XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 19: Task parallelism

Previous lectures

For much of this module have considered **loop parallel** problems.

- Same operation applied to multiple data sets.

In Lecture 13 we looked at a **work pool** in MPI:

- One processing unit, the **main process**, sent small independent **tasks** to all other units.
- These **worker** processes performed the task, and sent the result back to the main process.

We referred to this as **task parallelism** since the emphasis was on parallelising **tasks** rather than the data.

Today's lecture

Today we will look at task parallelism in more detail

- How GPU **command queues** or **streams** can permit:
  - Overlapping device and host computation.
  - Overlapping host-device transfer and device computation.
- How OpenCL's **events** specify **dependencies**.
- **Task graphs** that are derived from these inter-dependencies.
- The **work-span model** that estimates the maximum **speed up** from a task graph.

Firstly, we will see how to time an OpenCL program using an **event**.

Timing kernels in OpenCL

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

For **profiling** purposes we often want to **time** how long a kernel takes to complete, to see if modifications can make it faster.

Timing in OpenCL is straightforward to achieve using **events**.

1. Ensure the **command queue** supports profiling.
2. Declare an **event**, and attach to the kernel when it is enqueued.
3. Extract the time taken **once the kernel has finished**.

Some of previous code examples already do this.

```c
// Ensure queue supports profiling.
cl_command_queue queue = clCreateCommandQueue
(context,device,CL_QUEUE_PROFILING_ENABLE,&status);

// OpenCL event.
cl_event timer;

// Enqueue the kernel with timer as last argument.
status = clEnqueueNDRangeKernel(queue,...,&timer);

// ... (once the kernel has finished)
cl_ulong start, end;
clGetEventProfilingInfo(timer,
    CL_PROFILING_COMMAND_START,sizeof(cl_ulong),&start
    ,NULL);
clGetEventProfilingInfo(timer,CL_PROFILING_COMMAND_END
    ,sizeof(cl_ulong),&end,NULL);
printf("Time: %g ms\n",1e-6*(cl_double)(end-start));
```

Blocking communication

Recall that when we copy the data from device to host at the end of the calculation, we typically use a **blocking** call:


```
clEnqueueReadBuffer(queue,device_dot,CL_TRUE,...);
```

- The `CL_TRUE` denotes **blocking**; the routine does not return until the copy is complete.
- Similar to `MPI_Recv()` *[cf. Lecture 9]*.

Replacing `CL_TRUE` with `CL_FALSE` makes this a **non-blocking** copy command[1]:

- Will return 'immediately,' **before** the copy is complete.
- Similar to `MPI_Irecv()` *[cf. Lecture 12]*.

[1] In CUDA: Use `cudaMemcpyAsync()` rather than `cudaMemcpy()`.

Potential consequences of non-blocking

For this example, using a non-blocking copy can mean:

1. The check on the host **fails** — since check code reached before the data has been copied to the host.
2. The **timing** is meaningless — since kernel not yet finished.

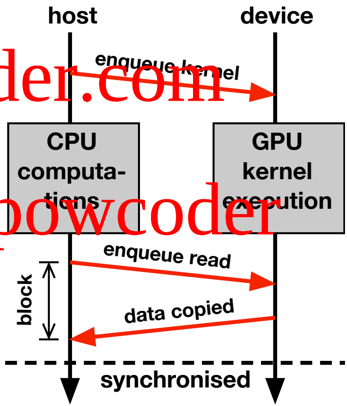
Note that the read did not start until the kernel had finished.

- It was enqueued on the **same command queue**.

Overlapping host and device computation

The queuing model means we can perform calculations on the host (CPU) and device (GPU) **simultaneously**.

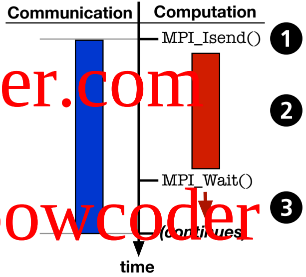

```
// Enqueue kernel.
clEnqueueNDRangeKernel(...);

// Perform useful operations
// on the host.
...

// Blocking copy device->host
clEnqueueReadBuffer(...);

// Device and host in sync.
```



Overlapping computation with communication

Recall from Lecture 12 that we can reduce **latency** by overlapping **computation** with **communication**.

1. **Start** communication with `MPI_Isend()`/`MPI_Irecv()`.
2. Perform **calculations**.
3. **Synchronise** using `MPI_Wait()`.



Similar benefits can be achieved on a GPU using multiple **command queues** (OpenCL) / **streams** (CUDA).

Multiple command queues: Example

Consider the following problem.

- Have two data arrays `a` and `b`.
- Needs to execute `kernelA` on `a`, and `kernelB` on `b`.
- These calculations are **independent**.

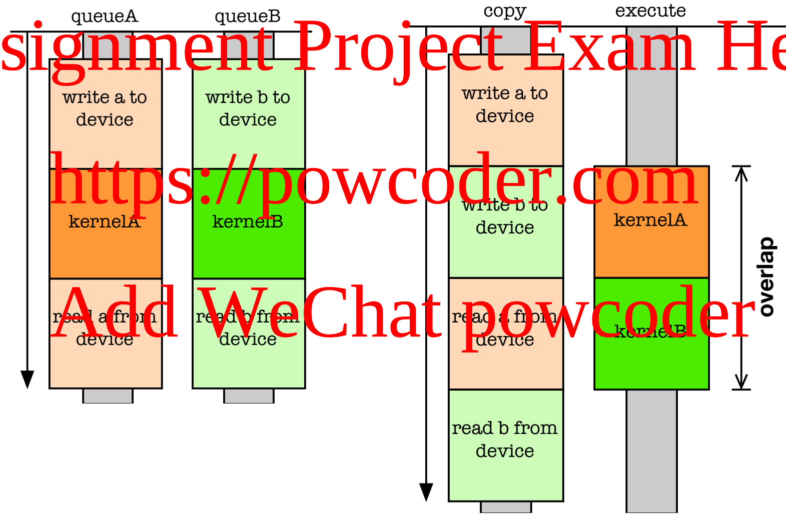
Suppose our device supports **asynchronous copy** and **simultaneous data transfer and kernel execution**.

- **Not guaranteed**, although common in modern GPUs.
- May require device to have **direct access** to host memory.

OpenCL with two command queues: Outline


```
// Initialise two command queues.
cl_command_queue queueA = clCreateCommandQueue(...);
cl_command_queue queueB = clCreateCommandQueue(...);

// Enqueue data transfer host->device (non-blocking).
clEnqueueWriteBuffer(queueA,...,CL_FALSE,...);
clEnqueueWriteBuffer(queueB,...,CL_FALSE,...);

// Enqueue both kernels.
clEnqueueNDRangeKernel(queueA,kernelA,...);
clEnqueueNDRangeKernel(queueB,kernelB,...);

// Enqueue data transfer device->host (blocking).
clEnqueueReadBuffer(queueA,...,CL_TRUE,...);
clEnqueueReadBuffer(queueB,...,CL_TRUE,...);

... // Process results; clear up.
```

Program logic **On the device**



Events in queues and streams

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

Earlier we saw how an **event** can be used as a timer.

```
cl_event timer;
```


In general, events are used to define **dependencies between kernels and data transfers**.

The last arguments on enqueue commands are:


```
clEnqueue...(...,numWait,waitEvents,event);
```

`cl_uint numWait`	Number of events to wait for.
`cl_event *waitEvents`	List of events to wait for.
`cl_event *event`	Used to identify when this operation completes.

Example (fragment)

Link together reads, writes and kernels **on multiple queues** — not necessary when using a single queue.

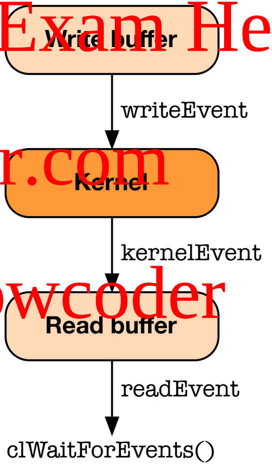

```
cl_event writeEvent, kernelEvent, readEvent;

// Non-blocking write host->device.
clEnqueueWriteBuffer(..,0,NULL,&writeEvent);

// Enqueue kernel.
clEnqueueNDRangeKernel(..,1,&writeEvent,&kernelEvent);

// Non-blocking read device->host.
clEnqueueReadBuffer(..,1,&kernelEvent,&readEvent);

// Synchronise (wait for read to complete).
clWaitForEvents(1,&readEvent);  // Sim. to MPI_Wait().
```

Task graphs

Events are used to **link** two tasks when the first must complete before the second starts.

Simple example of a **task graph**:

- **Directed, acyclic graph**.
- Nodes are **tasks**.
- Edges denote **dependencies**.
- **Direction** denotes which task must complete before the other begins.



Earlier task graphs

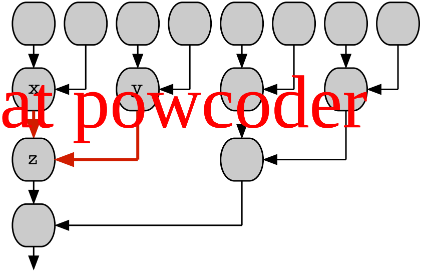
In fact, we have already seen examples of **task graphs**.

In **binary tree reduction** the arrows denote which calculations must be completed before continuing *[cf. Lecture 11]*.

Example:

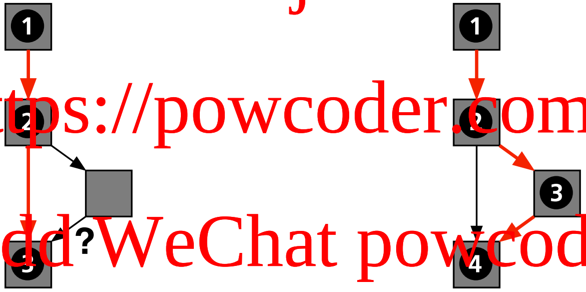
Must know x and y before we can calculate $z = x \otimes y$.

(These two dependencies highlighted in the diagram).



Satisfying dependencies

However many processing units, dependencies **must be satisfied**.



Invalid **Valid**

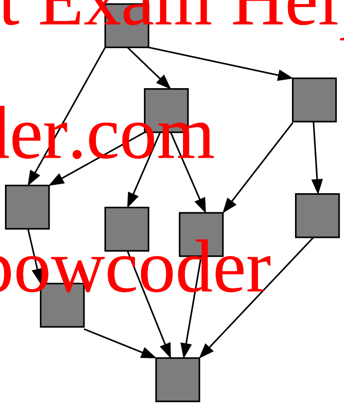
This means you must always take the 'longest path'.

Work-span model

Consider the task graph on the right:

- 9 **tasks** *(nodes)*.
- 13 **dependencies** *(arrows)*.

Assume all tasks take equal time.



The **performance** of a parallel program represented as a task graph can be estimated once the **work** and **span** have been identified.

Work and span

Definition

The **work** is the **total** time to complete all tasks.

This corresponds to a **serial** machine with $p = 1$ processing units.

Definition

The **span** is the time taken on an ideal machine with $p = \infty$.

As many tasks in parallel as possible **given the dependencies**.

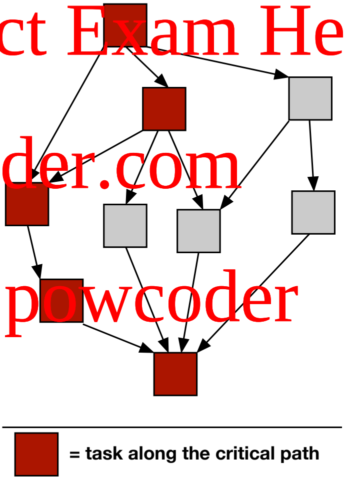
- The **span** is the **longest path** executed one after the other.
- Also called the **critical path**.

Span example

In this example, the number of tasks from start to finish, **given the dependencies**, is 5.

The **span** is therefore 5 tasks.

For uneven sized tasks would be measured in units of seconds/clock cycles/FLOPs *etc.*



Work-span model

Note that the **work** is just the serial execution time t_s.

We have argued that the parallel execution time can never become less than the **span**.

There is therefore a **maximum speed up** *[cf. Lecture 4]*:

$$S = \frac{t_\mathrm{s}}{t_\mathrm{p}} \leq \frac{t_\mathrm{s}}{t_{p=\infty}} = \frac{(\mathrm{work})}{(\mathrm{span})}$$

This is the **work-span model**.

- Upper limit[1] for S based purely on the task graph.
- $S \leq \frac{9}{5} = 1.8$ for this example.

[1] There is also a *lower* bound provided by Brent's lemma. R.P. Brent, *J. Ass. Comp. Mach.* **21**, 201 (1974).

Superscalar sequences and futures

Some parallel frameworks schedule tasks based on **dependencies specified by the programmer**.

- OpenCL from earlier in this lecture.
- OpenMP v4.0, and especially v4.5.
- **Futures** in C++11 and Java.

This is sometimes referred to as a **superscalar sequence**[1].

The benefit is that you do not need to **explicitly synchronise**.

- The runtime system synchronises when necessary, based on the dependencies you provide.

[1] McCool *et al.*, *Structured parallel programming* (Morgan-Kauffman, 2012).

Summary of GPGPU programming

Lec	Content	Key points
14	GPGPU architectures	SIMD cores; CUDA and OpenCL; starting with OpenCL.
15	Threads and kernels	Host vs. device; data transfer and kernel launches; work items and work groups.
16	Memory types	Global, local, private and constant.
17	Synchronisation	Barriers; breaking up kernels; subgroups advancing in lockstep; divergence.
18	Atomics	Global and local atomics; compare-and-exchange; lock-free data structures.
19	Task parallelism	GPU queues/streams; events; task graphs, work and span.

Next lecture

This penultimate lecture is the last containing new material.

In the next and last lecture, we will summarise the material by **parallel concept** rather than by architecture.

- Alternative perspective focussing on transferable insights.
- Also serves as a useful summary of the module material.