More on OpenCL

Synchronization

- Local Memory
  - In a work-group it's not pre-determined when each work-item will execute its instructions
  - Consequently, almost always need work-item synchronization to ensure correct use of local memory.
  - Instruction
    - **barrier(CLK_LOCAL_MEM_FENCE);**
  - inserts a "barrier"; no work-item (within the same work-group) is allowed to proceed beyond this point until the rest have reached it

Synchronization

- Already introduced barrier(); which forms a barrier – all threads wait until every one has reached this point.
- Use CLK LOCAL MEM FENCE and CLK GLOBAL MEM FENCE to ensure order of local/global memory read/writes resp.
- When writing conditional code, must be careful to make sure that all threads do reach the barrier();
- Otherwise, can end up in *deadlock*

Typical Application

```
// load in data to shared memory
   ...
   ...
   ...

// synchronisation to ensure this has finished

  barrier(CLK_LOCAL_MEM_FENCE |
          CLK_GLOBAL_MEM_FENCE);

// now do computation using shared data
   ...
   ...
   ...
```

Atomic Operations

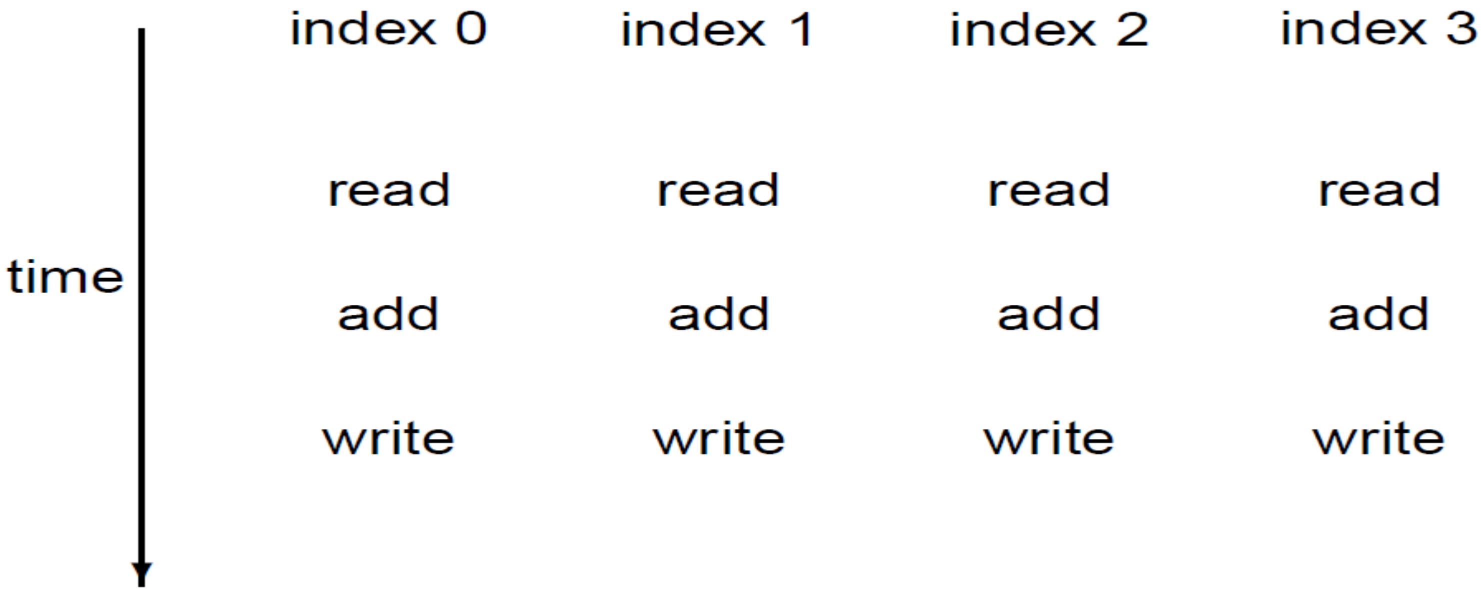
- Occasionally, an application needs work-items to update a counter in local memory.

```
__local int count;

...

if ( ... ) count++;
```

- In this case, there is a problem if two (or more) work-items try to do it at the same time

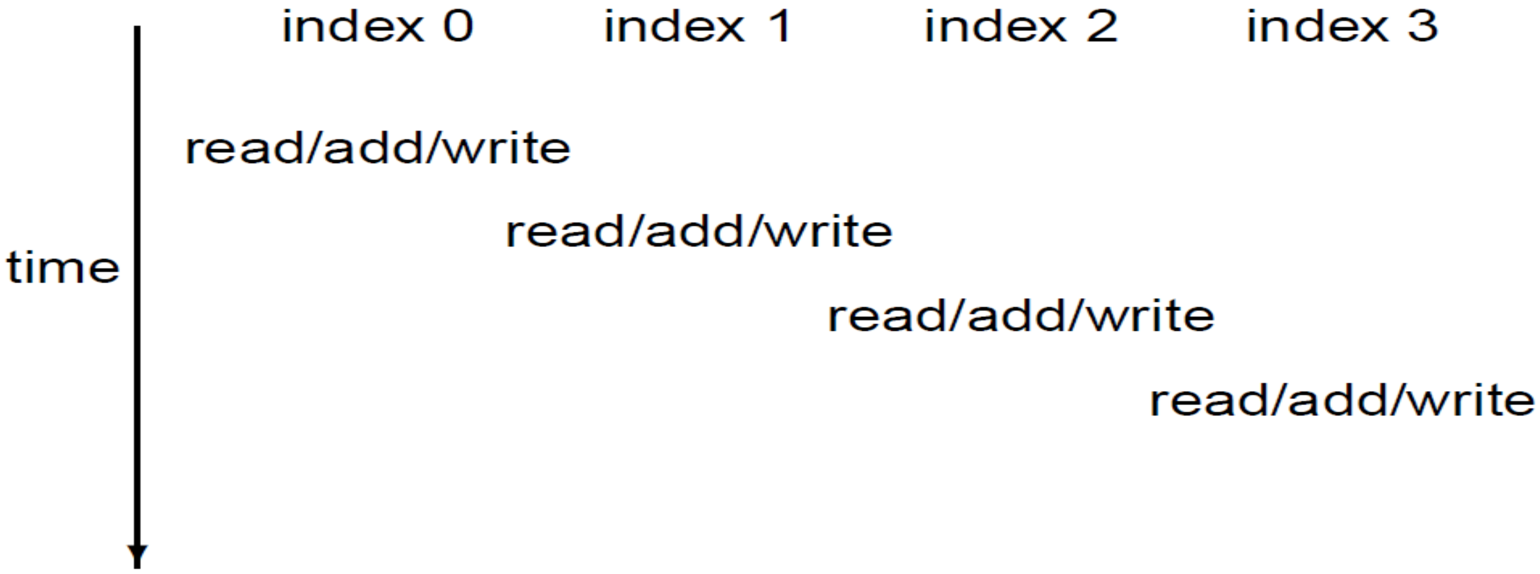
Atomic Operations

Using standard instructions, multiple work-items in the same work-group will only update it once.



Atomic Operations

With atomic instructions, the read/add/write becomes a single operation, and they happen one after the other



Atomic Operations

- Several different atomic operations are supported, almost all only for integers:
  - addition (integers and 32-bit floats)
  - minimum / maximum
  - increment / decrement
  - exchange / compare-and-swap
  - bitwise **AND** OR **XOR**
- These are
  - quite fast for data in local memory
  - slower for data in global memory
  - (better on new Kepler hardware)

Atomic Operations

- Compare-and-swap:

```
int atomic_cmpxchg(volatile __local int *p,
                                 int cmp, int val);
```

- if compare equals **old** value stored at address then **val** is stored instead
- in either case, routine returns the value of **old**
- seems a bizarre routine at first sight, but can be very useful for atomic locks
- also can be used to implement 64-bit floating point atomic addition

Global atomic lock

```
// global variable: 0 unlocked, 1 locked
__global volatile int lock=0;

__kernel void kernel(...) {
  ...

  if (get_local_id(0)==0) {
    // set lock
    do {} while(atomic_cmpxchg(&lock, 0, 1));
  }

  ...

  // free lock
  lock = 0;
}
```

Global atomic lock

- **Problem:** when a work-item writes data to global memory the order of completion is not guaranteed, so global writes may not have completed by the time the lock is unlocked

```
__kernel void kernel(...) {
  ...

  if (get_local_id(0)==0) {
    do {} while(atomic_cmpxchg(&lock,0,1));
    ...
    mem_fence(CLK_GLOBAL_MEM_FENCE); // order write

    // free lock
    lock = 0;
  }
}
```

Mem_fence

- mem fence();
  - order all preceding global or local (or both) reads and writes
    - means all loads/stores committed to memory before any following loads/stores
- mem fence write();
  - same as above, but only for stores
- mem fence read();
  - same as above, but only for loads
- Different to barrier() – non-blocking

Some Applications:

REDUCTION AND SCAN OPERATIONS

Reduction

- The most common reduction operation is computing the sum of a large array of values:
  - averaging in Monte Carlo simulation
  - computing RMS change in finite difference computation or an iterative solver
  - computing a vector dot product in a CG or GMRES iteration

Reduction

Other common reduction operations are to compute a minimum or maximum.

Key requirements for a reduction operator $\circ$ are:

- commutative: $a \circ b = b \circ a$
- associative: $a \circ (b \circ c) = (a \circ b) \circ c$

Together, they mean that the elements can be re-arranged and combined in any order.

(Note: in MPI there are special routines to perform reductions over distributed arrays.)

Approach

- Will describe things for a summation reduction – the extension to other reductions is obvious
- Assuming each thread starts with one value, the approach is to
  - first add the values within each thread block, to form a partial sum
  - then add together the partial sums from all of the blocks

Local reduction

The first phase is contructing a partial sum of the values within a work-group.

Question 1: where is the parallelism?

“Standard” summation uses an accumulator, adding one value at a time $\Longrightarrow$ sequential

Parallel summation of N values:

- first sum them in pairs to get $N/2$ values
- repeat the procedure until we have only one value

Local Reduction

Question 2: any problems with work-item divergence?

Note that not all work-items can be busy all of the time:

- $N/2$ operations in first phase
- $N/4$ in second
- $N/8$ in third
- etc.

For efficiency, we want to make sure that each processing element is either fully active or fully inactive, as far as possible.

Local Reduction

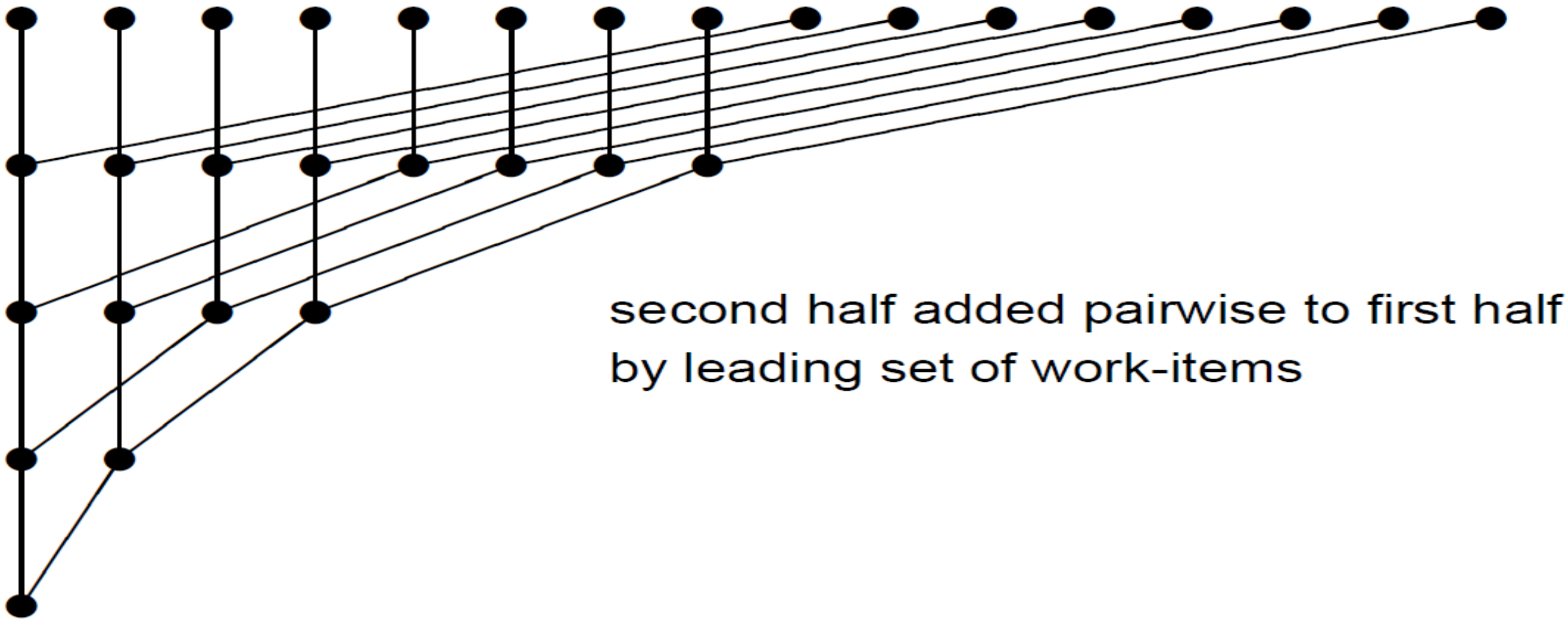
Question 3: where should data be held?

Work-items need to access results produced by other work-items:

- global device arrays would be too slow, so use local memory
- need to think about synchronisation

Local Reduction

Pictorial representation of the algorithm:



Local Reduction

```
__kernel void sum(__global float *d_sum,
                  __global float *d_data,
                  __local float *temp) {
 int tid = get_local_id(0);

 temp[tid] = d_data[get_global_id(0)];

 for (int d=get_local_size(0)>>1; d>=1; d>>=1) {
   barrier(CLK_LOCAL_MEM_FENCE)
   if (tid<d) temp[tid] += temp[tid+d];
 }

 if (tid==0) d_sum[get_group_id(0)] = temp[0];
}
```

Local Reduction

Note:

- use of dynamic local memory – size has to be declared when setting kernel argument
- use of `barrier(CLK_LOCAL_MEM_FENCE)` to make sure previous operations have completed
- first work-item outputs final partial sum into specific place for that work-group

Scan Operation

Given an input vector $u_i,\ i=0,\ldots,I\!-\!1$, the objective of a scan operation is to compute

$$v_j = \sum_{i<j} u_i \quad \text{for all} \quad j<I.$$

Why is this important?

- a key part of many sorting routines
- arises also in particle filter methods in statistics
- related to solving long recurrence equations:

$$v_{n+1} = (1\!-\!\lambda_n) v_n + \lambda_n u_n$$

- a good example that looks impossible to parallelise

Scan Operation

Before explaining the algorithm, here's the "punch line":

- some parallel algorithms are tricky – don't expect them all to be obvious
- check the OpenCL examples in the CUDA SDK, check the literature using Google – don't put lots of effort into re-inventing the wheel
- the relevant literature may be 20–25 years old – back to the glory days of CRAY vector computing and Thinking Machines' massively-parallel CM5

Scan Operations

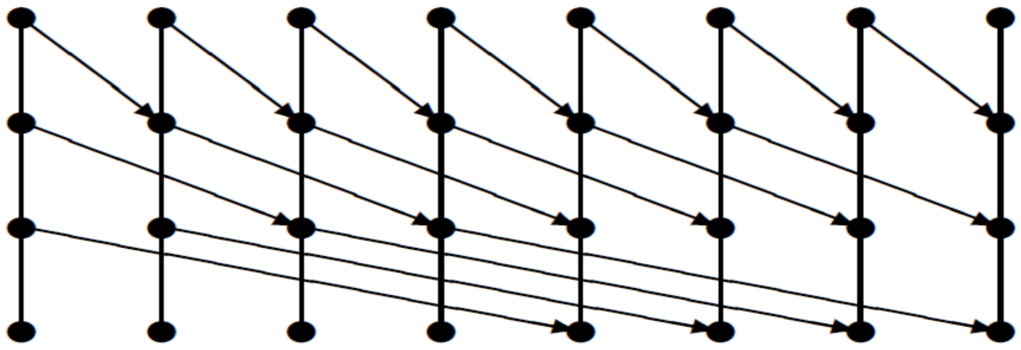
Similar to the global reduction, the top-level strategy is

- perform local scan within each work-group
- add on sum of all preceding work-groups

Will describe two approaches to the local scan, both similar to the local reduction but in slightly different ways

- first approach:
  - very simple but $O(N \log N)$ operations
- second approach:
  - similar to binary tree summation but with both downward and upward passes
  - $O(N)$ operations so slightly more efficient

Local Scan – Version 1



- after n passes, each sum has local plus preceding $2^n - 1$ values
- $\log_2 N$ passes, and $O(N)$ operations per pass
  $\implies \quad O(N \log N)$ operations in total

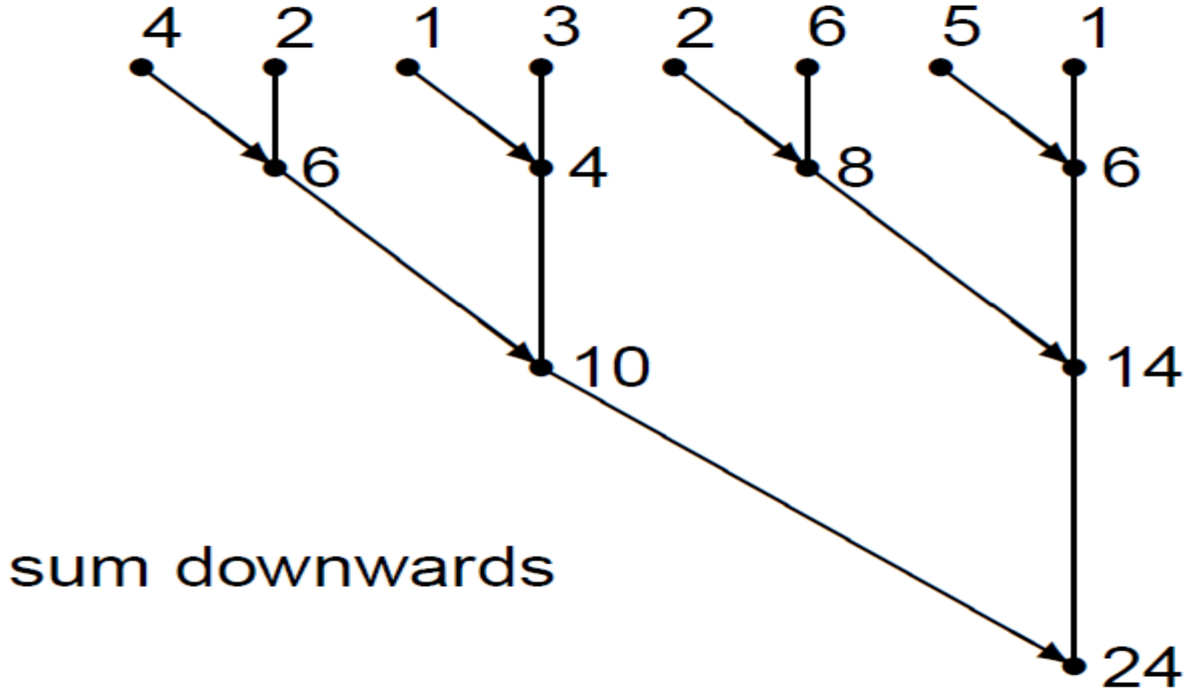
Local Scan – Version 1

```
__kernel void scan(__global float *d_sum,
                   __global float *d_data,
                   __local float* temp)
{
 int tid = get_local_id(0);
 temp[tid] = d_data[get_global_id(0)];

 for (int d=1; d<get_local_size(0); d<<=1) {
   barrier(CLK_LOCAL_MEM_FENCE);
   float temp2 = (tid >= d) ? temp[tid-d] : 0;
   barrier(CLK_LOCAL_MEM_FENCE);
   temp[tid] += temp2;
 }
 ...
}
```

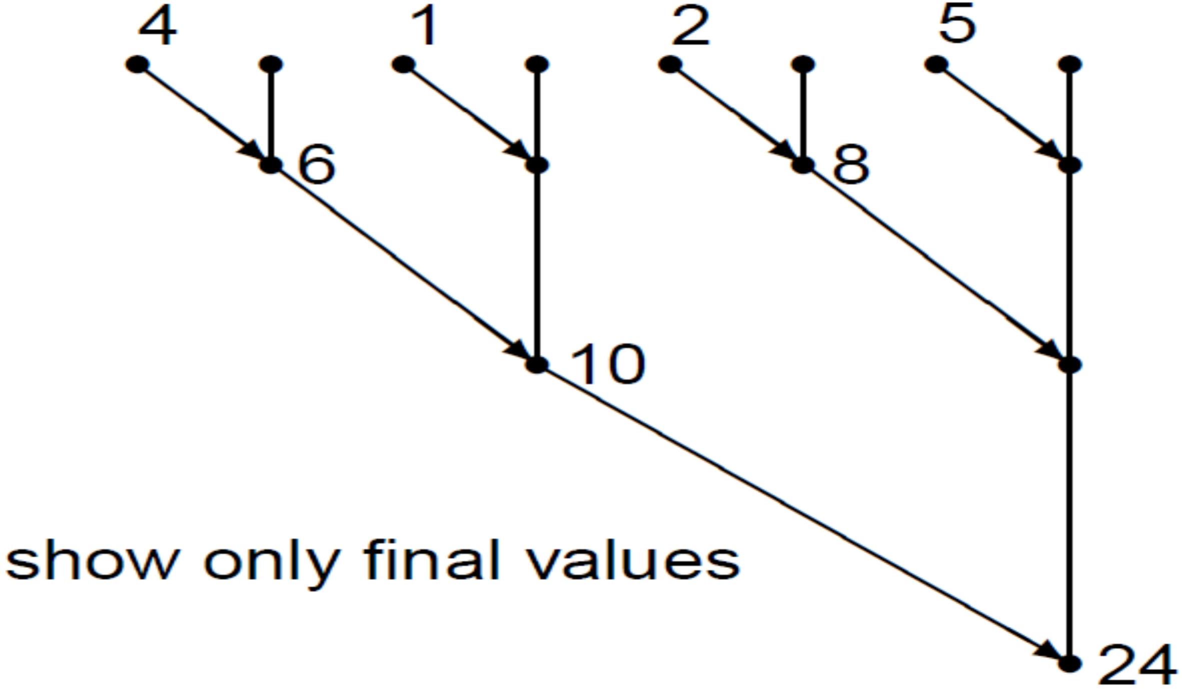
Local Scan – Version 1

- Notes
  - much simpler than version 2
  - at most only 40% slower
  - increment is set to zero if no element to the left
  - both `barrier()` points are needed

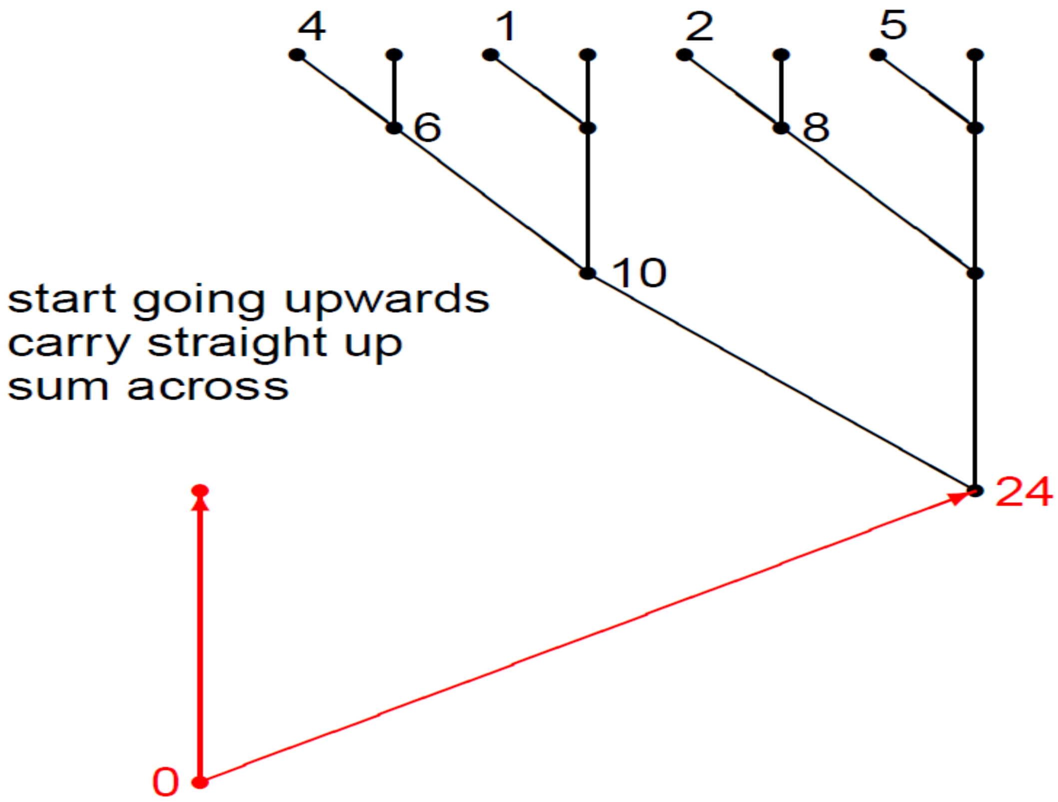
Local Scan – Version 2



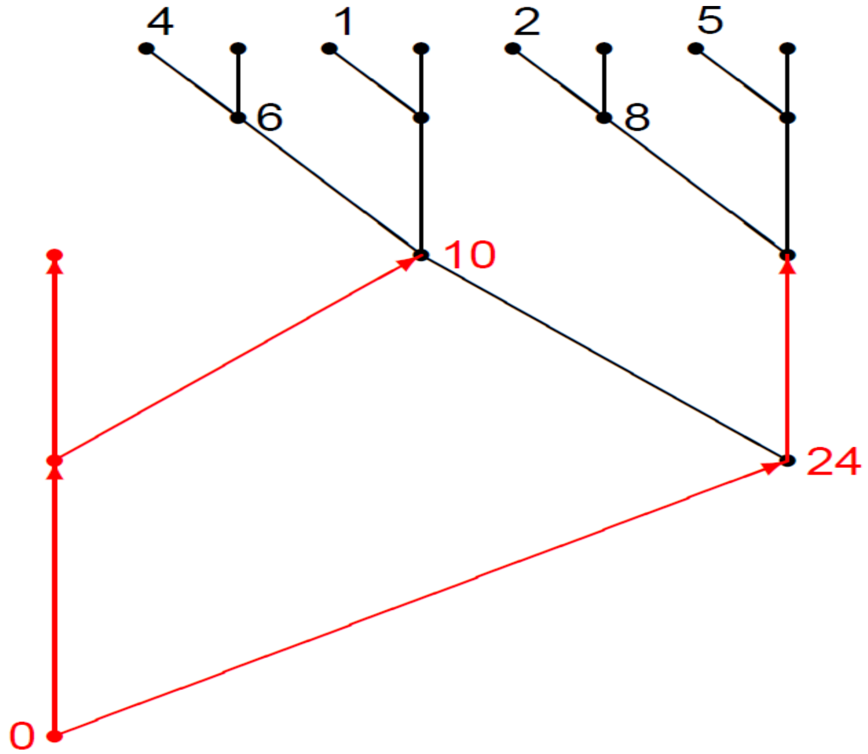
Local Scan – Version 2



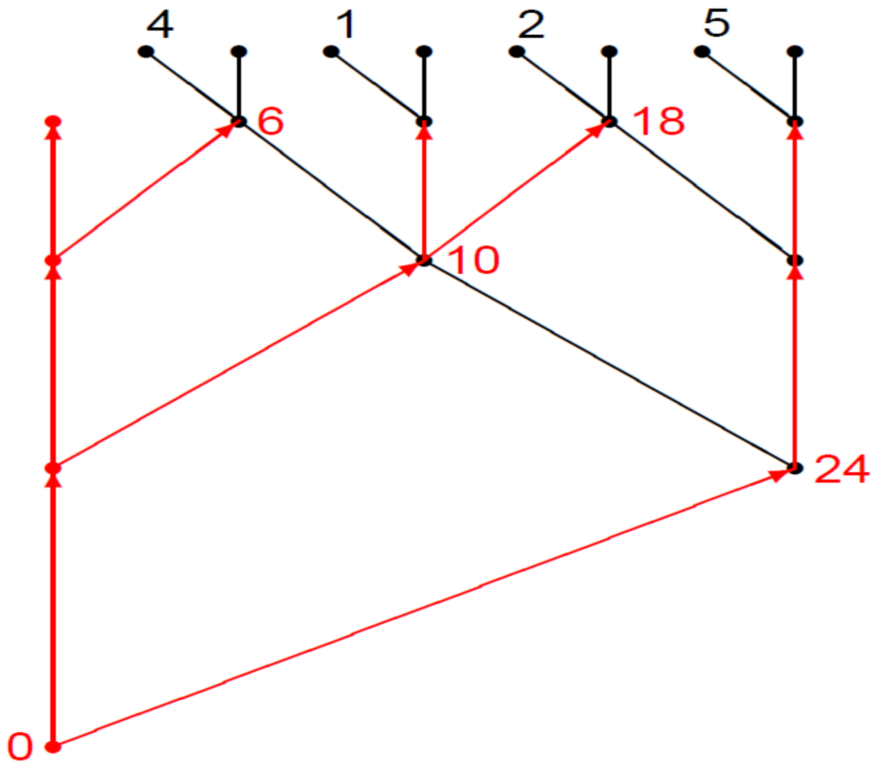
Local Scan – Version 2



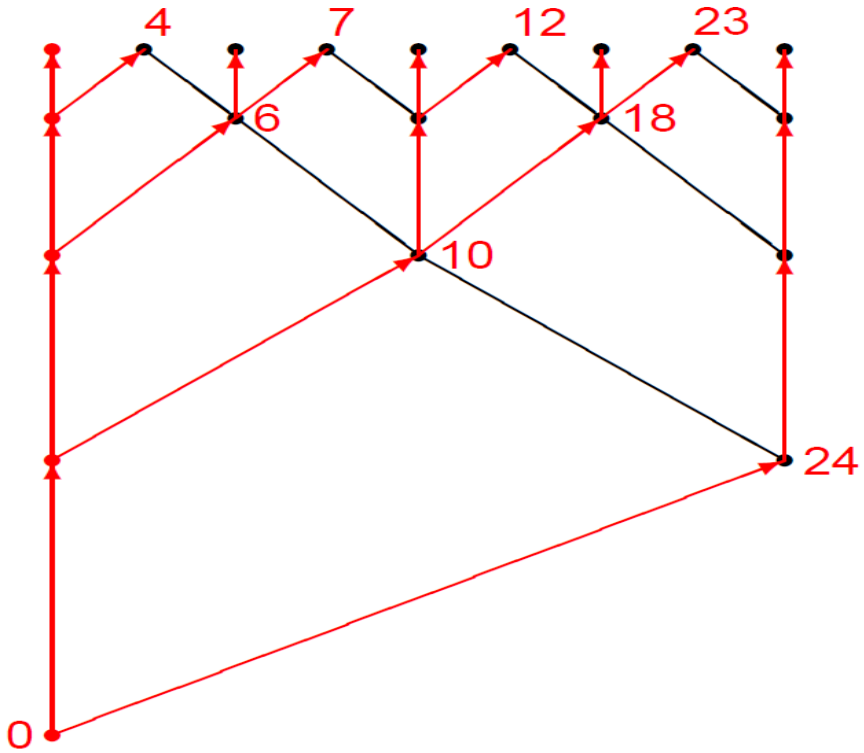
Local Scan – Version 2



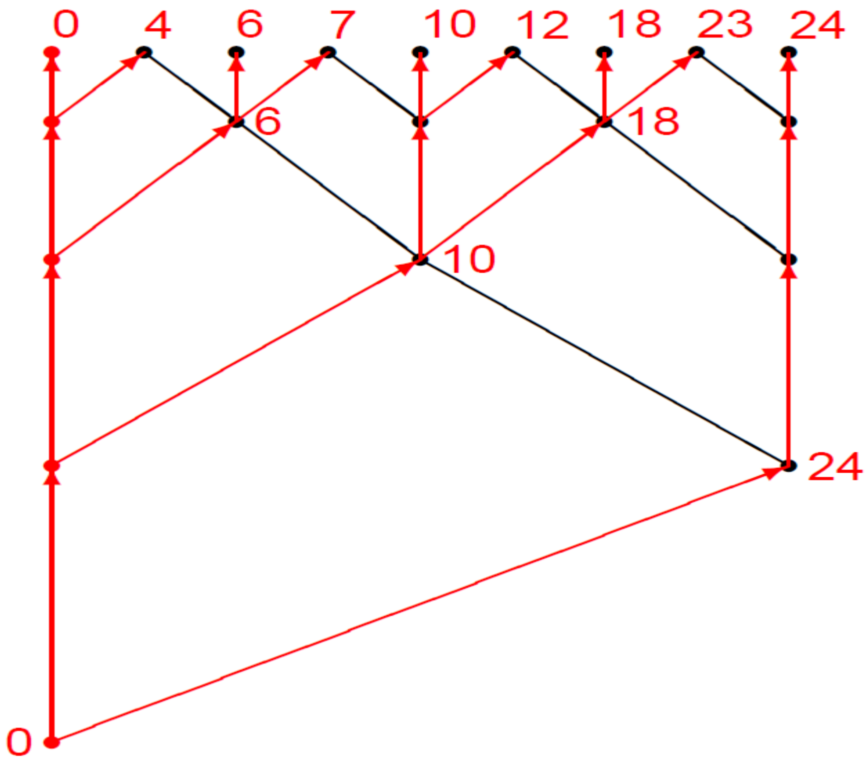
Local Scan – Version 2



Local Scan – Version 2



Local Scan – Version 2



Local Scan – Version 2

- Notes
  - not very easy to follow, maybe best to go through the example above to check it's doing the right thing
  - in the practical, the code puts the local scan values back in the global device array
  - however, really we need to complete the process by performing a global scan at the higher level

Questions?????