Lecture 1
An introduction to CUDA

Prof Wes Armour

wes.armour@eng.ox.ac.uk

Oxford e-Research Centre

Department of Engineering Science

Learning outcomes

In this first lecture we will set the scene for GPU computing.

You will learn about:

- GPU hardware and how it fits in to HPC.
- Different generations of GPUs and current state of the art.
- The design of a GPU.
- How to work with GPUs and the CUDA programming language.

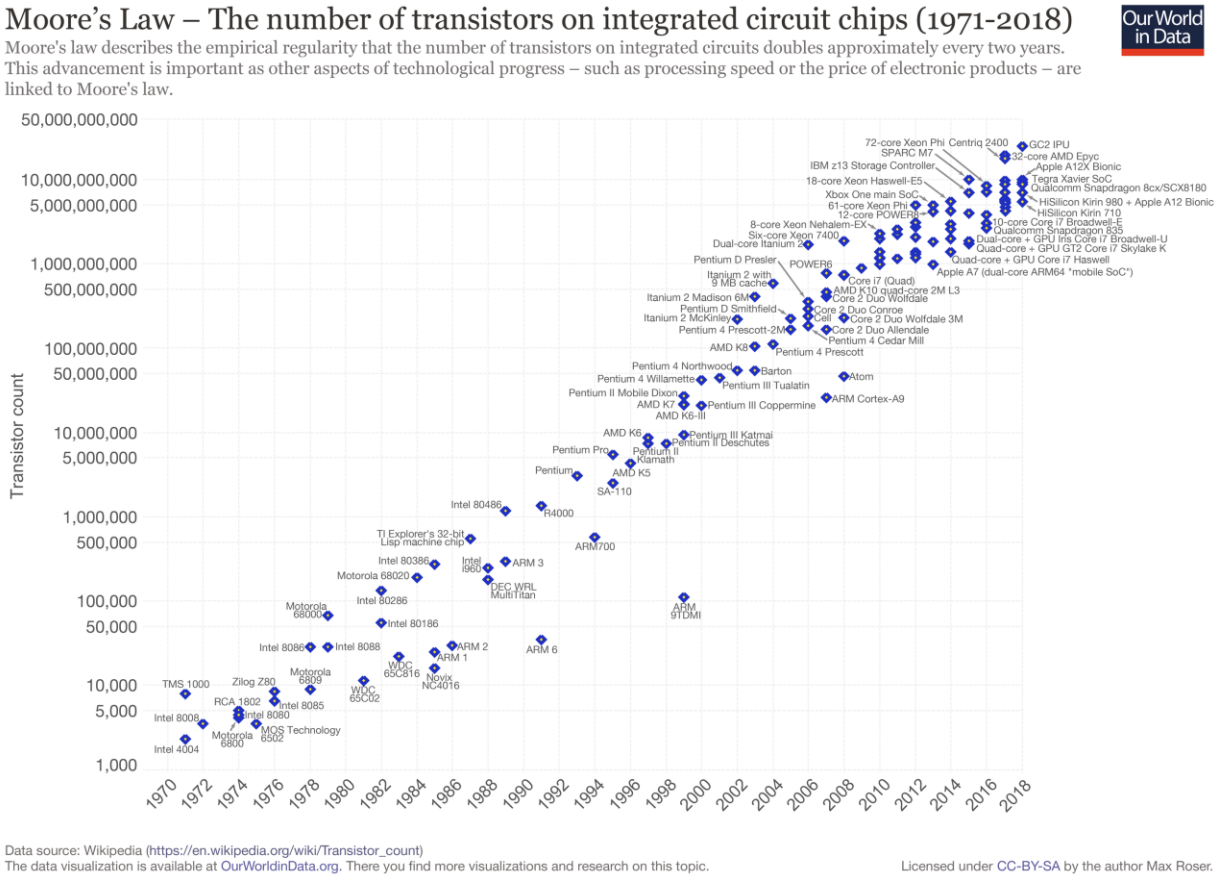
Motivation for GPU computing

Moore's law (roughly) states that the number of transistors in an integrated circuit will double every two years. Moore made this prediction in 1965!

Over the last decade, we have begun to see an end to this (see plot, more in Tim Lanfear's lecture).

This motivates the need for other ways to increase computational performance.

This is where Heterogeneous computing (specifically for us, GPU computing) comes in.



High level hardware overview

Heterogeneous computing makes use of more than one type of computer processor (for example a CPU and a GPU).

Heterogeneous computing is sometimes called hybrid computing or accelerated computing.

Some of the motivations for employing heterogeneous technologies are:

- Significant reductions in floor space needed.
- Energy efficiency.
- Higher throughput.

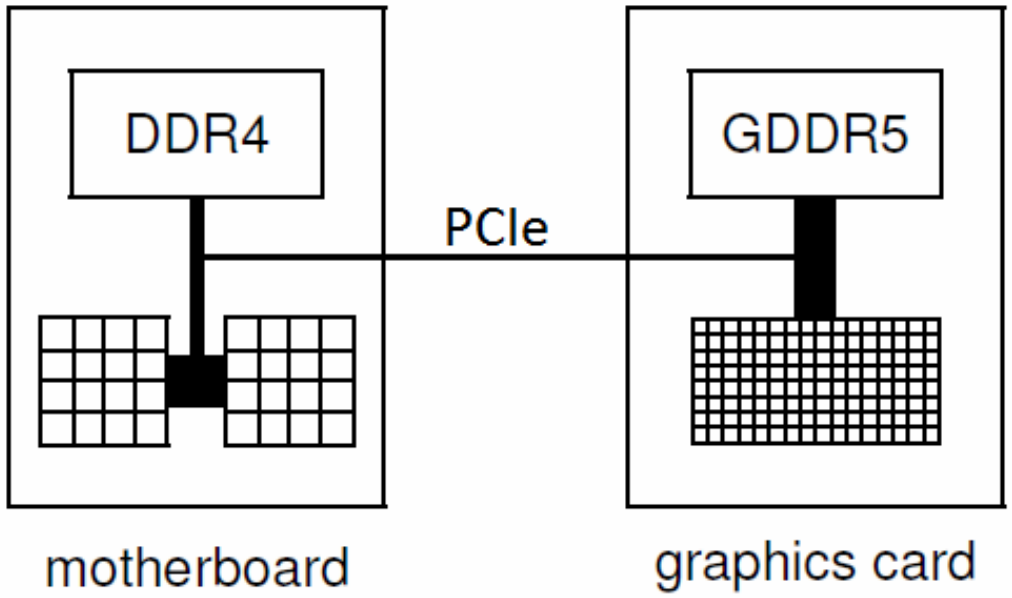
High level hardware overview

A typical server configuration is to have one or more CPUs (typically two) communicating with one or more GPUs through the PCIe bus.

The CPU has 10's of cores, the GPU has 1000's.

The server in which the GPU sits is often referred to as the "**host**"

The GPU itself is often called the **"device"**



Generations of GPUs

Several generations of GPUs have been released now.

Starting in 2007 with the first General-Purpose Graphics Processing Unit (GPGPU) called Tesla.

The “Tesla” name has been subsequently used as the identifier for all NVIDIA HPC cards.

Currently, 5 generations of hardware cards are in use, although the Kepler and Maxwell generations are becoming more scarce.

Kepler (compute capability 3.x):

- first released in 2012, including HPC cards.
- excellent double precision arithmetic (DP or fp64).
- **our practicals will use K40s and K80s.**

Maxwell (compute capability 5.x):

- first released in 2014.
- an architecture for gaming, so poor DP.

Pascal (compute capability 6.x):

- first released in 2016.
- many gaming cards and several HPC cards.

Volta (compute capability 7.x):

- first released end 2017 / start 2018.
- only HPC cards, excellent DP.

Turing (compute capability 7.5):

- first released Q3 2018.
- this is the gaming version of Volta.
- some HPC cards (specifically for AI inference). Poor DP.

Current start of the art

Due to the tailoring of GPUs for AI we now have two generations of GPU meeting the needs of:

Gamers: RTX cards based on the Turing architecture.

AI/AR/VR researchers: Titan cards and T4 based on both Volta and Turing architectures.

Scientific Computing researchers: Titan V and V100 cards with good DP.

Currently the biggest differentiation factor between the generations is the double precision performance

(Volta = good DP, Turing = bad DP)

The **Volta** generation has only HPC (Tesla) and "prosumer" (Titan) cards:

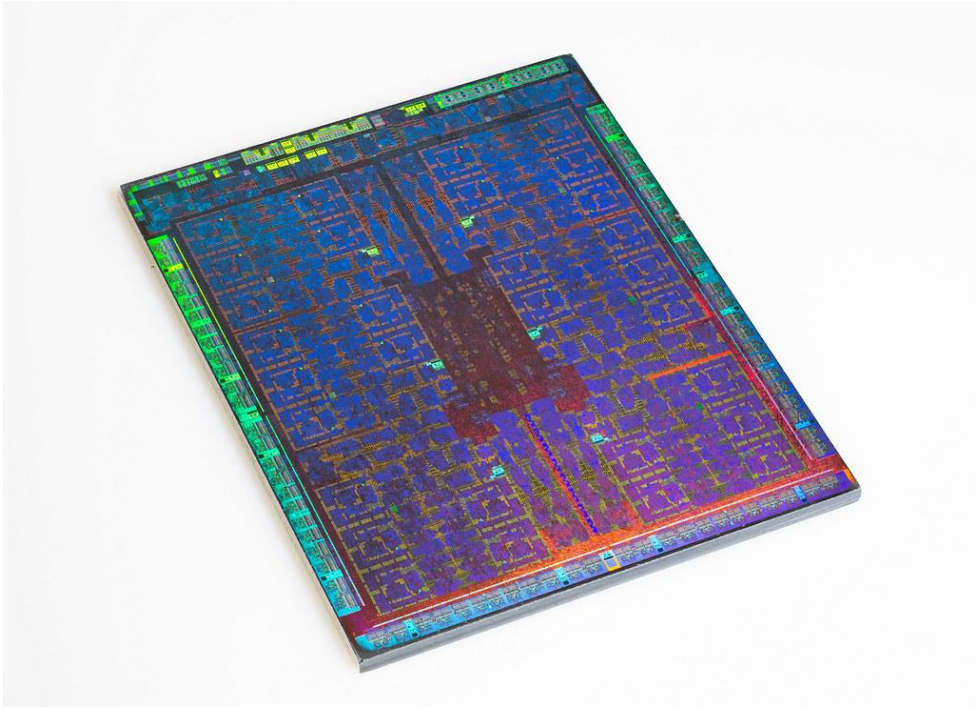
Titan V: 5120 cores, 12GB (£2900)
Tesla V100: 5120 cores, 16/32GB, PCIe or NVLink (£380)

The **Turing** generation has cards for gaming (GeForce) and AI cards (Titan and T4):

GeForce RTX 2060: 1920 cores, 6GB (£300)
GeForce RTX 2080Ti: 4352 cores, 11GB (£1100)

Titan RTX: 4608 cores, 24GB (£2400)
T4: 2560 cores, 16GB, HHHW, Low power (£2300)

The GPU



GPU chip (sometimes called the die)

A block diagram of the V100 GPU

Basic building blocks

The basic building block of a GPU is the ***"Streaming Multiprocessor"*** or ***SM***. This contains:

	Pascal	Volta	Turing
Cores	64	64	64
L1 / Shared Memory	64KB	96KB	64KB
L2 Cache	4096KB	6144KB	6144KB
Max # threads	2048	2048	1024

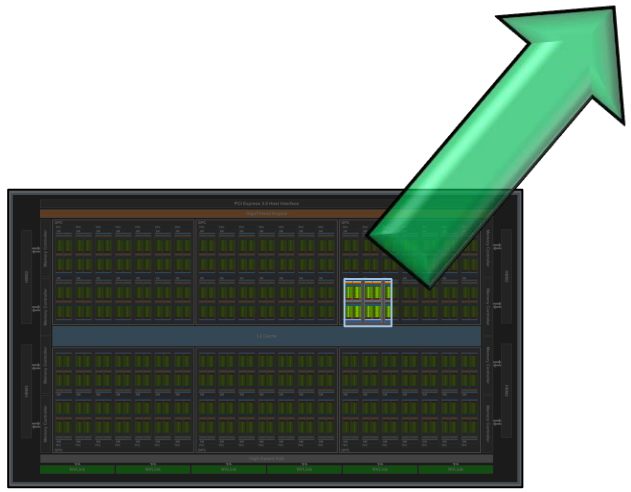
Looking into the SM

A V100 has 80 SMs (see right).

The GV100 SM incorporates 64 FP32 cores and 32 FP64 cores per SM.

The SM is partitioned into four processing blocks, each with:

- 16 FP32 Cores;
- 8 FP64 Cores;
- 16 INT32 Cores;
- 128KB L1 / Shared memory;
- 64 KB Register File.



Looking into the SM

Lets look at this in more detail

A **FP32 core** is the execution unit that performs single precision floating point arithmetic (floats).

A **FP64 core** performs double precision arithmetic (doubles).

A **INT32 Core** performs integer arithmetic.

The **warp scheduler** selects which warp (group of 32 threads) to send to which execution units (more to come).

64 KB **Register File** – lots of transistors used for vary fast memory.

128KB of configurable **L1 (data) cache** or **shared memory**
Shared memory is a used manged cache (more to come).

LD/ST units load and store data to/from cores.

SFU – special function units compute things like transcendentals.

Different numbers of SMs

Different products have different numbers of SMs, but although the number of SMs across the product range might vary, ***the SM is exactly the same for each generation***.

Product	Generation	SMs	Bandwidth	Memory	Power
RTX 2060	Turing	30	336 GB/s	6 GB	160 W
RTX 2070	Turing	36	448 GB/s	8 GB	175 W
RTX 2080	Turing	46	448 GB/s	8 GB	215 W
Titan RTX	Turing	72	672 GB/s	24 GB	250 W

Different numbers of SMs

Typically each GPU generation brings improvements in the number of SMs, the bandwidth to device (GPU) memory and the amount of memory on each GPU.
Sometimes NVIDIA use rather confusing naming schemes….

Product	Generation	SMs	Bandwidth	Memory	Power
GTX Titan	Kepler	14	288 GB/s	6 GB	230 W
GTX Titan X	Maxwell	24	336 GB/s	12 GB	250 W
Titan Xp	Pascal	30	548 GB/s	12 GB	250 W
Titan V	Volta	80	653 GB/s	12 GB	250 W
Titan RTX	Turing	72	672 GB/s	24 GB	250 W

Multiple GPU Chips

Some “GPUs” (the actual device that plugs into a PCIe slot) has multiple GPU chips on them…

Product	Generation	SMs	Bandwidth	Memory	Power
GTX 595	Fermi	2x 16	2x 164 GB/s	2x 1.5 GB	365 W
GTX 690	Kepler	2x 8	2x 192 GB/s	2x 2 GB	300 W
Tesla K80	Kepler	2x 13	2x 240 GB/s	2x 12 GB	300 W
Tesla M60	Maxwell	2x 16	2x 160 GB/s	2x 8 GB	300 W

Multithreading

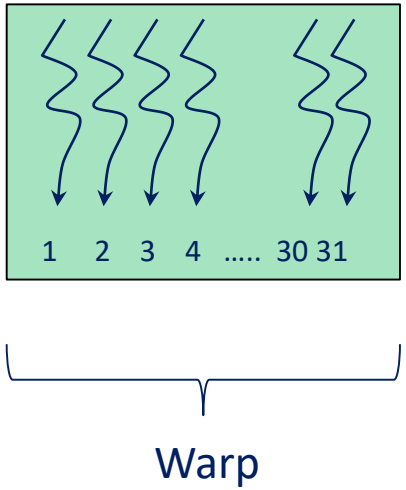
Key hardware feature is that the cores in a SM are SIMT (*Single Instruction Multiple Threads*) cores:

- Groups of 32 cores execute the ***same instructions*** simultaneously, but with ***different data.***
- Similar to vector computing on CRAY supercomputers.
- 32 threads all doing the **same thing** at the **same time** *(threads are in lock-step).*
- Natural for graphics processing and much of scientific computing.
- SIMT is also a natural choice for many-core chips to simplify each core.

Multithreading

Having **lots of active threads** is the **key to high performance**:

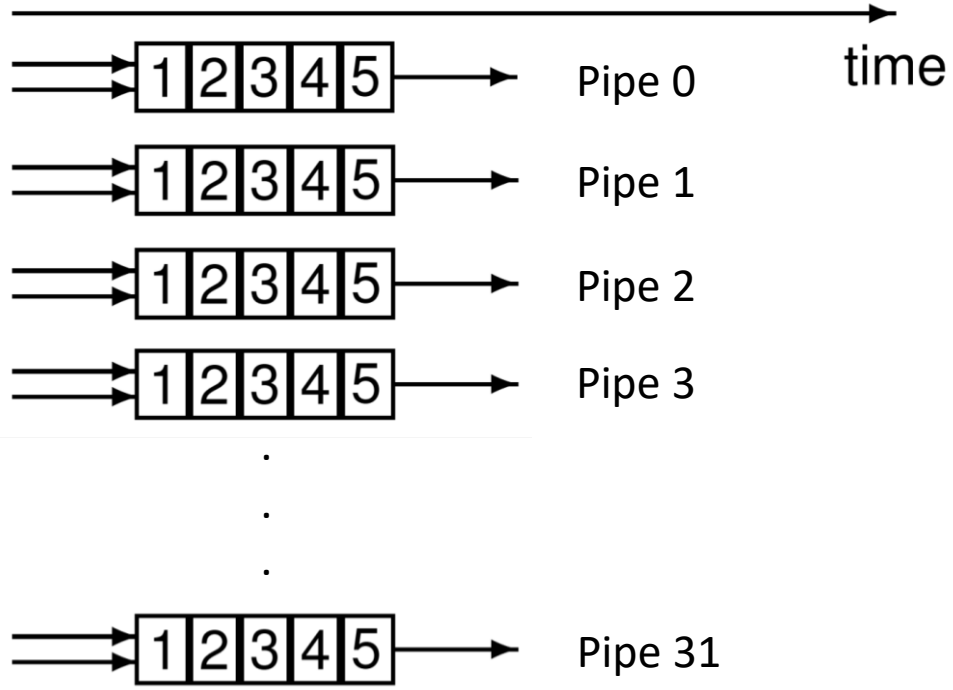
- GPUs do not exploit “context switching”; each thread has its own registers, which limits the number of active threads.
- Threads on each SM execute in groups of 32 called “***warps***”
- Execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data.



Multithreading

Originally, each thread completed one operation before the next started to avoid complexity of ***pipeline overlaps***, some examples are:

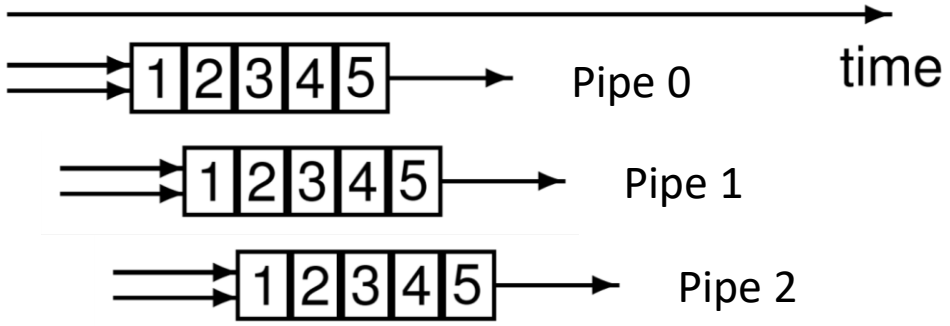
- **Structural** – two instructions need to use the same physical hardware component.
- **Data** – one instruction needs the result of a previous instruction and has to wait (stall - inefficient).
- **Branch** – waiting for a conditional branch to complete before we know whether to execute following instructions.



Multithreading

NVIDIA relaxed this restriction, so each thread can have multiple independent instructions overlapping, but for our purposes we will assume each instruction within a warp is lock-step.

Memory access from device memory has a delay of 200-400 cycles; with 40 active warps this is equivalent to 5-10 operations, so enough to hide the latency?

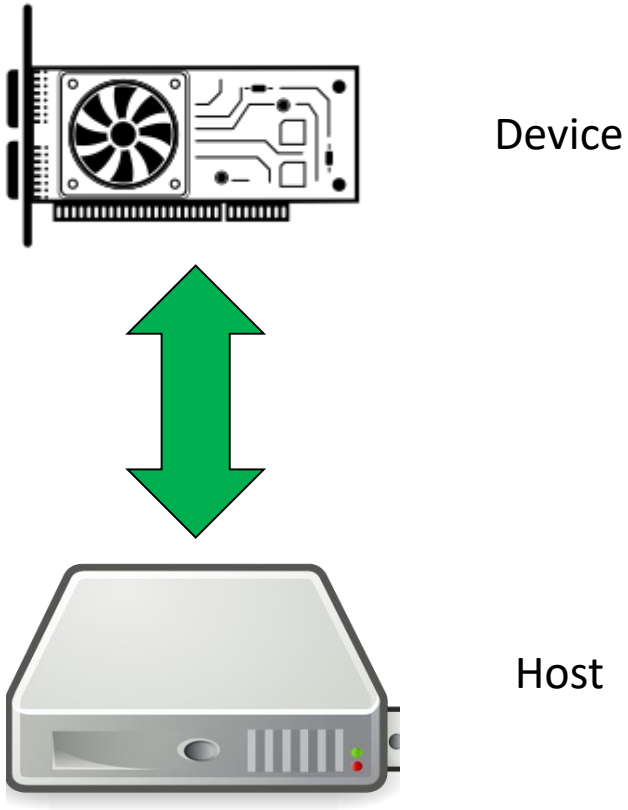


Working with GPUs

Recall from earlier in these slides, a GPU is attached to the computer by a PCIe bus.

It also has it's own memory (for example a V100 has 16/32 GB of HBM2 memory.

This means that for us to work with the GPU we need to allocate memory on the card (device) and then transfer data to and from the device.



Software view

In a bit more detail, at the top level, we have a master process which runs on the CPU and performs the following steps:

1. Initialises card.
2. Allocates memory on the host and on the device.
3. Copies data from the host memory to device memory.
4. Launches multiple instances of the execution “kernel” on the device.
5. Copies data from the device memory to the host.
6. Repeat 3-5 as needed.
7. De-allocates all memory and terminates.

Software view

In further detail, within the GPU:

- Each instance of the execution kernel executes on a SM.
- If the number of instances exceeds the number of SMs, then more than one will run at a time on each SM if there are enough registers and shared memory, and the others will wait in a queue and execute later.
- All threads within one instance can access local shared memory but can't see what the other instances are doing (even if they are on the same SM).
- ***There are no guarantees on the order in which the instances execute***.

Software view - CUDA

CUDA (*Compute Unified Device Architecture*) provides a set of programming extensions based on the C/C++ family of languages.

If you have a basic understanding of C and understand the concept of threads and SIMD execution, then CUDA is easy to pick up.

FORTRAN support is provided through a compiler from PGI (who are now owned by NVIDIA) and also the IBM XL compiler.

The language is now fairly mature, there is lots of example code available, good documentation, and a large user community on NVIDIA forums.

Installing CUDA

CUDA is supported on Windows, Linux and MacOSX.
https://developer.nvidia.com/cuda-downloads

Driver

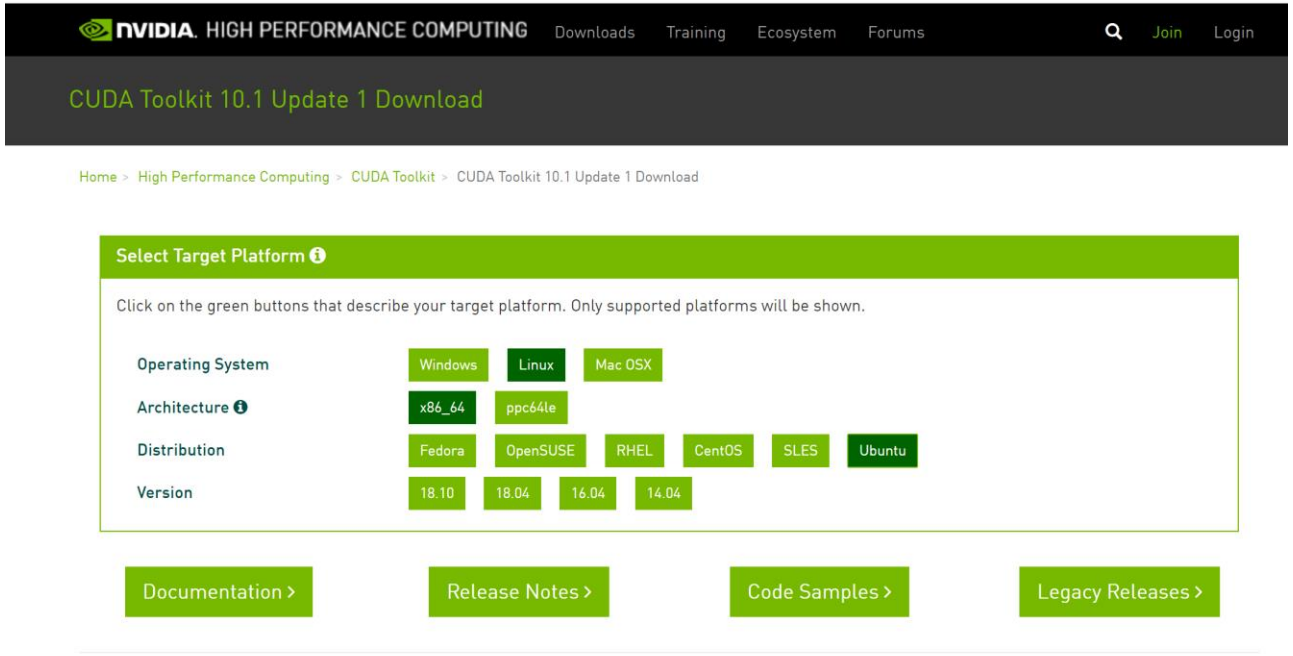
- Low-level software that controls the graphics card.

Toolkit

- nvcc CUDA compiler.
- Nsight IDE plugin for Eclipse or Visual Studio profiling and debugging tools.
- Several libraries (more to come on this).

SDK

- Lots of demonstration examples.
- Some error-checking utilities.
- Not officially supported by NVIDIA.
- Sparse documentation.



CUDA Programming

A CUDA program comes in two parts:

1. A host code that executes on the CPU which interfaces to the GPU.

2. Kernel code which runs on the GPU.

At the host level, there is a choice of 2 APIs (Application Programming Interfaces):

1. Runtime
   - simpler, more convenient

2. Driver
   - much more verbose
   - more flexible (e.g. allows run-time compilation)
   - closer to OpenCL

We will only use the runtime API in this course.

CUDA Programming – host code

At the host code level, there are library routines for:

- memory allocation on graphics card
- data transfer to/from device memory, including
  - constants
  - ordinary data
- error-checking
- timing

There is also a special syntax for launching multiple instances of the kernel process on the GPU.

```
// Allocate pointers for host and device memory
float *h_input, *h_output;
float *d_input, *d_output;

// malloc() host memory (this is in your RAM)
h_input  = (float*) malloc(mem_size);
h_output = (float*) malloc(mem_size);

// allocate device memory input and output arrays
cudaMalloc((void**)&d_input,  mem_size);
cudaMalloc((void**)&d_output, mem_size);

// Do something here!

// cleanup memory
free(h_input);
free(h_output);
cudaFree(d_input);
cudaFree(d_output);
```

CUDA Programming – host code

At the host code level, there are library routines for:

- memory allocation on graphics card
- data transfer to/from device memory, including
  - constants
  - ordinary data
- error-checking
- timing

There is also a special syntax for launching multiple instances of the kernel process on the GPU.

```
// Copy host memory to device input array
cudaMemcpy(d_input, h_input, mem_size, cudaMemcpyHostToDevice);

// Do something on the GPU

// copy result from device to host
cudaMemcpy(h_output, d_output, mem_size, cudaMemcpyDeviceToHost);
```

CUDA Programming – host code

At the host code level, there are library routines for:

- memory allocation on graphics card
- data transfer to/from device memory, including
  - constants
  - ordinary data
- error-checking
- timing

} ***Covered in practicals***

There is also a special syntax for launching multiple instances of the kernel process on the GPU…

```
__global__ void helloworld_GPU(void){
  printf("Hello world!\n");
}

int main(void) {

  // run CUDA kernel
  helloworld_GPU<<<1,1>>>();

  return (0);
}
```

CUDA Programming – host code

In its simplest form the special syntax looks like:

```
kernel_routine<<gridDim, blockDim>>(args);
```

`gridDim` is the number of instances of the kernel (the “grid” size).

`blockDim` is the number of threads within each instance (the “block” size).

`args` is a limited number of arguments, usually mainly pointers to arrays in graphics memory, and some constants which get copied by value.

The more general form allows `gridDim` and `blockDim` to be 2D or 3D to simplify application programs.

```
__global__ void helloworld_GPU(void){
  printf("Hello world!\n");
}

int main(void) {

  // run CUDA kernel
  helloworld_GPU<<<1,1>>>();

  return (0);
}
```

CUDA Programming

At the lower level, when one instance of the kernel is started on a SM it is executed by a number of threads, each of which knows about:

- some variables passed as arguments.
- pointers to arrays in device memory (also arguments).
- global constants in device memory.
- shared memory and private registers/local variables.

CUDA Programming

The CUDA language uses some reserved or special variables.
These are:

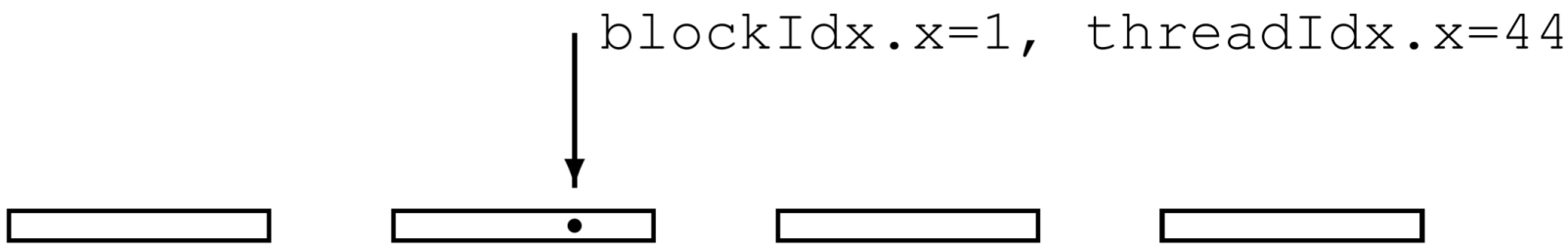
Variable	Example	Description
`gridDim`	`gridDim.x`	Size (or dimensions) of grid of blocks
`blockDim`	`blockDim.y`	Size (or dimensions) of each block
`blockIdx`	`blockIdx.z`	Index (or 2D/3D indices) of block
`threadIdx`	`threadIdx.y`	Index (or 2D/3D indices) of thread
`warpSize`		Currently 32 lanes (and has been so far)
`kernel<<<…>>>(args);`		Kernel launch

All can have an x, y or z component as in the examples listed.

CUDA Programming

Below is a conceptual example of a 1D grid, comprised of 4 blocks, each having 64 threads per block:

- `gridDim` = 4
- `blockDim` = 64
- `blockIdx` ranges from 0 to 3
- `threadIdx` ranges from 0 to 63



CUDA Programming

The kernel code looks fairly normal once you get used to two things:

Code is written from the point of view of a single thread…

- Quite different to OpenMP multithreading
- Similar to MPI, where you use the MPI “rank” to identify the MPI process
- All local variables are private to that thread

It’s important to think about where each variable lives (more on this in the next lecture)

- Any operation involving data in the device memory forces its transfer to/from registers in the GPU.
- It’s often better to copy the value into a local register variable

Our first host code

```
int main() {

    float *h_x, *d_x;                                                   // h=host, d=device.

    int nblocks=2, nthreads=8, nsize=2*8;                               // 2 blocks, 8 threads each.

    h_x = (float *)malloc(nsize*sizeof(float));                         // Allocate host memory.
    cudaMalloc((void **)&d_x, nsize*sizeof(float));                     // Allocate device memory.

    my_first_kernel<<<nblocks,nthreads>>>(d_x);                         // GPU kernel launch.

    cudaMemcpy(h_x,d_x,nsize*sizeof(float),cudaMemcpyDeviceToHost);     // Copy results back from GPU.

    for (int n=0; n<nsize; n++) printf(" n, x = %d %f \n",n,h_x[n]);    // Print the results.

    cudaFree(d_x); free(h_x);                                           // Free memory on host & device.
}
```

Our first kernel code

```
#include <helper_cuda.h>

__global__ void my_first_kernel(float *x)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    x[tid]  = (float) threadIdx.x;
}
```

- The `__global__` identifier says it's a kernel function.
- Each thread sets one element of the x array.
- Within each block of threads, `threadIdx.x` ranges from 0 to `blockDim.x-1`, so each thread has a unique value for `tid`.

Quick recap

Software

Hardware

A thread executes on a core.

Thread

FP32

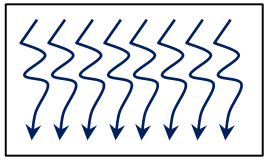
Core

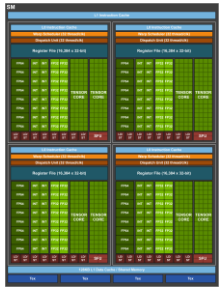
A group of threads, a thread block, comprised of groups of 32 threads (a warp), executes on a SM.

Thread blocks don't migrate between SMs.

Several concurrent thread blocks can reside on a SM.

Thread Block



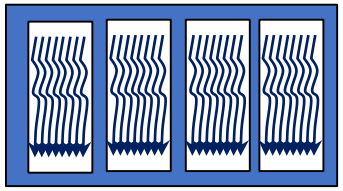


SM

A group of thread blocks form a grid and a grid runs on the device.

In CUDA this is a kernel launch.

Grid



Device

Scaling things up

Suppose we have 1000 blocks, and each one has 128 threads – how would this get executed?

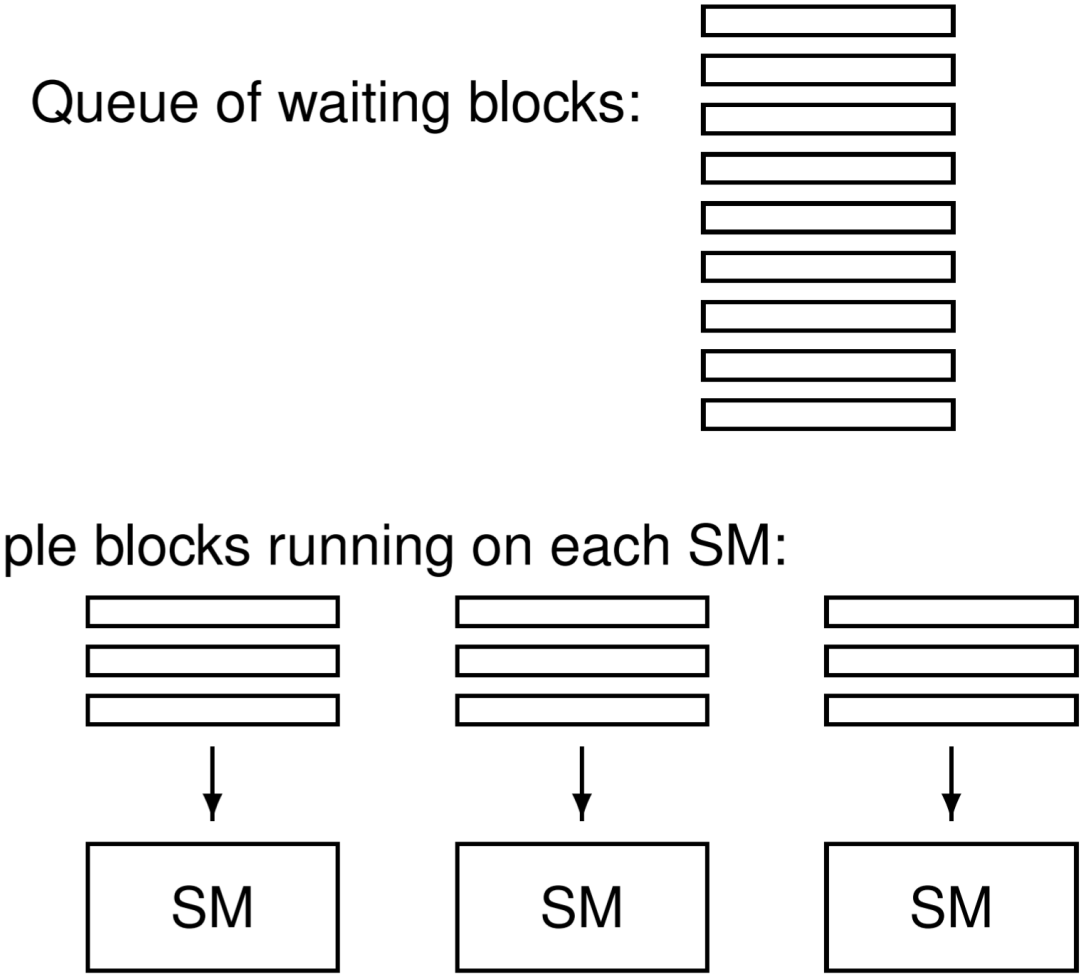
On the Kepler that we will use in our practicals, we would probably get 8-12 blocks running on each SM (Kepler SMs have 128 cores), and each block has 4 warps, so 32-48 warps running on each SM.

Each clock tick, the SM warp scheduler decides which warps to execute next, choosing from those not waiting for:

- data coming from device memory (memory latency)
- completion of earlier instructions (pipeline delay)

As a programmer, we don't need to worry about this level of detail, we just need to ensure there are lots of threads / warps.

Scaling things up



Quick recap

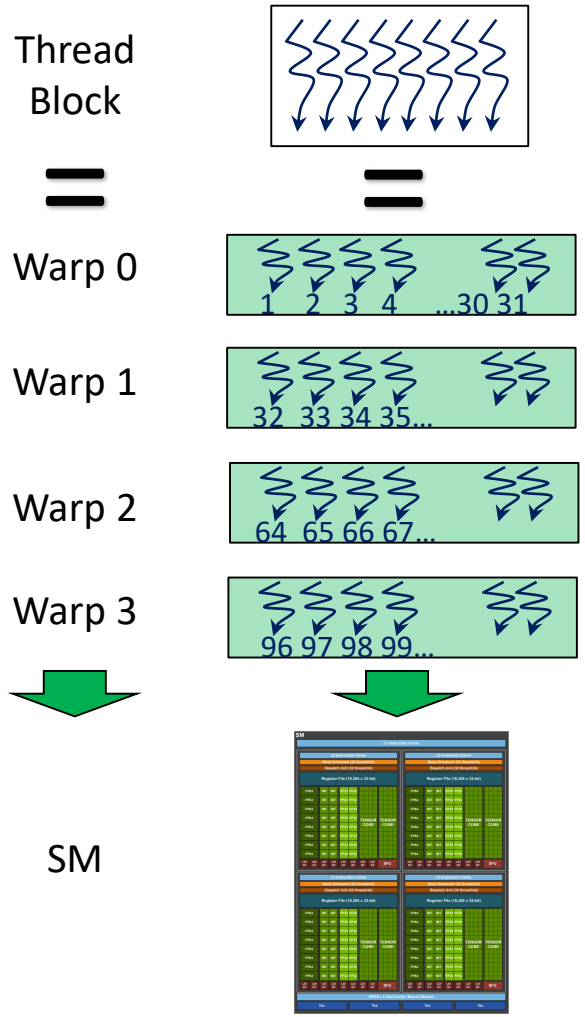
Thread blocks are formed from warps.

The warp is executed in parallel on the SM.

By this we mean that everything that happens within a warp is lock-step.

So the same operation (instruction) in thread 2, 3, 8 occurs in thread 7, 1, 4, 11… (from 0 to 31) at the same time.

So we have a Single Instruction Multiple Thread (SIMT) architecture.



Higher dimensions

So far, out simple example considers the case of a 1D grid of blocks, and within each block a 1D set of threads.

Many applications – Finite Element, CFD, MD,…., might need to use 2D or even 3D sets of threads.

As mentioned previously, if we want to use a 2D set of threads, then
`blockDim.x, blockDim.y` give the dimensions, and
`threadIdx.x, threadIdx.y` give the thread indices

and to launch the kernel we would use something like:

```
dim3 nblocks(2,3);                                  // 2 blocks in x, 3 blocks in y
dim3 nthreads(16,4);                                // 16 threads in x, 4 threads in y
my_new_kernel<<<nblocks, nthreads>>>(d_x);
```

Here, `dim3` is a special CUDA datatype with 3 components .x,.y,.z each initialised to 1.

Indexing in higher dimensions

To calculate a unique (or 1D) thread identifier (previously we called this `tid`) when working in 2D or 3D we simply use:

```
tid = threadIdx.x + threadIdx.y * blockDim.x + threadIdx.z * blockDim.x * blockDim.y;
```

and this is then broken up into warps of size 32.
How do 2D / 3D threads get divided into warps?
1D thread ID defined by

Mikes notes on Practical 1

- start from code shown above (but with comments)
- learn how to compile / run code within Nsight IDE (integrated into Visual Studio for Windows, or Eclipse for Linux)
- test error-checking and printing from kernel functions
- modify code to add two vectors together (including sending them over from the host to the device)
- if time permits, look at CUDA SDK examples

Practical 1

Things to note:

- memory allocation

```
cudaMalloc((void **)&d_x, nbytes);
```

- data copying

```
cudaMemcpy(h_x,d_x,nbytes,
                cudaMemcpyDeviceToHost);
```

- reminder: prefix `h_` and `d_` to distinguish between arrays on the host and on the device is not mandatory, just helpful labelling
- kernel routine is declared by `__global__` prefix, and is written from point of view of a single thread

Practical 1

Second version of the code is very similar to first, but uses an SDK header file for various safety checks – gives useful feedback in the event of errors.

- check for error return codes:
  `checkCudaErrors( ... );`

- check for kernel failure messages:
  `getLastCudaError( ... );`

Practical 1

One thing to experiment with is the use of `printf` within a CUDA kernel function:

- essentially the same as standard `printf`; minor difference in integer return code
- each thread generates its own output; use conditional code if you want output from only one thread
- output goes into an output buffer which is transferred to the host and printed later (possibly much later?)
- buffer has limited size (1MB by default), so could lose some output if there's too much
- need to use either `cudaDeviceSynchronize();` or `cudaDeviceReset();` at the end of the main code to make sure the buffer is flushed before termination

Practical 1

The practical also has a third version of the code which uses “managed memory” based on Unified Memory.

In this version

- there is only one array / pointer, not one for CPU and another for GPU
- the programmer is not responsible for moving the data to/from the GPU
- everything is handled automatically by the CUDA run-time system

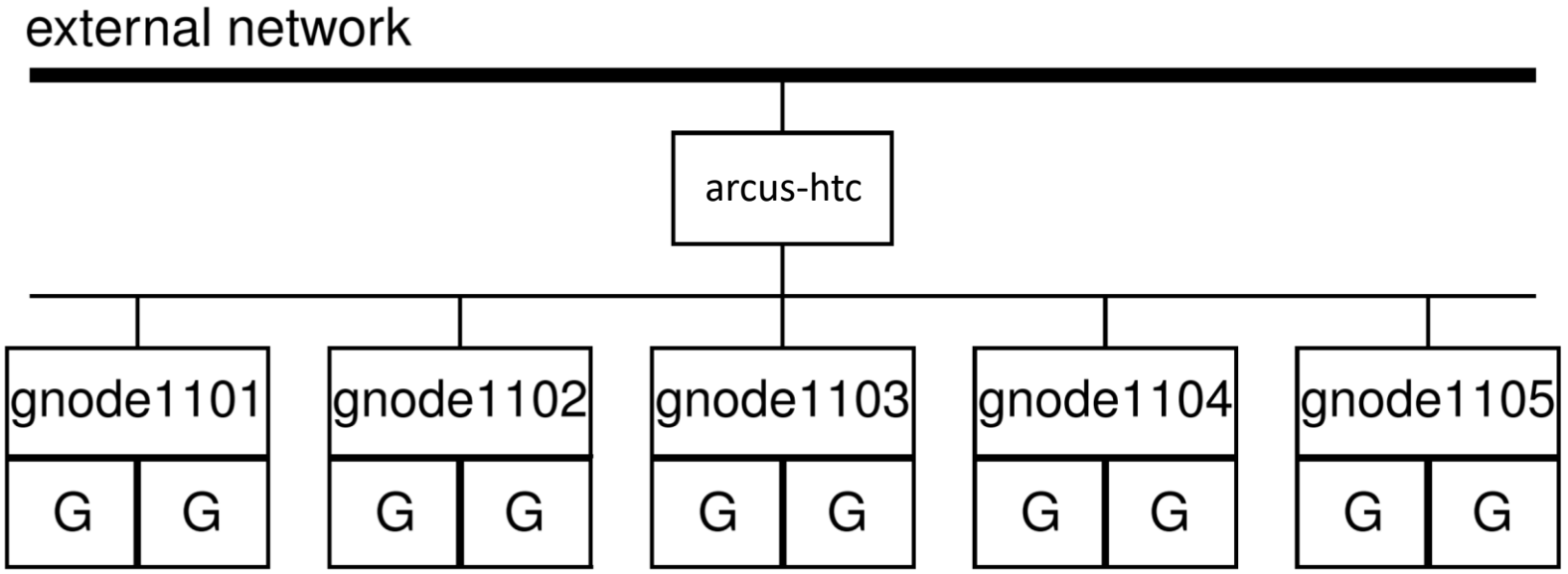
Practical 1

This leads to simpler code, but it's important to understand what is happening because it may hurt performance:

- if the CPU initialises an array x, and then a kernel uses it, this forces a copy from CPU to GPU
- if the GPU modifies x and the CPU later tries to read from it, that triggers a copy back from GPU to CPU

Personally, I prefer to keep complete control over data movement, so that I know what is happening and I can maximise performance.

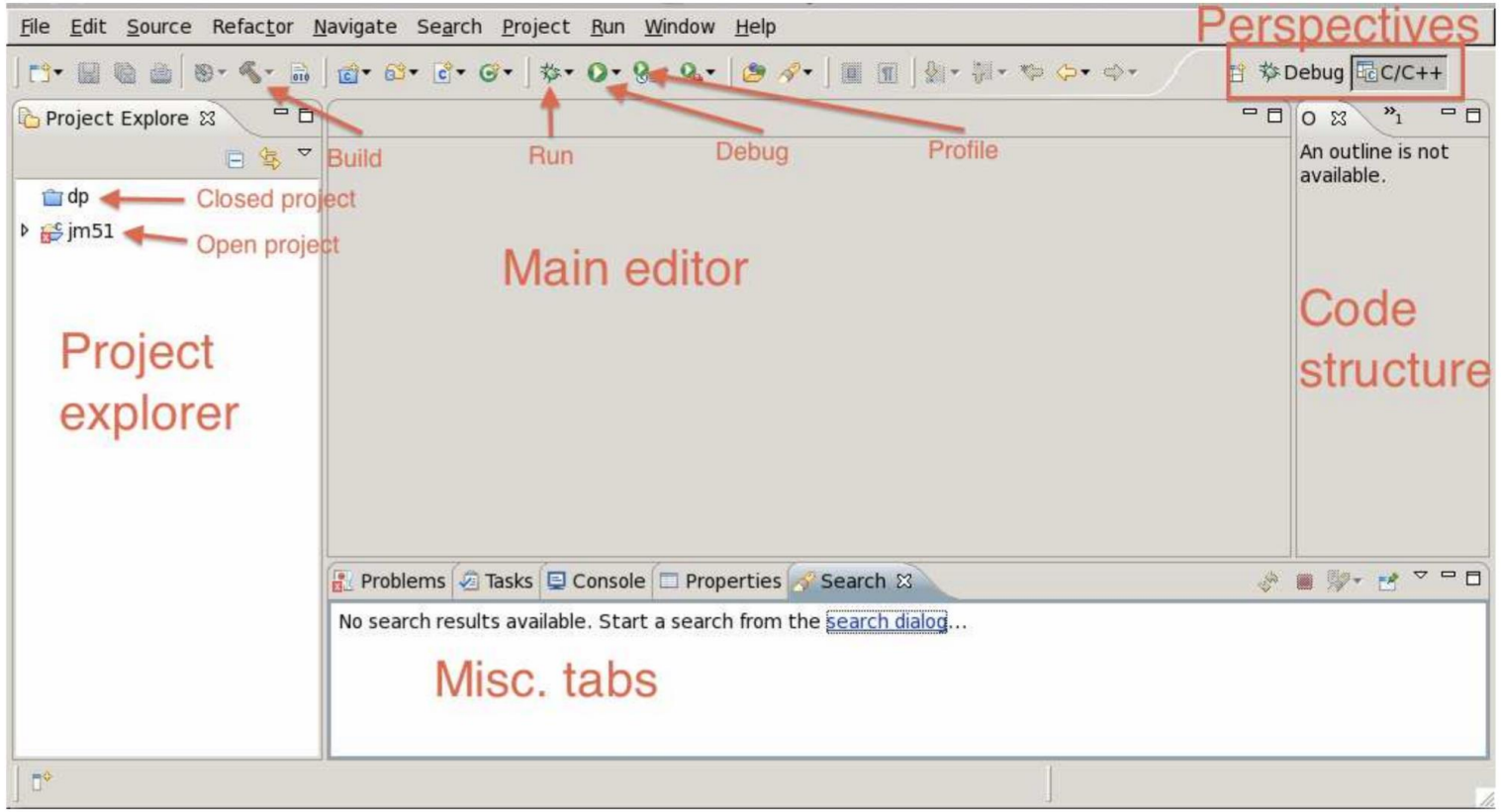
Arcus htc



- `arcus-htc.arc.ox.ac.uk` is the head node.
- The GPU compute nodes have two K80 cards with a total of 4 GPUs, numbered 0 – 3.
- Read the Arcus notes before starting the practical.

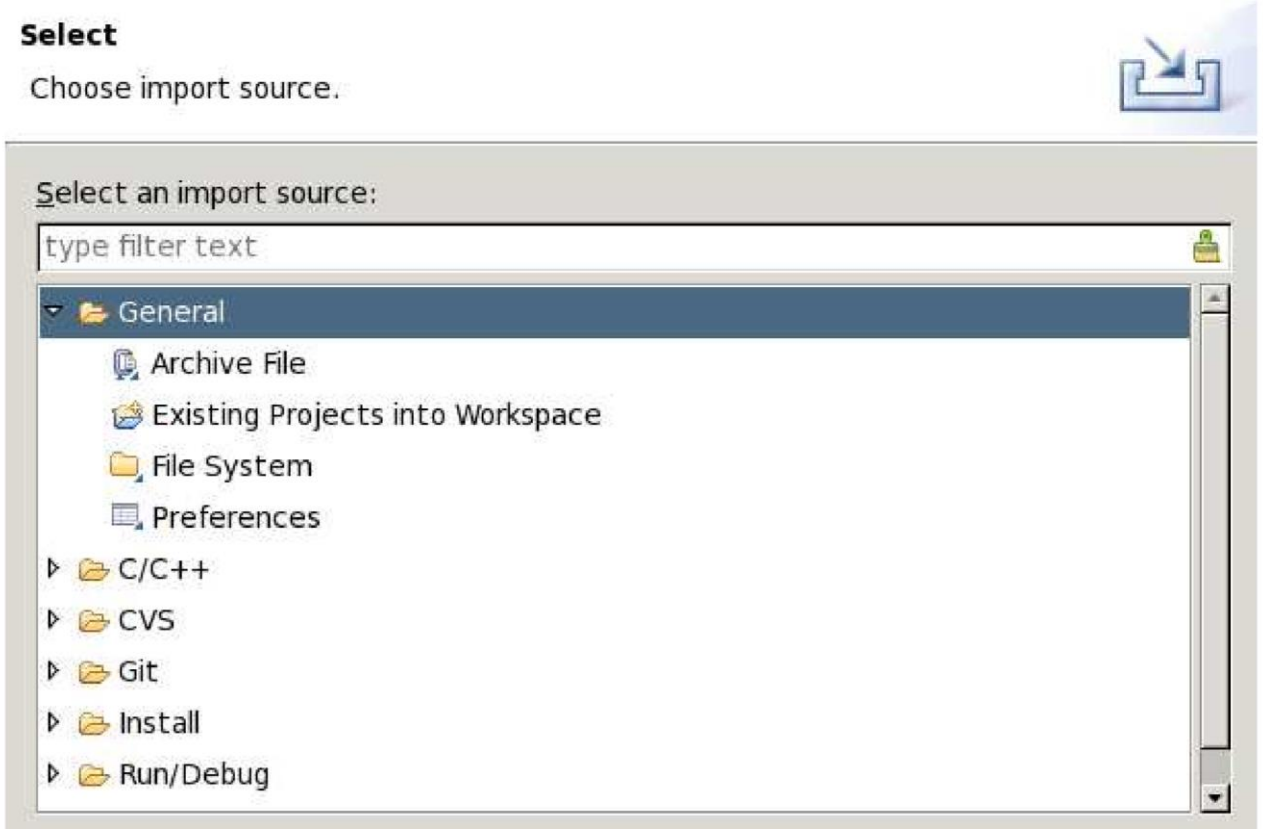
Nsight

General view:

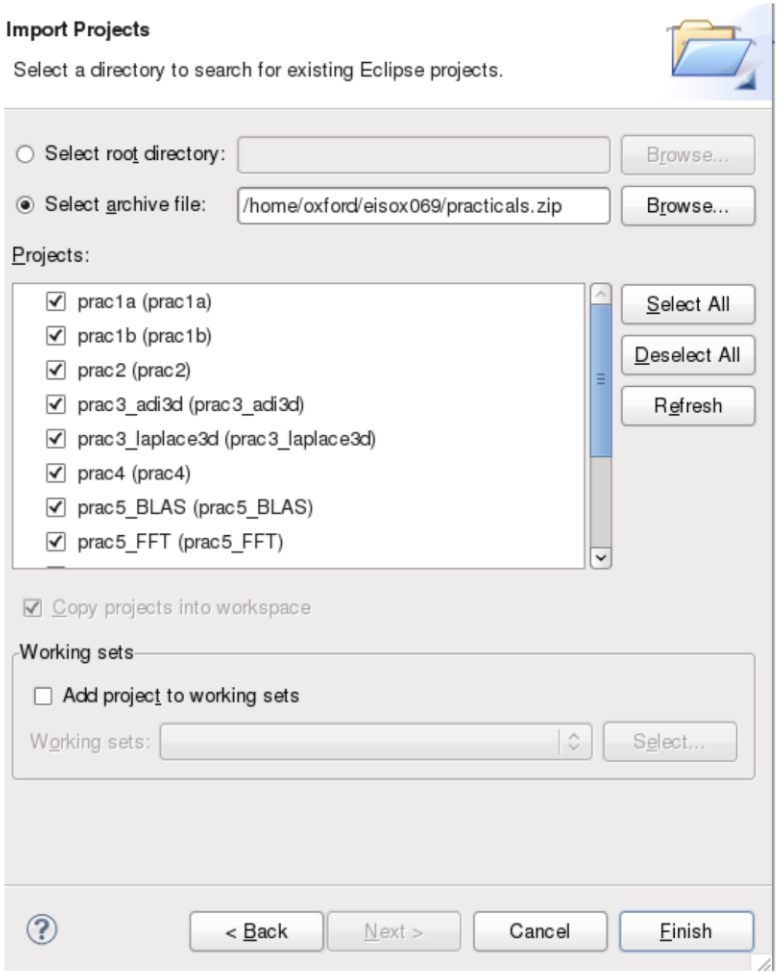


Nsight

Importing the practicals: select General – Existing Projects



Nsight


Import Projects
Select a directory to search for existing Eclipse projects.
Select root directory:
Browse...
Select archive file:
/home/oxford/eisox069/practicals.zip
Browse...
Projects:
prac1a (prac1a)
prac1b (prac1b)
prac2 (prac2)
prac3_adi3d (prac3_adi3d)
prac3_laplace3d (prac3_laplace3d)
prac4 (prac4)
prac5_BLAS (prac5_BLAS)
prac5_FFT (prac5_FFT)
Select All
Deselect All
Refresh
Copy projects into workspace
Working sets
Add project to working sets
Working sets:
Select...
< Back
Next >
Cancel
Finish

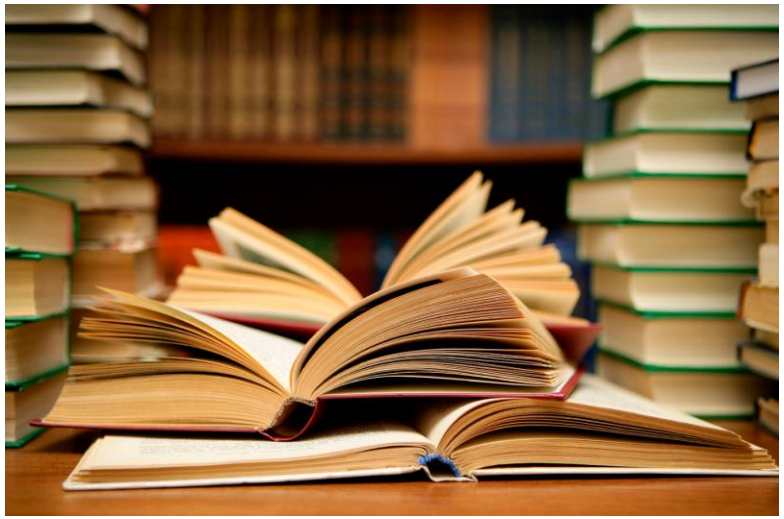
Key reading

CUDA Programming Guide, version 10.1:

- Chapter 1: Introduction
- Chapter 2: Programming Model
- Chapter 5: performance of different GPUs
- Appendix A: CUDA-enabled GPUs
- Appendix B, sections B.1 – B.4: C language extensions
- Appendix B, section B.20: Formatted (printf) output
- Appendix H, section H: Compute capabilities (features of different GPUs)

Wikipedia (clearest overview of NVIDIA products):

- https://en.wikipedia.org/wiki/Nvidia_Tesla
- https://en.wikipedia.org/wiki/GeForce_10_series
- https://en.wikipedia.org/wiki/GeForce_20_series



https://www.flickr.com/photos/abee5/8314929977

What have we learnt?

In this lecture you have learnt about the usage of GPUs in HPC. We have looked at different hardware generations of GPUs and some of there differences.

We've looked at the GPU architecture and how they execute code.

Finally we've looked at the CUDA programming language and how to create a basic host and device code.