Lecture 6
Using multiple GPUs and loose ends

Prof Wes Armour

wes.armour@eng.ox.ac.uk

Prof Mike Giles

mike.giles@maths.ox.ac.uk

Oxford e-Research Centre

Department of Engineering Science

Learning outcomes

In this sixth lecture we will look at CUDA streams and how they can be used to increase performance in GPU computing. We will also look at some other useful odds and ends.

You will learn about:

- Synchronicity between host and device.
- Multiple streams and devices.
- How to use multiple GPUs.
- Some other odds and ends.

Setting the scene

Modern computers are typically comprised of many different components.

- Central Processing Unit (CPU).
- Random Access Memory (RAM).
- Graphics Processing Unit (GPUs).
- Hard Disk Drive (HDD) / Solid State Drive (SSD).
- Network Interface Controller (NIC)…

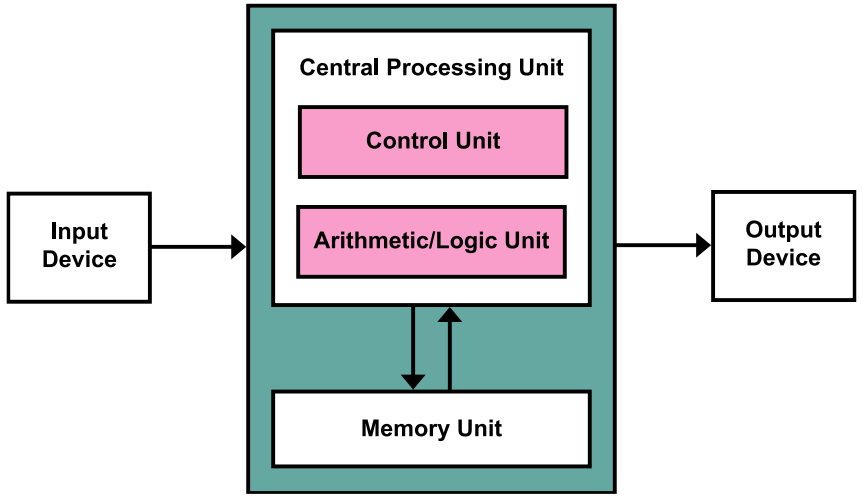
Typically, each of these different components will be performing a different task, maybe for different processes, at the same time.

Synchronicity

Synchronicity

The von Neumann model of a computer program is synchronous with each computational step taking place one after another (because instruction fetch and data movement share the same communication bus).

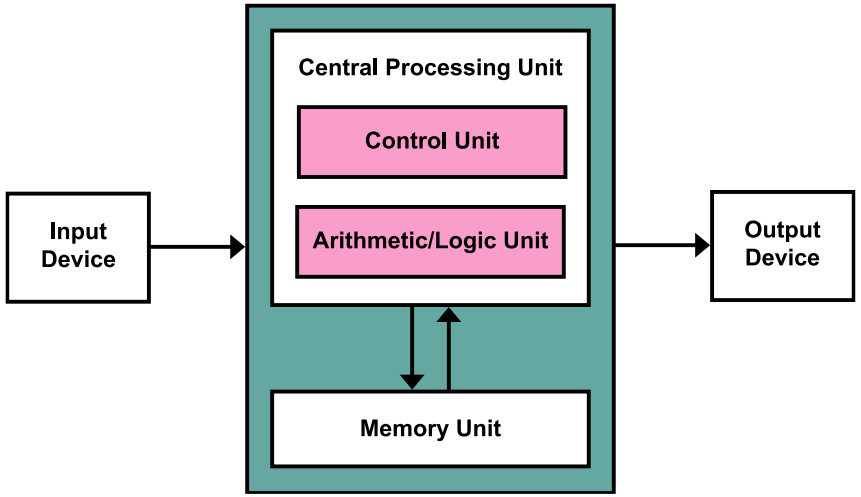
This is an idealisation, and is almost never true in practice.



Synchronicity

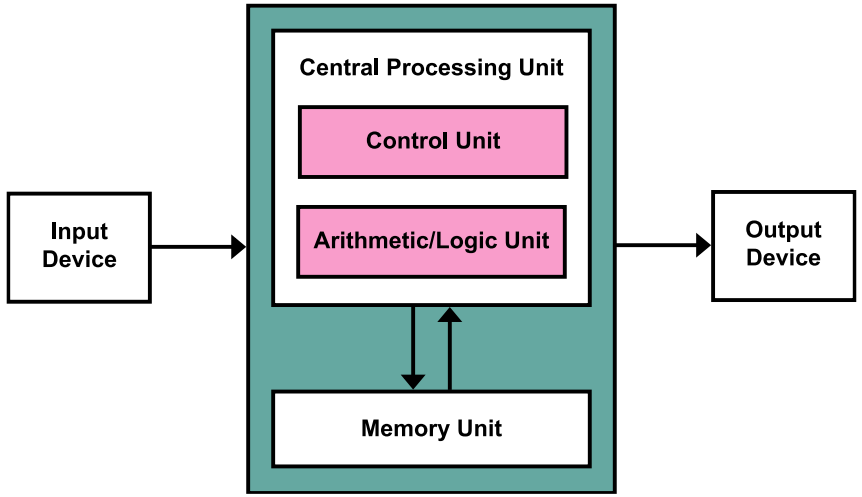
Compilers will generate code with overlapped instructions (pipelining – see lecture one), re-arrange execution order and avoid redundant computations to produce more optimal code.

As a programmer we don't normally worry about this and think of execution sequentially when working out whether a program gives the correct result.



Synchronicity

However, when things become asynchronous, the programmer has to think very carefully about what is happening and in what order!



Synchronicity - GPUs

When writing code for GPUs we have to think even more carefully, because:

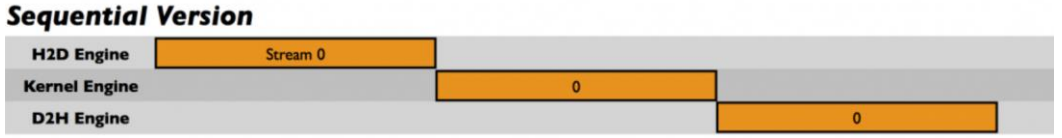
Our host code executes on the CPU(s);

Our kernel code executes on the GPU(s)

. . . but when do the different bits take place?

. . . can we get better performance by being clever?

. . . might we get the wrong results?



The most important thing is to try to get a clear idea of what is going on, and when – then you can work out the consequences…

Simple host code

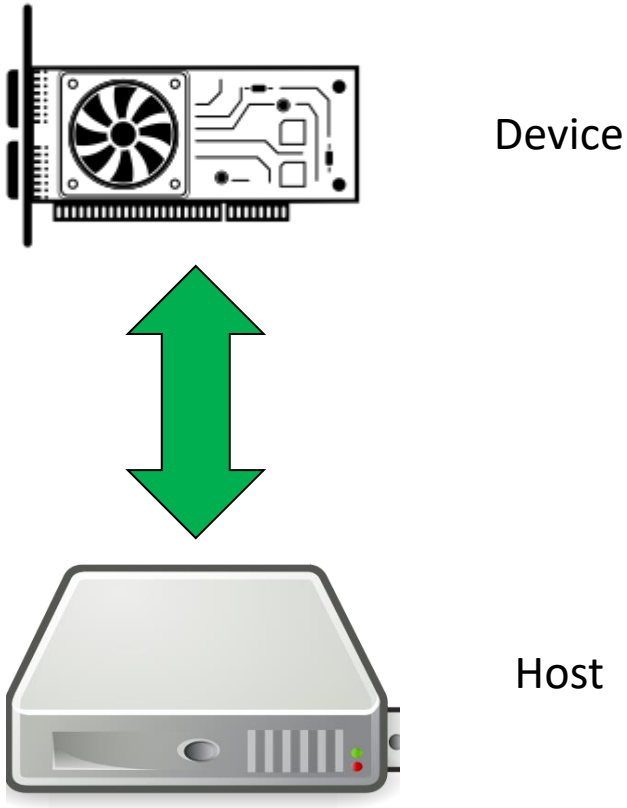
The basic / simple / default behaviour in CUDA is that we have:

1x CPU.

1x GPU.

1x thread on CPU (i.e. scalar code).

1x “**stream**” on GPU (called the “**default stream**”).



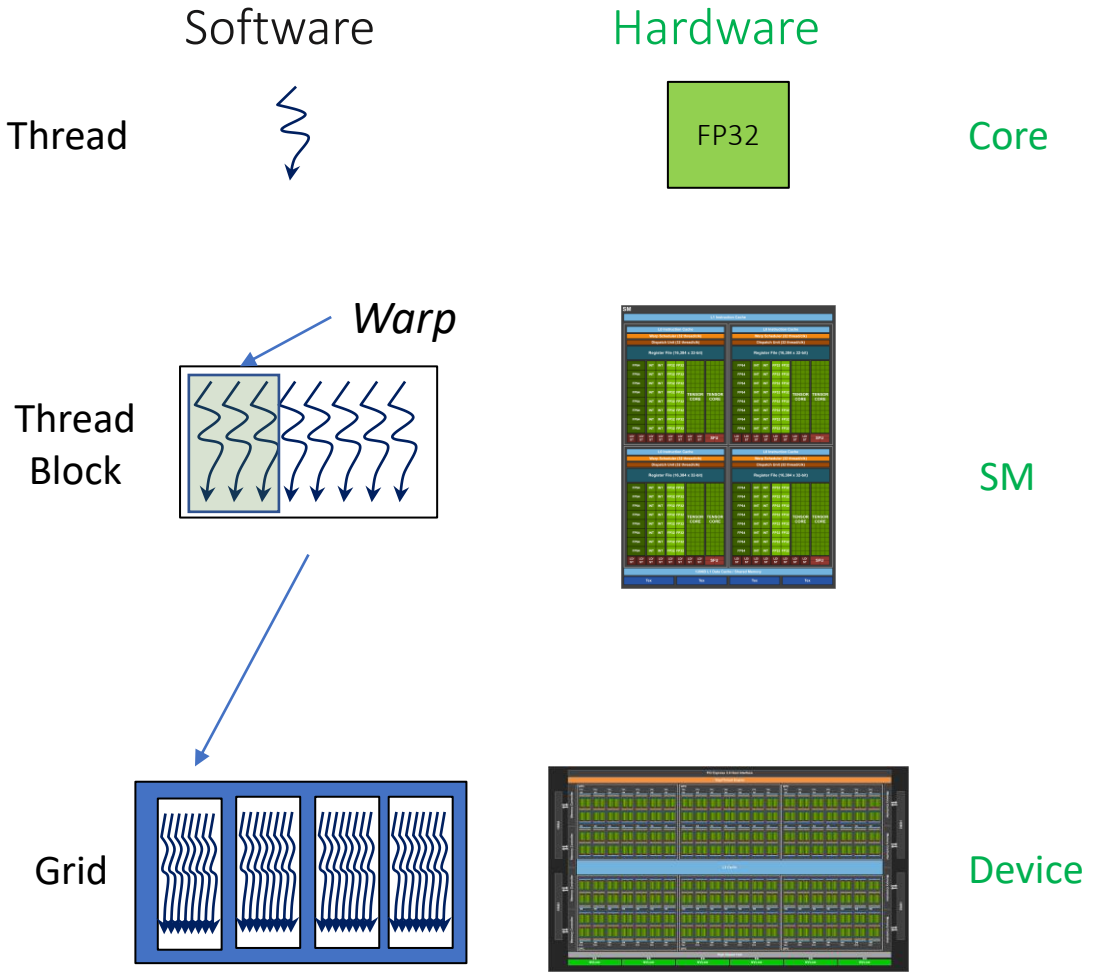
Recap – GPU Execution

We've looked at how code executes on GPUs, lets have a quick recap:

- For each warp, code execution is effectively synchronous within the warp.
- Different warps execute in an arbitrary overlapped fashion – use `__syncthreads()` if necessary to ensure correct behaviour.
- Different thread blocks execute in an arbitrary overlapped fashion.

So nothing new here.

*Over the next few slides we will discuss streams – **asynchronous execution** and the implications for CPU and GPU execution.*



Blocking and non-blocking calls

Host code – blocking calls

Most CUDA calls are synchronous (often called “blocking”).

An example of a blocking call is `cudaMemcpy()`.

1. Host call starts the copy (HostToDevice / DeviceToHost).
2. Host **waits** until it the copy has finished.
3. Host continues with the next instruction in the host code once the copy has completed.

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

Host code – blocking calls

Why do this???

This mode of operation ensures correct execution.

For example it ensures that data is present if the next instruction needs to read from the data that has been copied…

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

Host code – blocking calls

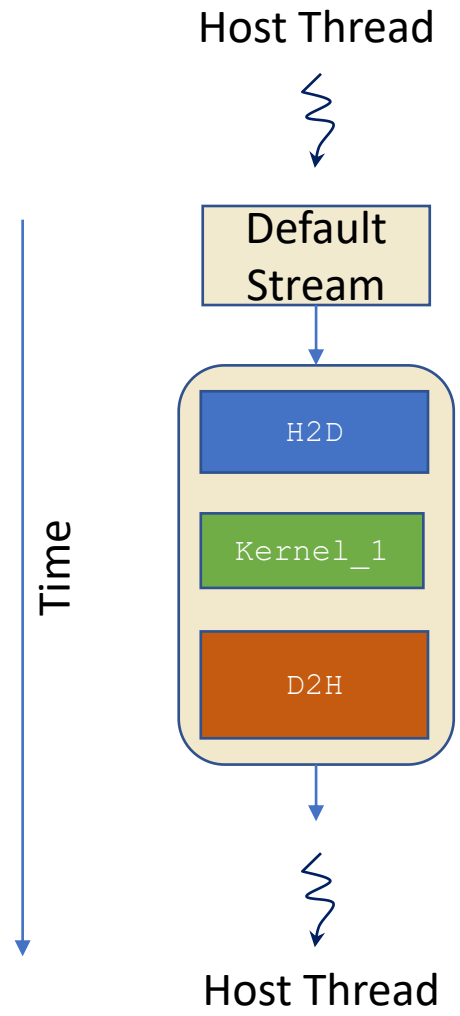
So the control flow for our code looks something like…

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
cudaMemcpy ( …, D2H );

…
```

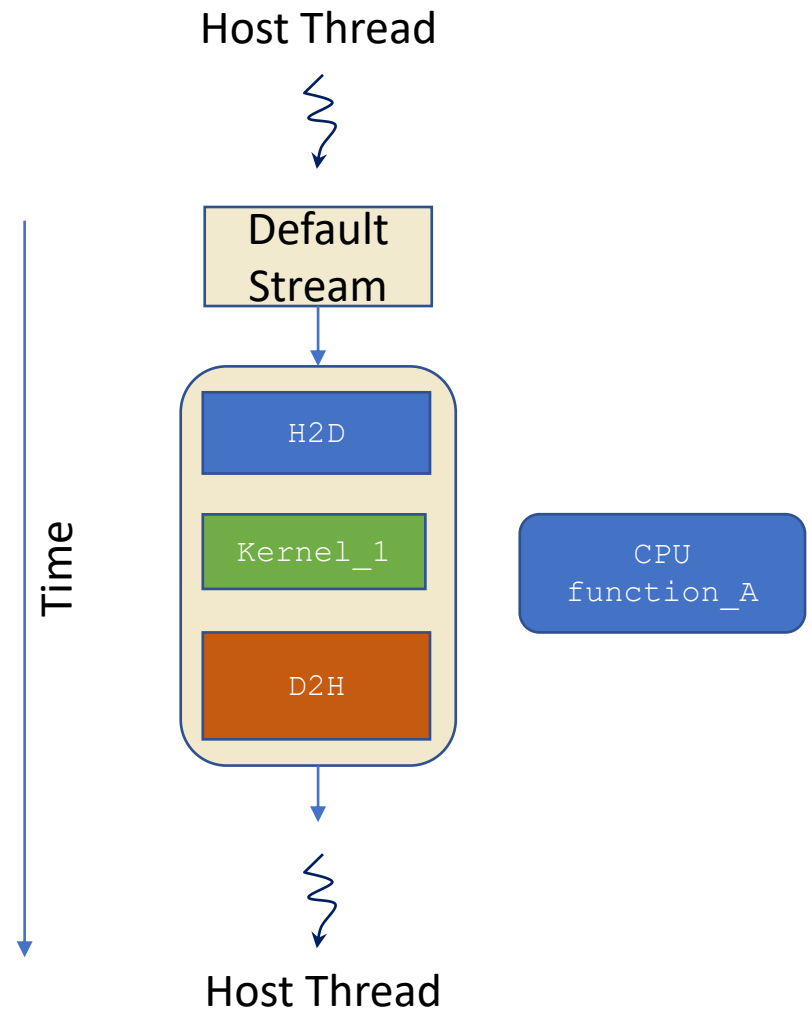


Host code – non-blocking calls

In CUDA, kernel launches are asynchronous (often called “non-blocking”).

An example of kernel execution from host perspective:

1. Host call starts the kernel execution.
2. Host does not wait for kernel execution to finish.
3. Host moves onto the next instruction.



Host code – non-blocking calls

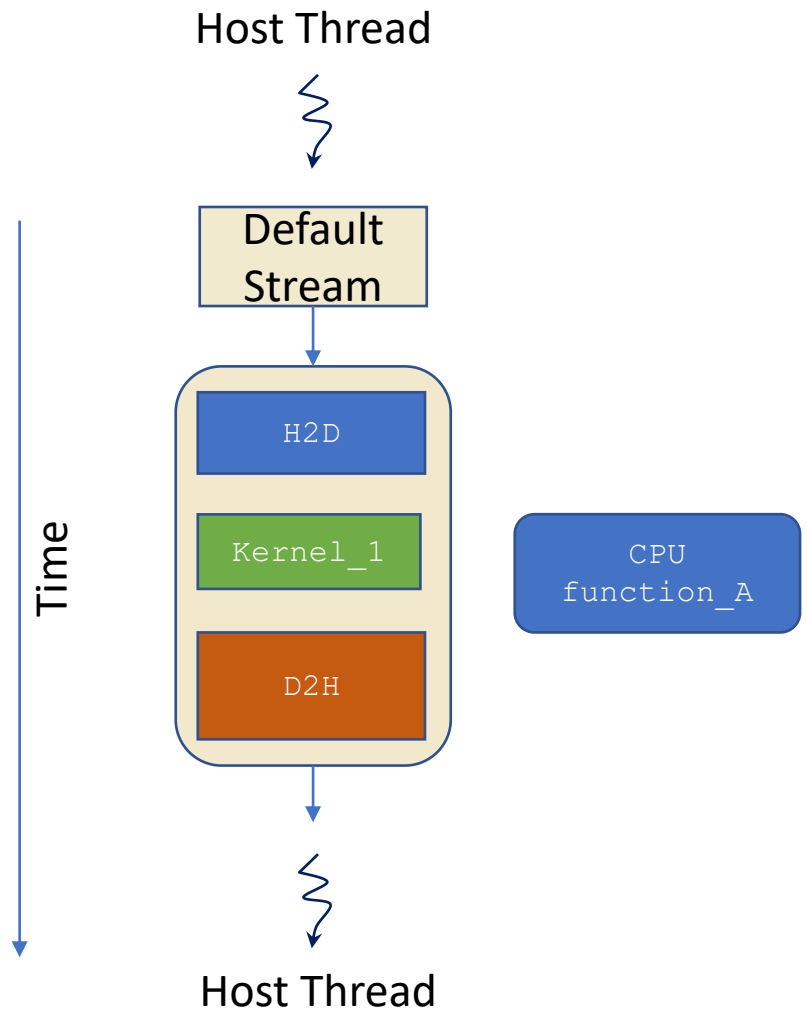
The “crazy code” for our last control flow diagram might look like…

```
cudaMalloc(&d_data, size);
float *h_data = (double*)malloc(size);

…

cudaMemcpy( d_data, h_data, size, H2D ) ;
kernel_1 <<< grid, block >>> ( … ) ;
CPU_function_A( … );
cudaMemcpy ( …, D2H );

…
```



Host code – non-blocking calls

Another example of a non-blocking call is `cudaMemcpyAsync()`.

This function starts the copy but doesn't wait for completion.

Synchronisation is performed through a "stream".

You must use page-locked memory (also known as pinned memory) – see Documentation.

In both of our examples, the host eventually waits when at (for example) a `cudaDeviceSynchronize()` call.

The benefit of using streams is that you can improve performance (in some cases, not all) by overlapping communication and compute, or CPU and GPU execution.

Asynchronous host code

When using asynchronous calls, things to watch out for, and things that can go wrong are:

- Kernel timing – need to make sure it's finished.
- Could be a problem if the host uses data which is read/written directly by kernel, or transferred by `cudaMemcpyAsync()`.
- `cudaDeviceSynchronize()` can be used to ensure correctness (similar to `syncthreads()` for kernel code).

CUDA Streams

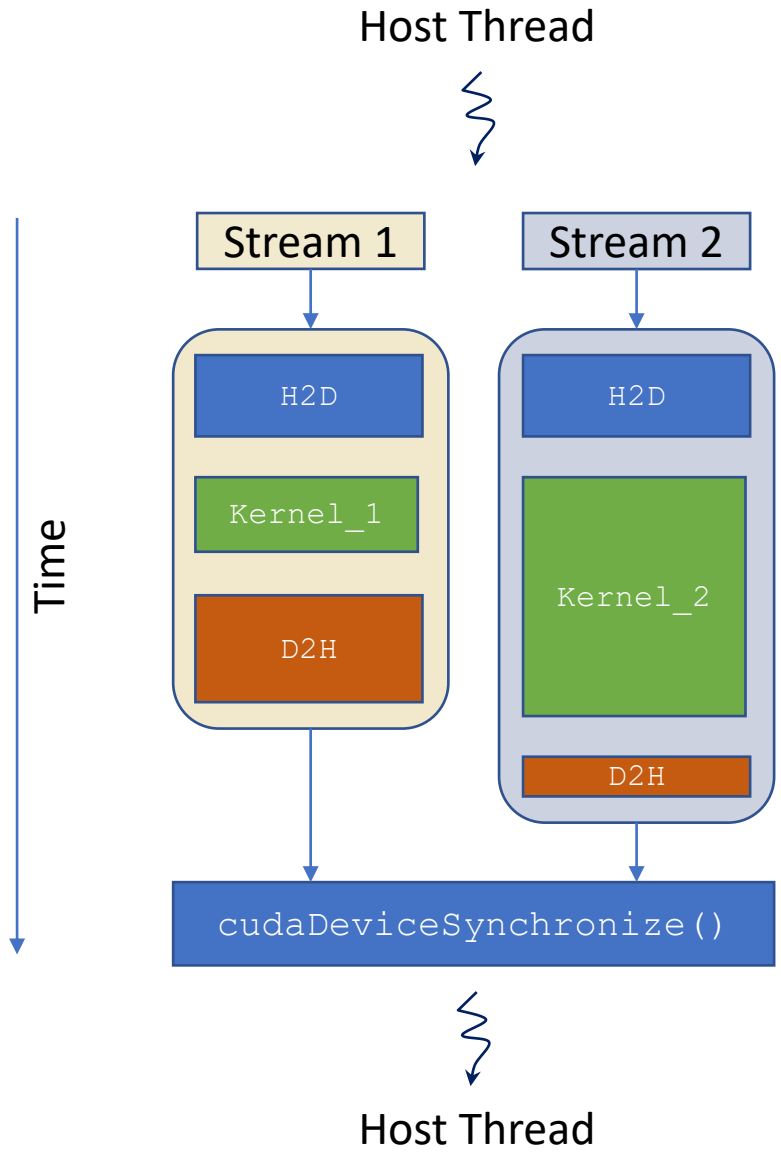
CUDA Streams

Quoting from section 6.2.8.5 in the CUDA Programming Guide:

Applications manage concurrency through streams.

A stream is a sequence of commands (*possibly issued by different host threads*) that execute in order.

Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently.



http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf

Multiple CUDA Streams

When using streams in CUDA, you must supply a "stream" variable as an argument to:

- kernel launch
- `cudaMemcpyAsync()`

Which is created using `cudaStreamCreate();`

As shown over the last couple of slides:

- Operations within the same stream are ordered - (i.e. FIFO – first in, first out) – they cant overlap.
- Operations in different streams are unordered wrt each other and can overlap.

Use multiple streams to increase performance by overlapping memory communication with compute.

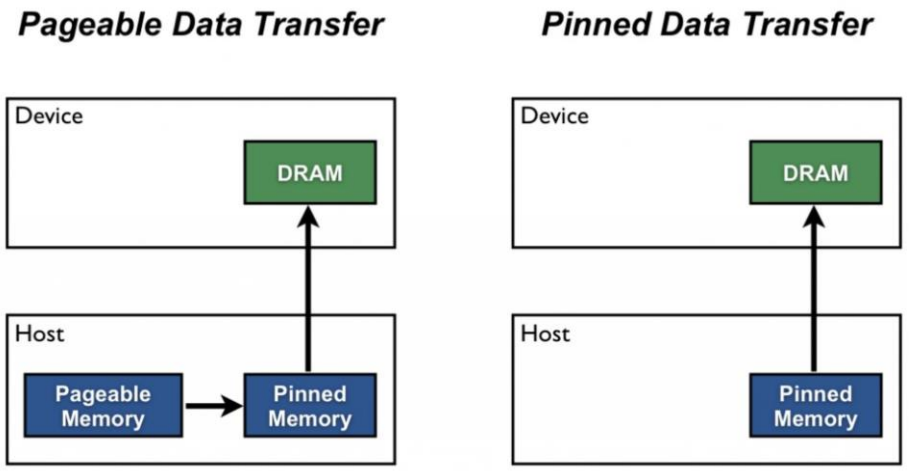
```
cudaStream_t stream1;
cudaStreamCreate(&stream1);
my_kernel_one<<<blocks,threads,0,stream1>>>(…);
cudaStreamDestroy(stream1);
```

An example of launching a kernel in a stream that isn't the "default stream".

Page-locked / Pinned memory

Section 6.2.6 of the cuda programming guide:

- Host memory is usually paged, so run-time system keeps track of where each page is located.
- For higher performance, pages can be fixed (fixed address space, always in RAM), but means less memory available for everything else.
- CUDA uses this for better host <–> GPU bandwidth, and also to hold “device” arrays in host memory.
- Can provide up to 100% improvement in bandwidth
- You must use page-locked memory with `cudaMemcpyAsync();`
- Page-locked memory is allocated using `cudaHostAlloc()`, or registered by `cudaHostRegister();`



Pinned memory is used as a staging area for transfers from the device to the host. We can avoid the cost of the transfer between pageable and pinned host arrays by directly allocating our host arrays in pinned memory.

https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/

The default stream

The way the default stream behaves in relation to others depends on a compiler flag:

no flag, or `--default-stream legacy`

This forces old (bad) behaviour in which a cudaMemcpy or kernel launch on the default stream blocks/synchronizes with other streams.

Or `--default-stream per-thread`

This forces new (good) behaviour in which the default stream doesn't affect the others.

For more info see the nvcc documentation:

https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#options-for-steering-cuda-compilation

Practical 11

An example is given in practical 11 for those interested, try with the two different flags:

```
cudaStream_t streams[8];
float *data[8];

for (int i = 0; i < 8; i++) {
  cudaStreamCreate(&streams[i]);
  cudaMalloc(&data[i], N * sizeof(float));

  // launch one worker kernel per stream
  kernel<<<1, 64, 0, streams[i]>>>(data[i], N);

  // do a Memcpy and launch a dummy kernel on default stream
  cudaMemcpy(d_data,h_data,sizeof(float),
             cudaMemcpyHostToDevice);
  kernel<<<1, 1>>>(d_data, 0);
}
cudaDeviceSynchronize();
```

The default stream

The second (most useful?) effect of the flag comes when using multiple threads (e.g. OpenMP or POSIX multithreading).

In this case the effect of the flag is to create separate independent (i.e. non-interacting) default streams for each thread.

Using multiple default streams, one per thread, is a useful alternative to using “proper” streams.

However “proper” streams within cuda are very versatile and fully featured, so might be worth the time and complexity investment.

```
omp_set_num_threads(8);
float *data[8];

for (int i = 0; i < 8; i++)
  cudaMalloc(&data[i], N * sizeof(float));

#pragma omp parallel for
for (int i = 0; i < 8; i++) {
  printf(" thread ID = %d \n",omp_get_thread_num());

  // launch one worker kernel per thread
  kernel<<<1, 64>>>(data[i], N);
}

cudaDeviceSynchronize();
```

Stream commands

As previously shown, each stream executes a sequence of cuda calls. However to get the most out of your heterogeneous computer you might also want to do something on the host.

There are at least two ways of coordinating this:

Use a separate thread for each stream

- It can wait for the completion of all pending tasks, then do what's needed on the host.

Use just one thread for everything

- For each stream, add a callback function to be executed (by a new thread) when the pending tasks are completed.
- It can do what's needed on the host, and then launch new kernels (with a possible new callback) if wanted.

Stream commands

Some useful stream commands are:

`cudaStreamCreate(&stream)`
Creates a stream and returns an opaque “handle” – the “stream variable”.

`cudaStreamSynchronize(stream)`
Waits until all preceding commands have completed.

`cudaStreamQuery(stream)`
Checks whether all preceding commands have completed.

`cudaStreamAddCallback()`
Adds a callback function to be executed on the host once all preceding commands have completed.

http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf
https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

Stream commands

Functions useful for synchronisation and timing between streams:

`cudaEventCreate(event)`
Creates an “event”.

`cudaEventRecord(event,stream)`
Puts an event into a stream (by default, stream 0).

`cudaEventSynchronize(event)`
CPU waits until event occurs.

`cudaStreamWaitEvent(stream,event)`
Stream waits until event occurs (doesn’t block the host).

`cudaEventQuery(event)`
Check whether event has occurred.

`cudaEventElapsedTime(time,event1,event2)`
Times between event1 and event2.

Multi-GPU computing

Multiple devices

What happens if there are multiple GPUs?

CUDA devices within the system are numbered, not always in order of decreasing performance!

- By default a CUDA application uses the lowest number device which is “visible” and available (this might not be what you want).
- Visibility controlled by environment variable `CUDA_VISIBLE_DEVICES`.
- The current device can be chosen/set by using `cudaSetDevice()`
- `cudaGetDeviceProperties()` does what it says, and is very useful.
- Each stream is associated with a particular device, which is the “current” device for a kernel launch or a memory copy.
- see `simpleMultiGPU` example in SDK or section 6.2.9 for more information.

https://www.flickr.com/photos/ebastian/8804000077

Multiple devices

If a user is running on multiple GPUs, data can go directly between GPUs (peer – peer), it doesn't have to go via CPU.

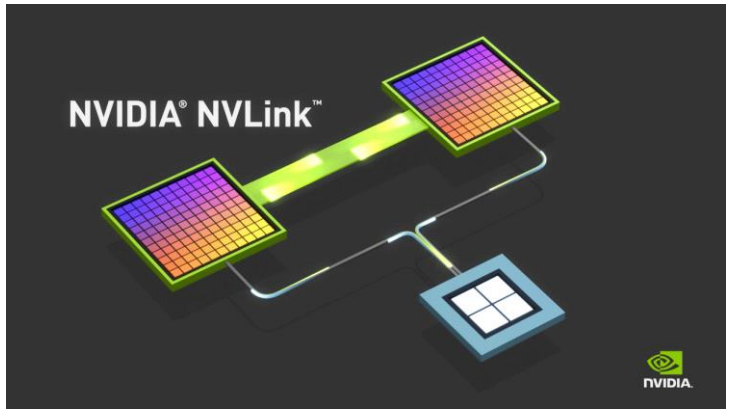
This is the premise of the NVlink interconnect, which is much faster than PCIe (900GB/s P2P on Hopper).

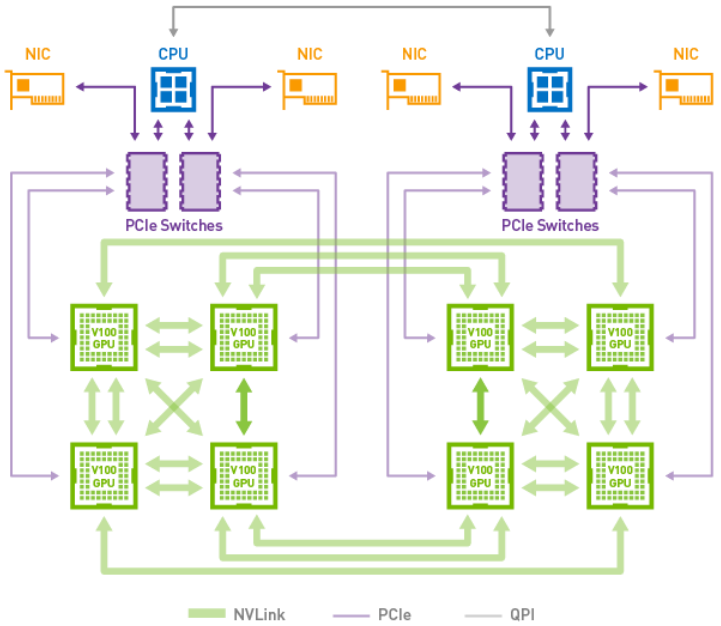
`cudaMemcpy()` can do direct copy from one GPU's memory to another.

A kernel on one GPU can also read directly from an array in another GPU's memory, or write to it.
This even includes the ability to do atomic operations with remote GPU memory.

For more information see Section 6.13, "Peer Device Memory Access" in CUDA Runtime API documentation:
https://docs.nvidia.com/cuda/cuda-runtime-api/

https://fuse.wikichip.org/news/1224/a-look-at-nvidias-nvlink-interconnect-and-the-nvswitch/





Multi-GPU computing

Multi-GPU computing exists at all scales, from cheaper workstations using PCIe, to more expensive Quadro / Titan products using fewer NVLink, to high-end NVIDIA DGX servers.

Single workstation / server:

- a big enclosure for good cooling!
- up to 4 high-end cards in 16x PCIe v4 slots – up to 16GB/s interconnect.
- 2x high-end CPUs.
- 2-3kW power consumption – not one for the office!

NVIDIA DGX H100 Deep Learning server:

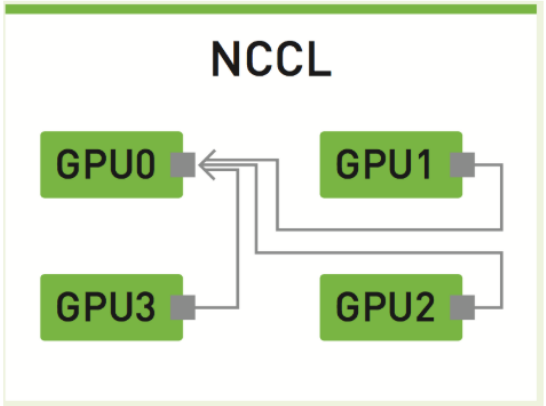
- 8 NVIDIA GH100 GPUs, each with 80GB HBM2.
- 2× 56-core Intel Xeons (Platinum 8480C 2.0 GHz).
- 2 TB RAM memory, 8x 3.84TB NVMe.
- 900GB/s NVlink interconnect between the GPUs.
- ~£379,000

Multi-GPU Computing

How do you use these machines?

This depends on hardware choice:

- For single machines, use shared-memory multithreaded host application.
- For DGX products you must use the NVIDIA Collective Communications Library (NCCL).
- For clusters / supercomputers, use distributed-memory MPI message-passing.



https://devblogs.nvidia.com/fast-multi-gpu-collectives-nccl/

MPI approach

In the MPI approach:

- One GPU per MPI process (nice and simple).
- Distributed-memory message passing between MPI processes (tedious but not difficult).
- Scales well to very large applications.
- Main difficulty is that the user has to partition their problem (break it up into separate large pieces for each process) and then explicitly manage the communication.
- Note: should investigate GPU Direct for maximum performance in message passing.

Multi-user support

What if different processes try to use the same device?

The behaviour of the device depends on the system compute mode setting (section 3.4):

In “default” mode, each process uses the fastest device:
- This is good when one very fast card, and one very slow card.
- But not very useful when you have two identical fast GPUs (one sits idle).

In “exclusive” mode, each process is assigned to first unused device;
However code will return an error if none are available.

`cudaGetDeviceProperties()` reports the mode setting

The mode can be changed by a user account with sys-admin privileges using the `nvidia-smi` command line utility.

Some tips and tricks

Loose ends – Loop unrolling

Section 10.37 (of the programming guide):
loop unrolling, If you have a loop:

```
for (int k=0; k<4; k++) a[i] += b[i];
```

Then nvcc will automatically unroll this to give:

```
a[0] += b[0];
a[1] += b[1];
a[2] += b[2];
a[3] += b[3];
```

This is a standard compiler trick to avoid the cost of incrementing and looping.

The pragma

```
#pragma unroll 5
```

will also force unrolling for loops that do not have explicit limits.

Loose ends – const __restrict__

Section 10.2.6 (of the programming guide):

`__restrict__` keyword

The qualifier asserts that there is no overlap (in memory space) between `a,b,c`, for example we do not have:

```
a[i]=q[i]
b[i]=q[i+1]
```

(you have no pointer aliasing) so the compiler can perform more optimisations.

The following blog post demonstrates how this can achieve a good speed increase:

https://devblogs.nvidia.com/cuda-pro-tip-optimize-pointer-aliasing/#disqus_thread

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
               float* __restrict__ c) {

        for (i=1; i<N; i++) {
                a[i] = b[i] + c[i];
        }
        ...
}
```

Loose ends - volatile

Section 17.5.3.3 (of the programming guide):
`volatile` keyword

Tells the compiler **the variable may change at any time**, so not to re-use a value which may have been loaded earlier and apparently not changed since.

This can sometimes be important when using shared memory because the compiler can optimize locations in shared memory by locating them in registers (but register scope is specific to a single thread), for any thread.

Loose ends - Compilation

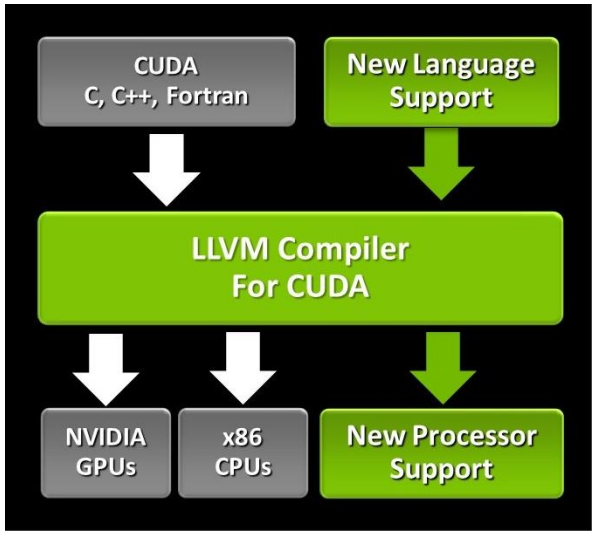
Compiling:

The `Makefile` for first few practicals uses `nvcc` to compile both the host and the device code.

Internally `nvcc` uses `gcc` for the host code (at least by default). The device code compiler is based on the open source LLVM compiler.

It often makes sense to use different compilers, for example `icc` which is for host code which does not have kernel launches.

To do this you must use the `-fPIC` flag to produce position-independent-code (this just generates machine code that will execute properly, independent of where it's held in memory).



https://developer.nvidia.com/cuda-llvm-compiler

Loose ends - Compilation

Prac 6 Makefile:

```
INC   := -I$(CUDA_HOME)/include -I.
LIB   := -L$(CUDA_HOME)/lib64 -lcudart
FLAGS := --ptxas-options=-v --use_fast_math

main.o: main.cpp
	g++ -c -fPIC -o main.o main.cpp

prac6.o: prac6.cu
	nvcc prac6.cu -c -o prac6.o $(INC) $(FLAGS)

prac6: main.o prac6.o
	g++ -fPIC -o prac6 main.o prac6.o $(LIB)
```

Loose ends - Compilation

Prac 6 Makefile to create a library:

```
INC   := -I$(CUDA)/include -I.
LIB   := -L$(CUDA)/lib64 -lcudart
FLAGS := --ptxas-options=-v --use_fast_math

main.o: main.cpp
     g++ -c -fPIC -o main.o main.cpp

prac6.a: prac6.cu
     nvcc prac6.cu -lib -o prac6.a $(INC) $(FLAGS)

prac6a: main.o prac6.a
     g++ -fPIC -o prac6a main.o prac6.a $(LIB)
```

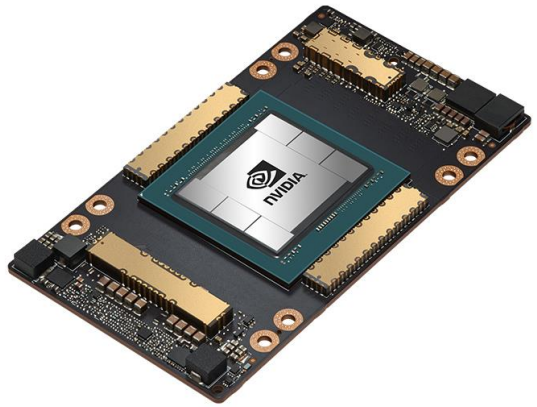
Loose ends - Compilation

Other useful compiler options:

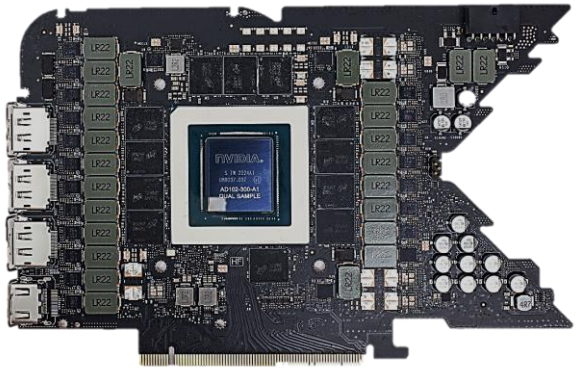
`-arch=sm_80`
This specifies GPU architecture (in this case sm_80 is for Ampere A100).

`-maxrregcount=n`
This asks the compiler to **generate code using at most n registers**; the compiler may ignore this if it's not possible, but it may also increase register usage up to this limit.

This is less important now since threads can have up to 255 registers, but can be useful in some instances to reduce register pressure and enable more thread blocks to run.



or



Loose ends – Compilation

Launch bounds (10.36):

`-maxrregcount` is given as an argument to the compiler (`nvcc`) and modifies the default for all kernels.

A per kernel approach can be taken by using the `__launch_bounds__` qualifier:

```
__global__ void
__launch_bounds__(maxThreadsPerBlock,minBlocksPerMultiprocessor)
MyKernel(...) {
…
}
```

Summary

This lecture has discussed a number of more advanced topics. As a beginner, you can ignore almost all of them As you get more experienced, you will probably want to start using some of them to get the very best performance.