
       1,           // number of devices
       &device,     // pointer to device list
       NULL, NULL, // optional (callback function for reporting errors)
       NULL);       // no error code returned

  cl_command_queue queue = clCreateCommandQueue(
       context,     // valid context
       device,      // device associated with context
       0,           // optional (command queue properties)
       NULL);       // no error code returned

  (...)
}
```

Vector Addition - Host

2. Build program (Compiler)

```
void main(){
  (...)

  // An OpenCL program is a set of OpenCL kernels and
  // auxiliary functions called by the kernels

  // Create program object and load source code into program object
  cl_program program = clCreateProgramWithSource(context,
                         1,        // number of strings
                         &source, // strings
                         NULL,    // string length or NULL terminated
                         NULL);   // no error code returned

  (...)
}
```

Vector Addition - Host

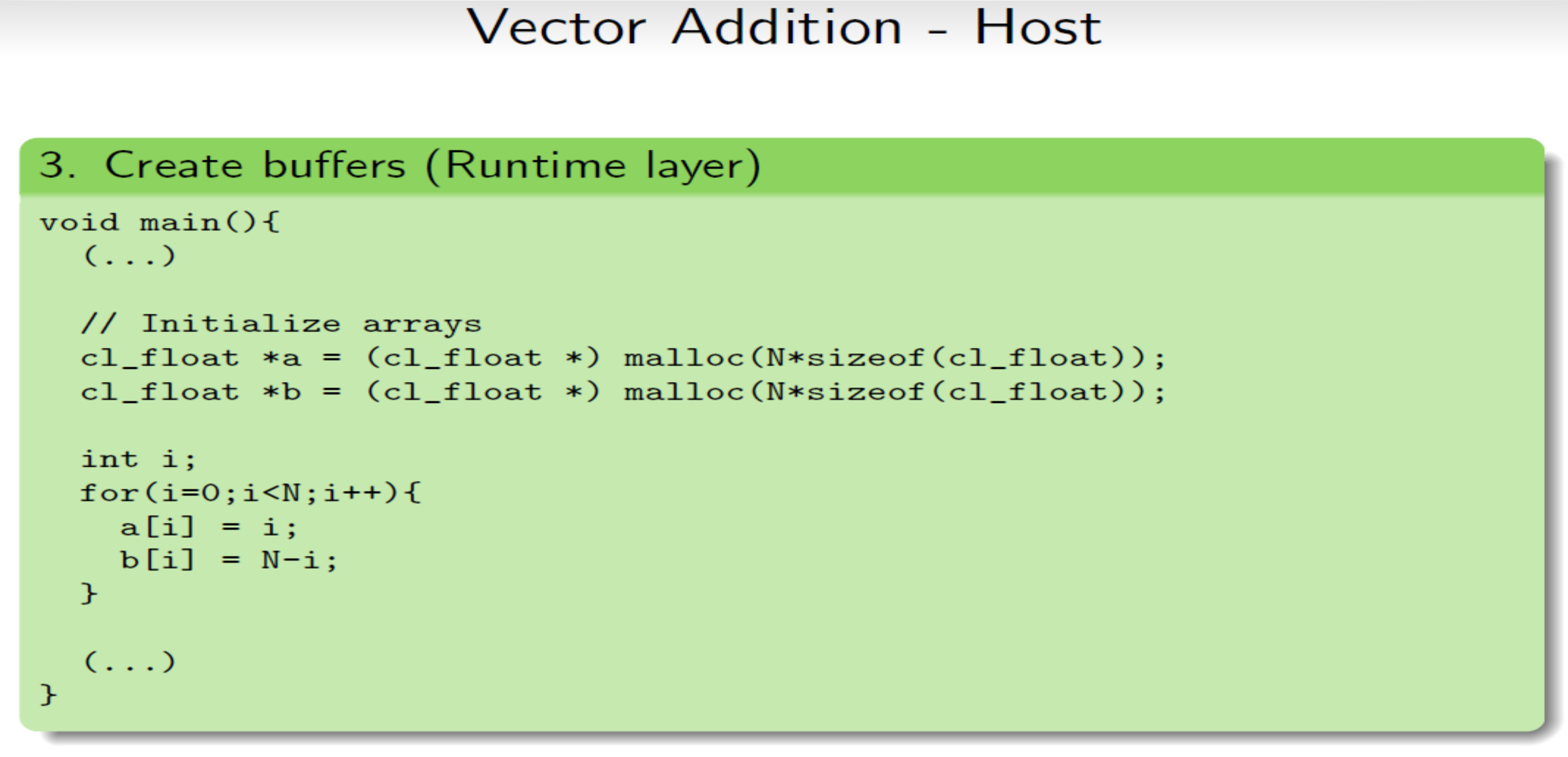
2. Build program (Compiler)

```
void main(){
  (...)

  // Build program executable from program source
  clBuildProgram(program,
        1,           // number of devices
        &device,     // pointer to device list
        NULL,        // optional (build options)
        NULL, NULL); // optional (callback function, argument)

  // Create kernel object
  cl_kernel kernel = clCreateKernel(
        program,     // program object
        "vec_add",   // kernel name in program
        NULL);       // no error code returned

  (...)
}
```



Vector Addition - Host

3. Create buffers (Runtime layer)

```
void main(){
  (...)

  // Initialize arrays
  cl_float *a = (cl_float *) malloc(N*sizeof(cl_float));
  cl_float *b = (cl_float *) malloc(N*sizeof(cl_float));

  int i;
  for(i=0;i<N;i++){
    a[i] = i;
    b[i] = N-i;
  }

  (...)
}
```

Vector Addition - Host

3. Create buffers (Runtime layer)

```
void main(){
  (...)

  // A buffer object is a handle to a region of memory

  cl_mem a_buffer = clCreateBuffer(context,
          CL_MEM_READ_ONLY |    // buffer object read only for kernel
          CL_MEM_COPY_HOST_PTR, // copy data from memory referenced
                                // by host pointer
          N*sizeof(cl_float),   // size in bytes of buffer object
          a,                    // host pointer
          NULL);                // no error code returned
  cl_mem b_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY |
                                   CL_MEM_COPY_HOST_PTR,
                                   N*sizeof(cl_float), b, NULL);
  cl_mem c_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                                   N*sizeof(cl_float), NULL, NULL);
  (...)
}
```

Vector Addition - Host

4. Set arguments, enqueue kernel (Runtime layer)

```
void main(){
  (...)
  size_t global_work_size = N;

  // Set the kernel arguments
  clSetKernelArg(kernel, 0, sizeof(a_buffer), (void*) &a_buffer);
  clSetKernelArg(kernel, 1, sizeof(b_buffer), (void*) &b_buffer);
  clSetKernelArg(kernel, 2, sizeof(c_buffer), (void*) &c_buffer);

  // Enqueue a command to execute the kernel on the GPU device
  clEnqueueNDRangeKernel(queue, kernel,
           1, NULL, // global work items dimensions and offset
           &global_work_size, // number of global work items
           NULL,    // number of work items in a work group
           0, NULL, // don't wait on any events to complete
           NULL);   // no event object returned

  (...)
}
```

Vector Addition - Host

5. Read back results (Runtime layer)

```
void main(){
  (...)

  // Block until all commands in command-queue have completed
  clFinish(queue);

  // Read back the results
  cl_float *c = (cl_float *) malloc(N*sizeof(cl_float));
  clEnqueueReadBuffer(
     queue,    // command queue in which read command will be queued
     c_buffer, // buffer object to read back
     CL_TRUE,  // blocking read - doesn’t return until buffer copied
     0,        // offset in bytes in buffer object to read from
     N * sizeof(cl_float), // size in bytes of data being read
     c,        // pointer to host memory where data is to be read into
     0, NULL,  // don’t wait on any events to complete
     NULL);    // no event object returned
}
```

Vector Addition - Host

Cleanup

```
void main(){
  (...)

  free(a);
  free(b);
  free(c);
  clReleaseMemObject(a_buffer);
  clReleaseMemObject(b_buffer);
  clReleaseMemObject(c_buffer);
  clReleaseKernel(kernel);
  clReleaseProgram(program);
  clReleaseContext(context);
  clReleaseCommandQueue(queue);
}
```

Optimization Strategies

- Expose data parallelism in algorithms
- Minimize host-device data transfer
- Overlap memory transfer with computation
- Prevent path divergence between work items
- Number of work items per work group should be a multiple of the wavefront size (64 for AMD Radeon HD 6970)
- Use local memory as a cache
- Others: memory coalescing, bank conflicts, OpenCL C vector data types..

OpenCL Resources

- Khronos OpenCL specification, reference card, tutorials, etc: http://www.khronos.org/opencl
- AMD OpenCL Resources: http://developer.amd.com/opencl
- NVIDIA OpenCL Resources: http://developer.nvidia.com/opencl
- MacResearch: 6 OpenCL tutorials: http://www.macresearch.org/opencl-tutorials
- June 2011 Cern Computing Seminar: http://indico.cern.ch/conferenceDisplay.py?confId=138427