Parallel Computing with GPUs

CUDA Streams
Part 1 - Synchronous and Asynchronous Execution

The University Of Sheffield.

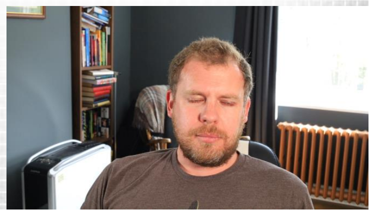
Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

- Synchronous and Asynchronous Execution
  - Classify synchronous and asynchronous execution
  - Demonstrate examples of synchronous execution with CUDA
  - Demonstrate examples of asynchronous execution with CUDA

Blocking and Non-Blocking Functions

- Synchronous vs Asynchronous
  - Synchronous:
    - Blocking call
    - Executed sequentially
  - Asynchronous:
    - Non-Blocking call
    - Control returns to host thread
- Asynchronous Advantages
  - Overlap execution and data movement on different devices
    - Not just GPU and CPU
    - Also consider disk or network (low latency)



Asynchronous Behaviour so far…

- CPU pipeline
  - Programmer writes code considering it to be synchronous operations
  - Compiler generates overlapping instructions to maximise pipe utilisation
  - Same end result as non overlapping instructions (hopefully)
- CPU threading
  - Similar threads execute asynchronously on different multiprocessors
  - Requires careful consideration of race conditions
  - OpenMP gives us critical sections etc. to help with this
- CUDA Warp execution
  - Threads in the same warp execute instructions synchronously
  - Warps on a SMP are interleaved and executed asynchronously
  - Careful use of `__syncthreads()` to ensure no race conditions

CUDA Host and Device

- ❑Most CUDA Host functions are synchronous (blocking)

- ❑Exceptions (synchronous with the host)
  - ❑Kernel calls
  - ❑`cudaMemcpy` within a device (`cudaMemcpyDeviceToDevice`)
  - ❑`cudaMemcpy` host to device of less than 64kB
  - ❑*Asynchronous memory copies and streams… (this lecture)*

- ❑Asynchronous functions will block when
  - ❑`deviceSynchronize()` is called
  - ❑A new kernel must be launched (implicit synchronisation)
  - ❑Memory must be copied to or from the device (implicit synchronisation)

Asynchronous Execution

```
//copy data to device
cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

//execute kernels on device
kernelA<<<blocks, threads>>>(d_a, d_b);
kernelB<<<blocks, threads>>>(d_b, d_c);

//copy back result data
cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

Is there any Asynchronous Execution?

Asynchronous Execution

```
//copy data to device
cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

//execute kernels on device
kernelA<<<blocks, threads>>>(d_a, d_b);
kernelB<<<blocks, threads>>>(d_b, d_c);

//copy back result data
cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

Completely Synchronous

Asynchronous Execution

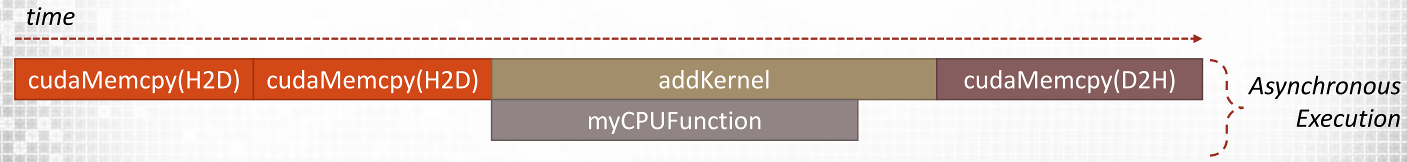
```
//copy data to device
cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

//execute kernel on device
addKernel<<<blocks, threads>>>(dev_c, dev_a, dev_b);

//host execution
myCPUFunction();

//copy back result data
cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

Is there any Asynchronous Execution?

Asynchronous Execution

```
//copy data to device
cudaMemcpy(dev_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

//execute kernel on device
addKernel<<<blocks, threads>>>(dev_c, dev_a, dev_b);

//host execution
myCPUFunction();

//copy back result data
cudaMemcpy(c, dev_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

Asynchronous GPU and CPU Execution



Summary

- ❑Synchronous and Asynchronous Execution
  - ❑Classify synchronous and asynchronous execution
  - ❑Demonstrate examples of synchronous execution with CUDA
  - ❑Demonstrate examples of asynchronous execution with CUDA
- ❑Next Lecture: CUDA Streams

Parallel Computing with GPUs

CUDA Streams
Part 2 - CUDA Streams

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

- ❑CUDA Streams
  - ❑Demonstrate how to make asynchronous memory calls
  - ❑Demonstrate how have *copy* and *compute* concurrency
  - ❑Give examples of the issue ordering implications of stream scheduling and concurrency

Opportunities for Device Concurrency

- Most CUDA Devices have an asynchronous Kernel execution and Copy Engine
  - Allows data to be moved at the same time as execution
  - Most device have dual copy engines
    - PCIe upstream (D2H)
    - PCIe downstream (H2D)
  - Ideally we should hide data movement with execution
  - Check your device capability: deviceQuery example "`Concurrent copy and kernel execution:`
- All modern GPU devices are able to execute kernels simultaneously
  - Allows task parallelism on GPU
  - Each kernel represents a different task
  - Very useful for smaller problem sizes

Streams

- CUDA Streams allow operations to be queued for the GPU device
  - All calls are asynchronous by default
  - The host retains control
  - Device takes work from the streams when it is able to do so
- Operations in a stream are ordered and can not overlap (FIFO)
- Operations in different streams can overlap

```
// create a handle for the stream
cudaStream_t stream;
//create the stream
cudaStreamCreate(&stream);

//do some work in the stream ...

//destroy the stream (blocks host until stream is complete)
cudaStreamDestroy(stream);
```

Work Assignment for Streams

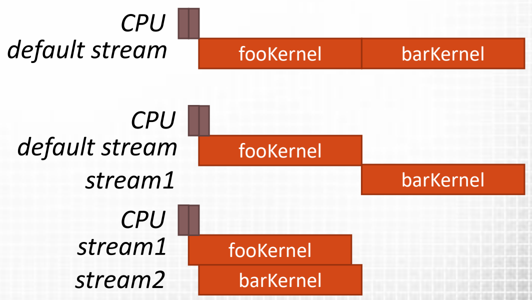
```
//execute kernel on device in specified stream
fooKernel<<<blocks, threads, 0, stream>>>();
```

- Kernel Execution is assigned to streams as 4th parameter of kernel launch
- Care must be taken with the default stream
  - Only stream which is synchronous with others!

```
fooKernel<<<blocks, threads, 0>>>();
barKernel<<<blocks, threads, 0>>>();
```

```
fooKernel<<<blocks, threads, 0>>>();
barKernel<<<blocks, threads, 0, stream>>>();
```

```
fooKernel<<<blocks, threads, 0, stream1>>>();
barKernel<<<blocks, threads, 0, stream2>>>();
```



Asynchronous Memory

- CUDA is able to asynchronously copy data
  - Only if it is Pinned (Page-locked) memory
- Paged Memory
  - Allocated using `malloc(…)` on host and released using `free(…)`
- Pinned Memory
  - Can not be swapped (paged) out by the OS
  - Has higher overhead for allocation
  - Can reach higher bandwidths for large transfers
  - Allocated using `cudaMallocHost(…)` and released using `cudaFreeHost(…)`
  - Can also pin non pinned memory using `cudaHostRegister(…)` / `cudaHostUnregister(…)`
    - Very slow

Concurrent Copies in Streams

- ❑Memory copies can be replaced with `cudaMemcpyAsync()`
  - ❑Requires an extra argument (a stream)
  - ❑Places transfer into the stream and returns control to host
  - ❑Conditions of use
    - ❑Must be pinned memory
    - ❑Must be in the non-default stream

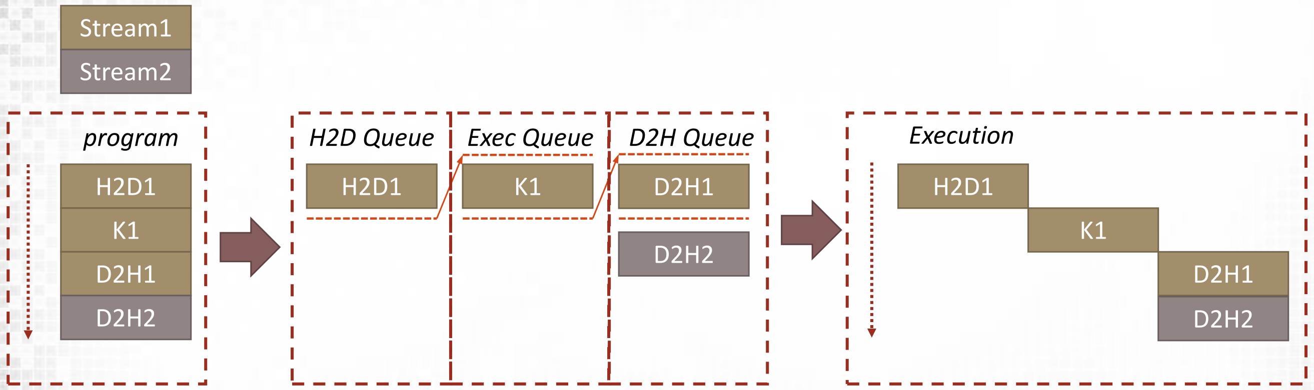
```
int *h_A, *d_A;
cudaStream_t stream1;

cudaStreamCreate(&stream1);
cudaMallocHost(&h_A, SIZE);
cudaMalloc(&d_A, SIZE);
initialiseA(h_A);

cudaMemcpyAsync(d_A, h_A, SIZE, cudaMemcpyHostToDevice, stream1);

//work in other streams ...

cudaStreamDestroy(stream1);
```

Stream Scheduling

- ❑CUDA operations dispatched to hardware in sequence that they were issued
  - ❑Hence issue order is important (FIFO)
- ❑Kernel and Copy Engine (x2) have different queues

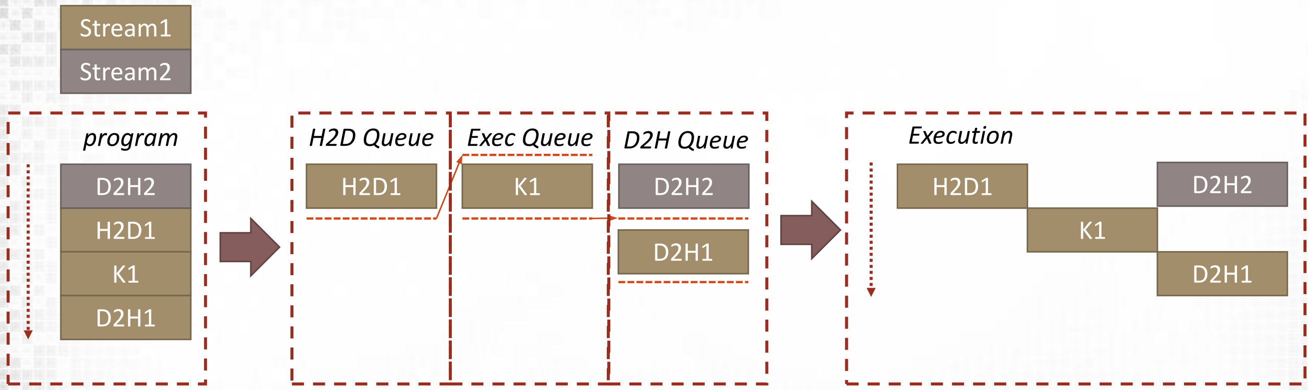
- ❑Operations are de-queued if
  1. Preceding call in the same stream have completed
  2. Preceding calls in the same queue have been dispatched, and
  3. Resources are available
     - ❑i.e. kernels can be concurrently executed if in different streams
- ❑Blocking operations (e.g. `cudaMemcpy` will block all streams)

Issue Ordering



- ❑No Concurrency of D2H2
- ❑Blocked by D2H1
  - ❑Issued first (FIFO)

Issue Ordering



- ❑Concurrency of D2H2 and H2D1

Issue Ordering (Kernel Execution)

Stream1
Stream2

Exec Queue: fooKernel, barKernel, fooKernel, barKernel → *Execution*

Exec Queue: fooKernel, fooKernel, barKernel, barKernel → *Execution*

❑ Which has best Asynchronous execution?

Issue Ordering (Kernel Execution)

Stream1
Stream2

Exec Queue: fooKernel, barKernel, fooKernel, barKernel → *Execution*: fooKernel; barKernel, fooKernel; barKernel

Exec Queue: fooKernel, fooKernel, barKernel, barKernel → *Execution*: fooKernel, fooKernel; barKernel, barKernel

❑ barKernel can't be removed from queue until fooKernel has completed

❑ Blocks fooKernel

❑ Both fooKernels can be concurrently executed

❑ Both barKernels concurrently executed

Levels of Concurrency

H2D | KernelA<<<...>>> | D2H

Fully Synchronous (Serial Execution)

H2D KA1 D2H / KA2 D2H / KA3 D2H / KA4 D2H

2-way Concurrency
- ❑ H2D and D2H not concurrent

H2D KA1 D2H / H2D KA2 D2H / H2D KA3 D2H / H2D KA4 D2H

3-way Concurrency
- ❑ Both Copy Engines active
- ❑ Execution Engine active
  - ❑ May or may not be fully utilised

H2D KA1 KB1 KC1 D2H / H2D KA2 KB2 KC2 D2H / H2D KA3 KB3 KC3 D2H / H2D KA4 KB4 KA4 D2H / H2D KA5 KB5 KC5 D2H

5-way Concurrency
- ❑ Both Copy Engines active
- ❑ Execution Engine active
  - ❑ Higher independent workload
  - ❑ Better chance of 100% utilisation
- ❑ What about Host?

Summary

- ❑ CUDA Streams
  - ❑ Demonstrate how to make asynchronous memory calls
  - ❑ Demonstrate how have *copy* and *compute* concurrency
  - ❑ Give examples of the issue ordering implications of stream scheduling and concurrency

❑ Next Lecture: Synchronisation

Parallel Computing with GPUs

CUDA Streams
Part 3 - Synchronisation

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

- ❑Synchronisation
  - ❑Identify and explain different levels of synchronisation
  - ❑Introduce and give examples of events
  - ❑Demonstrate the use of callbacks

Explicit Device Synchronisation

- ❑What if we want to ensure an asynchronous kernel call has completed?
  - ❑For timing kernel execution
  - ❑Accessing data copied asynchronously without causing race conditions
- ❑`cudaDeviceSynchronize()`
  - ❑Will ensure that all asynchronous device operations are completed
  - ❑Synchronise everything!
- ❑`cudaStreamSyncronize(stream)`
  - ❑Blocks host until all calls in stream are complete
- ❑*CUDA Event synchronisation…*

Events

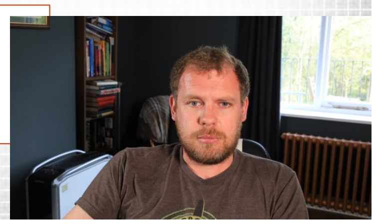
- ❑Mechanism in which to signal when operations have occurred in a stream
  - ❑Places an event into a stream (default stream unless specified)
- ❑We have seen events already!
  - ❑When timing our code…

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaEventRecord(start);
my_kernel <<<(N /TPB), TPB >>>();
cudaEventRecord(stop);

cudaEventSynchronize(stop);
float milliseconds = 0;
cudaEventElapsedTime(&milliseconds, start, stop);

cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Events and Streams

- ❑ `cudaEventRecord(event, stream)`
  - ❑ Places an event in the non default stream
- ❑ `cudaEventSynchronize(event)`
  - ❑ Blocks until the stream completes all outstanding calls
  - ❑ Should be called <u>after</u> the event is inserted into the stream
- ❑ `cudaStreamWaitEvent(stream, event)`
  - ❑ Blocks the stream until the event occurs
  - ❑ Only blocks launches after event
  - ❑ Does not block the host
- ❑ `cudaEventQuery(event, stream)`
  - ❑ Has the event occurred in the stream

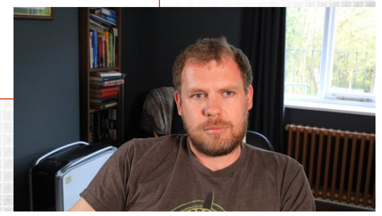
```
cudaMemcpyAsync(d_in, in, size, H2D, stream1);
cudaEventRecord(event, stream1); // record event

cudaStreamWaitEvent(stream2, event); // wait for event in stream1
kernel << <BLOCKS, TPB, 0, stream2 >> > (d_in, d_out);
```



Callbacks

- ❑ Callbacks are functions on the host which should be called when an event is reached
- ❑ `cudaStreamAddCallback(stream, callback, user_data, 0)`
  - ❑ Good for scheduling host code once event has completed
  - ❑ Allows GPU to initiate operations that only the CPU can perform
    - ❑ Disk or network IO
    - ❑ System calls, etc.

```
void CUDART_CB MyCallback(void *data){
    //some host code
}

MyKernel << <BLOCKS, TPB, 0, stream >> >(d_i);
cudaStreamAddCallback(stream, MyCallback, (void*)d_i, 0);
```



WDDM Command Queues

- ❑ GPUs driving a display in windows use the Windows Display Driver Model Command Queues.
  - ❑ All CUDA calls (sync/async) are buffered within a WDDM *Command Buffer*
  - ❑ The *Command Buffer* will usually be flushed by
    - ❑ Forcing it by calling `cudaEventQuery(0)`
    - ❑ Issuing a synchronous call. E.g. a stream/device sync or synchronous memcpy
    - ❑ Waiting until it gets full (unpredictable)
    - ❑ Magic???
- ❑ Implications
  - ❑ Only things in the same command buffer can be concurrent
  - ❑ Stuff *might* not get queued into copy/compute engines as you would expect
  - ❑ `cudaEventElapsedTime` *may* not be accurate for asynchronous host timing

https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9957-using-cuda-on-windows.pdf

Summary

- ❑ Synchronisation
  - ❑ Identify and explain different levels of synchronisation
  - ❑ Introduce and give examples of events
  - ❑ Demonstrate the use of callbacks

- ❑ Next Lecture: Multi GPU Programming

Parallel Computing with GPUs

CUDA Streams
Part 4 – Multi GPU Programming

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

This Lecture (learning objectives)

- Multi GPU Programming
  - Demonstrate how to change the GPU device
  - Explain how devices can be concurrently operated and synchronised using streams
  - Demonstrate mechanisms for device to device asynchronous memory copying

Multi GPU Programming

- By default CUDA uses the first device in the system
  - Not necessarily the fastest device!
- Device can be changed using `cudaSetDevice(int)`
  - Device capabilities can be queried using device properties API

```
int deviceCount = 0;
cudaGetDeviceCount(&deviceCount);

for (int dev = 0; dev < deviceCount; ++dev)
{
    cudaSetDevice(dev);
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, dev);
    …
}
```

Multi GPU Devices and Streams

- Streams and events belong to a single device
  - The device which is active when created
  - Synchronising and Querying of streams across devices is allowed

```
cudaStream_t streamA, streamB;
cudaEvent_t eventA, eventB;

cudaSetDevice(0);
cudaStreamCreate(&streamA); // streamA and eventA belong to device-0
cudaEventCreate(&eventA);

cudaSetDevice(1);
cudaStreamCreate(&streamB); // streamB and eventB belong to device-1
cudaEventCreate(&eventB);
kernel << <..., streamB >> >(...);
cudaEventRecord(eventB, streamB);

cudaSetDevice(0);
cudaEventSynchronize(eventB);
kernel << <..., streamA >> >(...);
```

Error: eventA belongs to device 0

Event can be synchronised across devices

Peer to Peer Memory Copies

- For devices to interact memory must be copied between them
- Memory can be copied using
  - `cudaMemcpyPeerAsync( void* dst_addr, int dst_dev, void* src_addr, int src_dev, size_t num_bytes, cudaStream_t stream )`
    - Uses shortest PCI path or GPUDirect if available
    - Not staged through CPU
- You can check that a peer (device) can access another using
  - `cudaDeviceCanAccessPeer( &accessible, dev_X, dev_Y )`
- Also possible to use CUDA aware MPI
  - Allows direct transfers over the network
  - With NVLink this will allow GPU to GPU peer access via infiniband
  - *Not covered in this course…*

Further Reading & Acknowledgements

- Most slide examples are based on the excellent GTC and SC material
  - http://www.sie.es/wp-content/uploads/2015/12/cuda-streams-best-practices-common-pitfalls.pdf
  - http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf
  - http://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf
- More reading
  - https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/
  - https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/

Summary

- Multi GPU Programming
  - Demonstrate how to change the GPU device
  - Explain how devices can be concurrently operated and synchronised using streams
  - Demonstrate mechanisms for device to device asynchronous memory copying