XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 17: Synchronisation

Previous lectures

Many of the previous lectures have mentioned **parallel synchronisation** in some form. However, there are many ways to synchronise:

- **Locks** in shared memory systems *[Lectures 6 and 7]*.
- **Synchronisation barrier** at each level of a binary tree **reduction** *[Lecture 11]*.
- **Blocking communication**, which affords a form of synchronisation, in distributed memory systems *[Lecture 9]*.
- . . .

Also recall that GPU's have multiple memory types, some of which can be viewed as *shared* (`__global`), and some which can be viewed as *distributed* (`__local`) *[Lecture 16]*.

This lecture

In this lecture we will look at **synchronisation** on a GPU:

- How to synchronise **within** a work group.
- How to synchronise **between** work groups.

We will also see how the SIMD cores can potentially **reduce** or **improve** performance:

- Threads within a **subgroup** are **automatically synchronised**.
- Threads performing different calculations can lead to **divergence** and reduced performance.

Reminder: Scalar product

As an example, we will use the **scalar product** between two n-vectors **a** and **b**, as in Lecture 11.

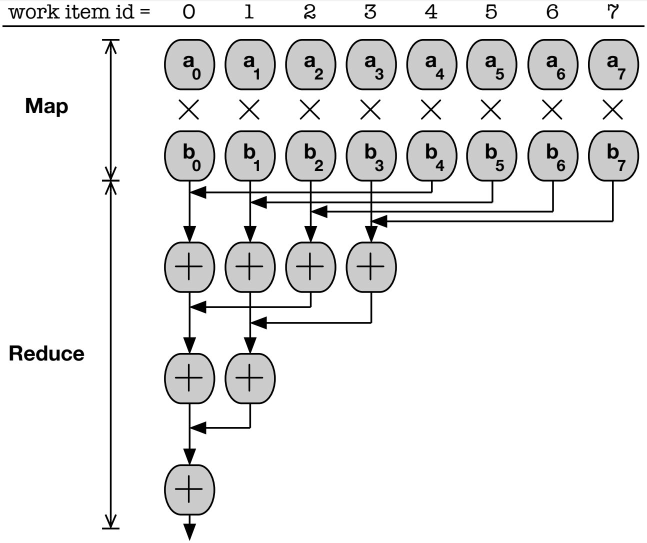
Written mathematically as *(indexing starting from 1)*:

$$\mathbf{a}\cdot\mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \ldots + a_n b_n$$

In serial CPU code *(indexing starting from 0)*:

```
float dot = 0.0f;
for( i=0; i<n; i++ ) dot += a[i] * b[i];
```

MapReduce pattern for $n = 8$



Reduction in local memory

First consider *n* equal to or less than the **work group size**.

Use **local memory** for the intermediate quantities *[Lecture 16]*.

- **Faster** than global memory; allows **communication** within a work group.

Each work item copies `a[i]*b[i]` to local memory first.

- Reduce using the binary tree pattern on the previous slide[1].
- Each addition performed by the work item with the *lower* i.d.
- Final result in work item with i.d. = 0 copied to the answer (in global memory).

[1]Divide-by-two implemented by **compound bitwise right shift** operator '>>='.

Kernel code

Code on Minerva: `workGroupReduction.c`, `workGroupReduction.cl`, `helper.h`

```
__kernel
void reduceNoSync( __global float *device_a, __global
    float *device_b, __global float *dot, __local
    float *scratch )
{
  int stride,
      id        = get_local_id  (0),
      groupSize = get_local_size(0);  //=work group

  scratch[id] = device_a[id] * device_b[id];

  for( stride=groupSize/2; stride>0; stride>>=1 )
    if( id < stride )
      scratch[id] += scratch[id+stride];

  if(id==0) *dot = scratch[0];
}
```

Calling C-code

```c
// float array of size 1 on device.
cl_mem device_dot = clCreateBuffer(...);

... // Set kernel arguments 0, 1 and 2.
clSetKernelArg(kernel,3,N*sizeof(float),NULL);
  // NULL => __local memory of given size.

// Add to the command queue.
size_t indexSpaceSize[1]={N}, workGroupSize[1]={N};
clEnqueueNDRangeKernel(queue,kernel,1,NULL,
    indexSpaceSize,workGroupSize,0,NULL,NULL);

// Get the result back to host float ‘dot’.
float dot;
clEnqueueReadBuffer(queue,device_dot,CL_TRUE,0,sizeof(
    float),&dot,0,NULL,NULL);
```

Barriers

Without synchronisation, this reduction is *not* guaranteed to work on *all* systems.

Recall that **barriers** are points in code that no processing unit can leave until **all** units reach it *[c.f. Lecture 11]*.

- `#pragma omp barrier` in OpenMP.
- `MPI_Barrier()` in MPI.

In OpenCL[1], a barrier **within a work group** is implemented as:

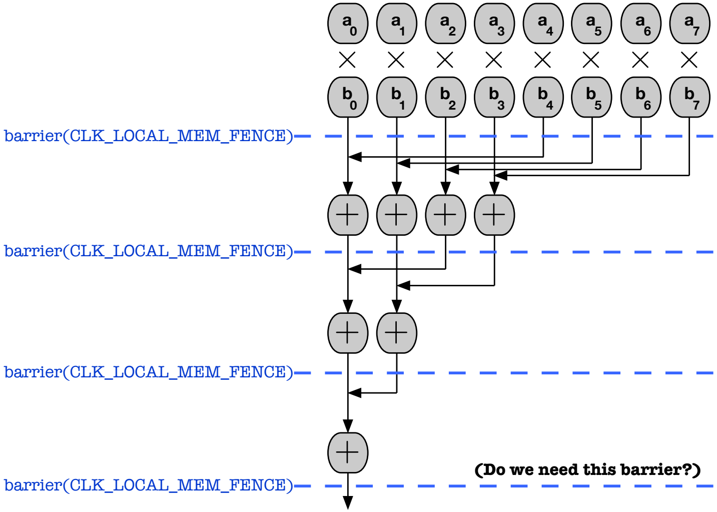
```
barrier(CLK_LOCAL_MEM_FENCE);
```

[1]In CUDA: `__syncthreads()` synchronises within a **thread block**=work group.

Reduction with synchronisation

```
void reduceWithSync(...)     // Same arguments.
{
  int id=..., groupSize=..., stride; // As before.

  scratch[id] = device_a[id] * device_b[id];
  barrier(CLK_LOCAL_MEM_FENCE);        // Sync.

  for( stride=groupSize/2; stride>0; stride>>=1 )
  {
    if( id < stride )
      scratch[id] += scratch[id+stride];

    barrier(CLK_LOCAL_MEM_FENCE);      // Sync.
  }

  if(id==0) *dot = scratch[0];
}
```

Reduction with `barrier(CLK_LOCAL_MEM_FENCE)`


a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7
barrier(CLK_LOCAL_MEM_FENCE)
barrier(CLK_LOCAL_MEM_FENCE)
barrier(CLK_LOCAL_MEM_FENCE)
barrier(CLK_LOCAL_MEM_FENCE)
(Do we need this barrier?)

Problems larger than a single work group?

If we could synchronise **between** work groups, could use the same method as before:

1. Make device vectors and `scratch` **global**.
2. Replace **local** barriers with **global** barriers.

However, **no such global barrier exists**[1].

GPUs cannot synchronise between work groups/thread blocks[2]

[1]`barrier(CLK_GLOBAL_MEM_FENCE)` *does* exist, but refers to *accesses* to global memory; it still only synchronises *within* a work group.
[2]Some modern GPUs support **cooperative groups** that allow synchronisation across multiple thread blocks; *e.g.* CUDA 9.0.

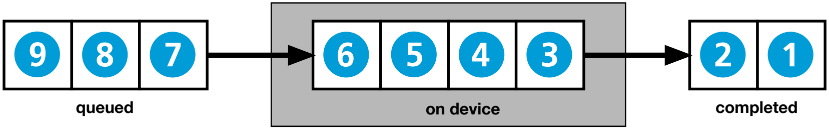
Warning!

You might see claims that it is possible to synchronise globally on any GPU by constantly **polling** a global memory location.

- *i.e.* work items constantly read/write to synchronise.

This may work, but **only for small problems.**

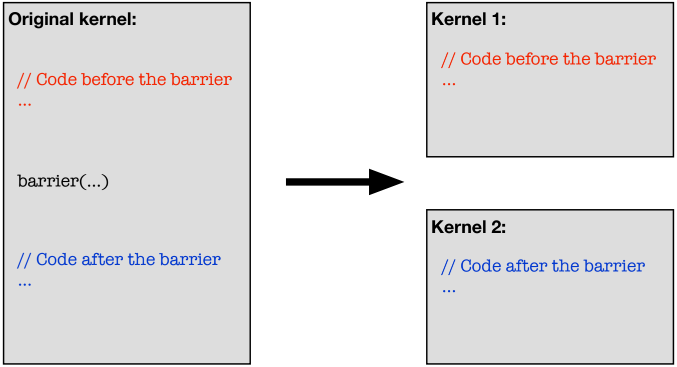
If there are too many work groups for the device, it **queues** them:



If they are not all on the device at the same time, it is **impossible to synchronise <u>within</u> one kernel** using this method.

Solution: Multiple kernels

The solution is to break the kernel at the barrier point into **multiple kernels** called consecutively:


Original kernel:
// Code before the barrier
...
barrier(...)
// Code after the barrier
...
Kernel 1:
// Code before the barrier
...
Kernel 2:
// Code after the barrier
...

This way kernel 1 completes before kernel 2 starts.

Reduction across work groups

It is possible to use this method for reduction[1]:

1. Repeatedly call kernel that reduces an array of **partial sums** until less than maximum work group size.
2. Final kernel call to reduce these partial sums.

It is simpler (although less efficient) to use the CPU:

1. Each work group inserts its partial sum into a global array.
2. Final summation performed **on the host**.

This is conceptually similar to an MPI program performing final calculations on rank 0.

[1]Wilt, *The CUDA handbook* (Addison-Wesley, 2013).

Subgroups (warp, wavefront, *etc.*)

Recall that GPUs are based on SIMD cores.

- Each core contains **multiple hardware threads** that perform the **same operation**.

In OpenCL, the number of **work items** (*i.e.* **threads**) simultaneously on a single SIMD core is known as a **subgroup**.

- **Smaller** than a work group.

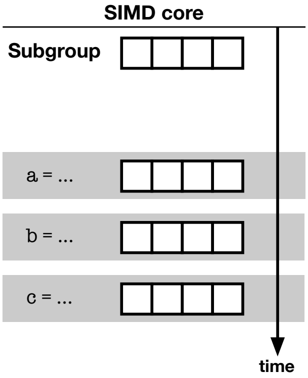
The actual size is vendor specific. For example:

- Nvidia call them **warps**, each of which has 32 threads.
- AMD have 64-thread **wavefronts**.

Lockstep

The SIMD core applies the **same** operation to all items in the subgroup **simultaneously**. We say it advances in **lockstep**.

```
__kernel
void kernel(...)
{
  int id = get_global_id(0);
  float a, b, c;

  a = 4*array[id];

  b = a*a;

  c = b + a;
}
```



Reduction with a subgroup

For reduction, this means that once the problem has been reduced to the size of a subgroup, there is **no longer any need for explicit synchronisation**[1]:

```
__kernel
void reduce(...)
{
  ...  // Start as before.

  // Split the loop into two.
  for(stride=group/2;stride>subgroup;stride>>=1) {
    if(id<stride) scratch[id] += scratch[id+stride];
    barrier(CLK_LOCAL_MEM_FENCE);      // Sync.
  }
  // See next slide ...
```

[1]Wilt, *The CUDA handbook* (Addison-Wesley, 2013).

Final reduction

For the final reduction, exploit **lockstepping** by removing the **synchronisation**:

```
for(;stride>0;stride>>=1)
{
  if(id<stride)
    scratch[id] += scratch[id+stride];

  // No barrier().
}
```

This avoids any overheads with calling `barrier()` (*i.e.* unnecessarily checking if all threads have reached this point, when we know they must have).

Divergence

Recall that SIMD cores can also be termed **SIMT** *[Lecture 14]*:

- <u>S</u>ingle <u>I</u>nstruction stream, <u>M</u>ultiple **<u>T</u>hreads**.

This means it *is* possible to perform different operations with a subgroup, but they **remain in lockstep**.

- Only **one** distinct operation can be performed **at a time**.

This can lead to **serialisation** where operations are performed **one after the other**.

- Can lead to a severe performance penalty.

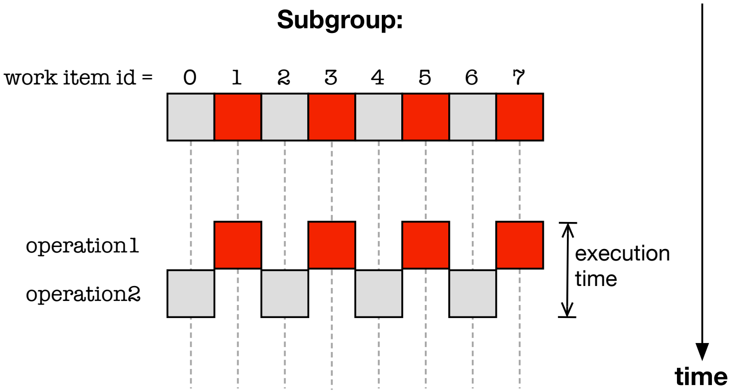
Code that leads to divergence

Suppose we want even-numbered work items to perform one operation, and odd-numbered items a different one[1]:

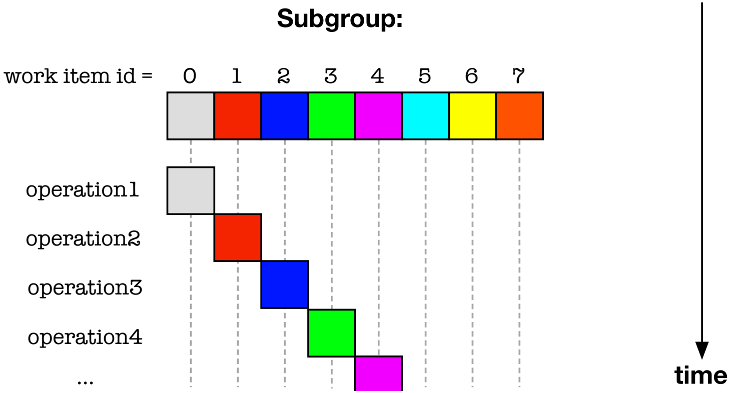
```
__kernel
void kernel(...)
{
  int id = get_global_id(0);

  if( id%2 )
    operation1;       // id odd.
  else
    operation2;       // id even.
}
```

[1]Recall `i%2==1` for `i` odd, `0` for `i` even.

In this example the execution time is **double** what was expected:



For more operations the execution time increases further, *e.g.* a switch-case clause where every thread performs a different operation.



This is true **serialisation**.

Summary and next lecture

Today we have looked at **synchronisation**, focussing on GPUs.

- **Barriers** can synchronise within a work group.
- **Cannot** synchronise between work groups within a kernel - must split into separate kernels.
- Work items of threads within a **subgroups** execute in lockstep.
    - No need for explicit synchronisation mechanisms.
    - Can lead to **divergence** and reduced performance.

Next time we will look at **atomic** instructions, continuing what we started in Lecture 6.