Lecture 1: an introduction to CUDA

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

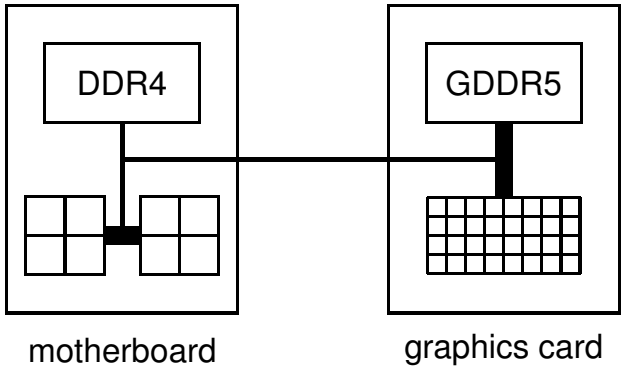
Oxford e-Research Centre

Overview

- hardware view
- software view
- CUDA programming

Hardware view

At the top-level, a PCIe graphics card with a many-core GPU and high-speed graphics "device" memory sits inside a standard PC/server with one or two multicore CPUs:



Hardware view

There are multiple products in the current Maxwell generation, but none with good double precision performance, or designed for HPC

Consumer graphics cards (GeForce):

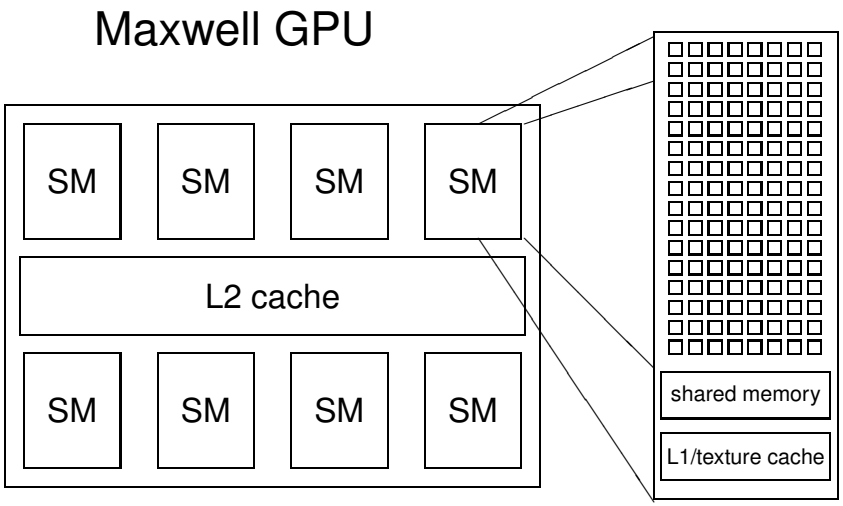
- GTX 960: 1024 cores, 2GB (£170)
- GTX 980: 2048 cores, 4GB (£450)
- GTX Titan X: 3076 cores, 12GB (£1000)

Hardware view

- building block is a “streaming multiprocessor” (SM):
  - 128 cores and 64k registers
  - 64-96KB of shared memory
  - 12-48KB L1 cache / read-only texture cache
  - 10KB cache for constants
  - up to 2K threads per SM
- different chips have different numbers of these SMs:

product	SMs	bandwidth	memory	power
GTX 960	8	112 GB/s	2 GB	120W
GTX 980	16	224 GB/s	4 GB	165W
GTX Titan X	24	336 GB/s	12 GB	250W

Hardware View



Hardware view

There are multiple products in the Kepler generation

Consumer graphics cards (GeForce):

- GTX Titan Black: 2880 cores, 6GB (£600)
- GTX Titan Z: 2×2880 cores, 2×6GB (£1200)

HPC cards (Tesla):

- K80: 2×2496 cores, 2×12GB (£3.5k)

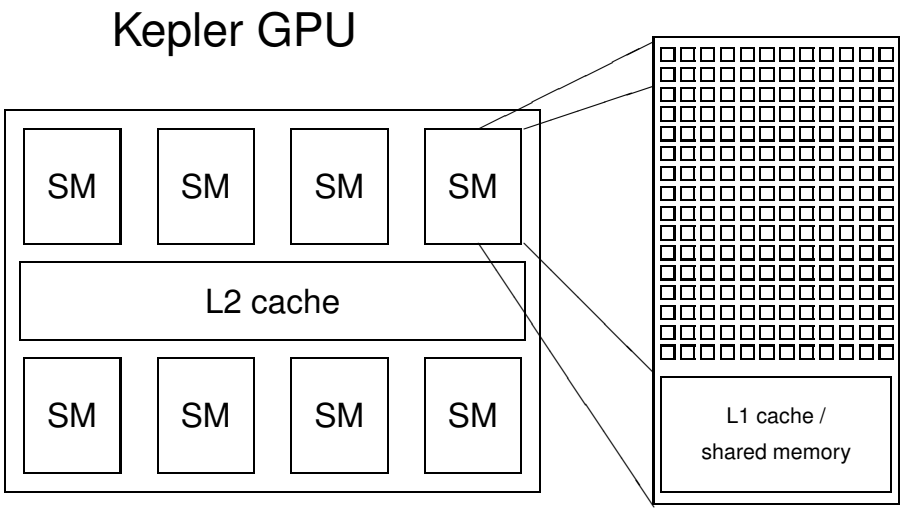
These cards have the best double precision capability available today

Hardware view

- building block is a “streaming multiprocessor” (SM):
  - 192 cores and 64k registers
  - 64KB of shared memory / L1 cache
  - 8KB cache for constants
  - 48KB texture cache for read-only arrays
  - up to 2K threads per SM
- different chips have different numbers of these SMs:

product	SMs	bandwidth	memory	power
GTX Titan Black	15	336 GB/s	6 GB	250W
GTX Titan Z	2×15	2×336 GB/s	2×6 GB	375W
K80	2×14	2×240 GB/s	2×12 GB	300W

Hardware View

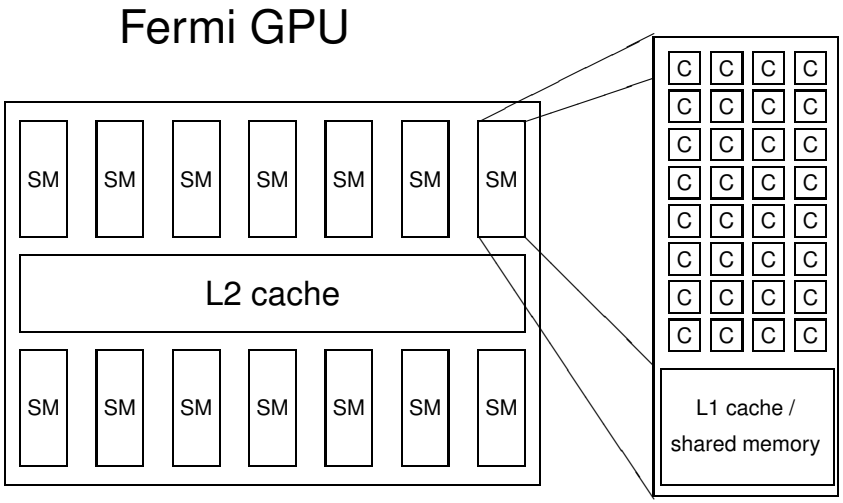


Hardware view

- older Fermi GPU has SM with
  - 32 cores and 32k registers
  - 64KB of shared memory / L1 cache
  - 8KB cache for constants
  - up to 1536 threads per SM
- different chips have different numbers of these SMs:

product	SMs	bandwidth	memory
GTX 560	14	130 GB/s	1/2 GB
GTX 580	16	190 GB/s	1.5 GB
M2050/2070	14	140 GB/s	3/6 GB
M2075/2090	16	140 GB/s	3/6 GB

Hardware View



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
- natural for graphics processing and much scientific computing
- SIMT is also a natural choice for many-core chips to simplify each core

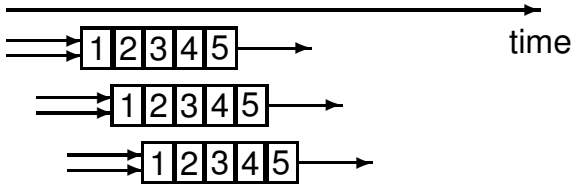
Multithreading

Lots of active threads is the key to high performance:

- no “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

- for each thread, one operation completes before the next starts – avoids the complexity of pipeline overlaps which can limit the performance of modern processors



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

Software view

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 in host and on device
3. copies data from host to device memory
4. launches multiple instances of execution “kernel” on device
5. copies data from device memory to host
6. repeats 3-5 as needed
7. de-allocates all memory and terminates

Software view

At a lower level, 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

CUDA

CUDA (Compute Unified Device Architecture) is NVIDIA's program development environment:

- based on C/C++ with some extensions
- FORTRAN support provided by compiler from PGI (now owned by NVIDIA)
- lots of example code and good documentation – fairly short learning curve for those with experience of OpenMP and MPI programming
- large user community on NVIDIA forums

CUDA Components

Installing CUDA on a system, there are 3 components:

- 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
- SDK
  - lots of demonstration examples
  - some error-checking utilities
  - not officially supported by NVIDIA
  - almost no documentation

CUDA programming

Already explained that a CUDA program has two pieces:

- host code on the CPU which interfaces to the GPU
- kernel code which runs on the GPU

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

- runtime
  - simpler, more convenient
- 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, and that is all I use in my own research.

CUDA programming

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

- memory allocation on graphics card
- data transfer to/from device memory
  - constants
  - texture arrays (useful for lookup tables)
  - ordinary data
- error-checking
- timing

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

CUDA programming

In its simplest form it 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

CUDA programming

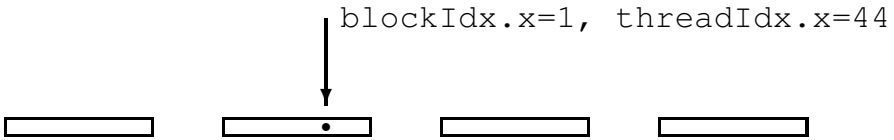
At the lower level, when one instance of the kernel is started on a SM it is executed by a number of threads, each of which knows about:

- 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
- some special variables:
  - `gridDim` size (or dimensions) of grid of blocks
  - `blockDim` size (or dimensions) of each block
  - `blockIdx` index (or 2D/3D indices) of block
  - `threadIdx` index (or 2D/3D indices) of thread

CUDA programming

1D grid with 4 blocks, each with 64 threads:

- `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
- need 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
  - often better to copy the value into a local register variable

Host code

```
int main(int argc, char **argv) {
 float *h_x, *d_x;         //  h=host, d=device
 int   nblocks=2, nthreads=8, nsize=2*8;

 h_x = (float *)malloc(nsize*sizeof(float));
 cudaMalloc((void **)&d_x,nsize*sizeof(float));

 my_first_kernel<<<nblocks,nthreads>>>(d_x);

 cudaMemcpy(h_x,d_x,nsize*sizeof(float),
            cudaMemcpyDeviceToHost);

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

 cudaFree(d_x); free(h_x);
}
```


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;
}
```

- `__global__` identifier says it's a kernel function
- each thread sets one element of `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`

CUDA programming

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

On Kepler hardware, would probably get 8-12 blocks running at the same time on each SM, and each block has 4 warps $\Longrightarrow$ 32-48 warps running on each SM

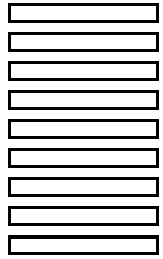
Each clock tick, SM warp scheduler decides which warps to execute next, choosing from those not waiting for

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

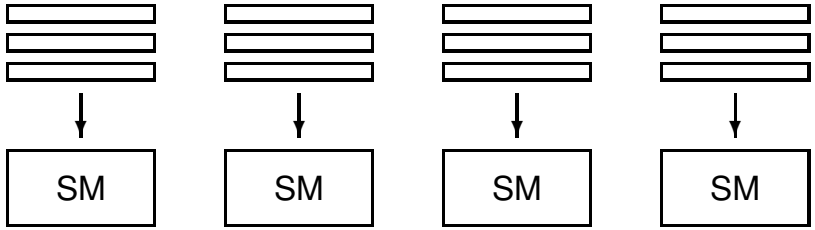
Programmer doesn't have to worry about this level of detail, just make sure there are lots of threads / warps

CUDA programming

Queue of waiting blocks:



Multiple blocks running on each SM:



CUDA programming

In this simple case, we had a 1D grid of blocks, and a 1D set of threads within each block.

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 nthreads(16,4);
my_new_kernel<<<nblocks,nthreads>>>(d_x);
```

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

CUDA programming

A similar approach is used for 3D threads and 2D / 3D grids; can be very useful in 2D / 3D finite difference applications.

How do 2D / 3D threads get divided into warps?

1D thread ID defined by

```
threadIdx.x +
threadIdx.y * blockDim.x +
threadIdx.z * blockDim.x * blockDim.y
```

and this is then broken up into warps of size 32.

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.

(This is only supported on Kepler and later hardware.)

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

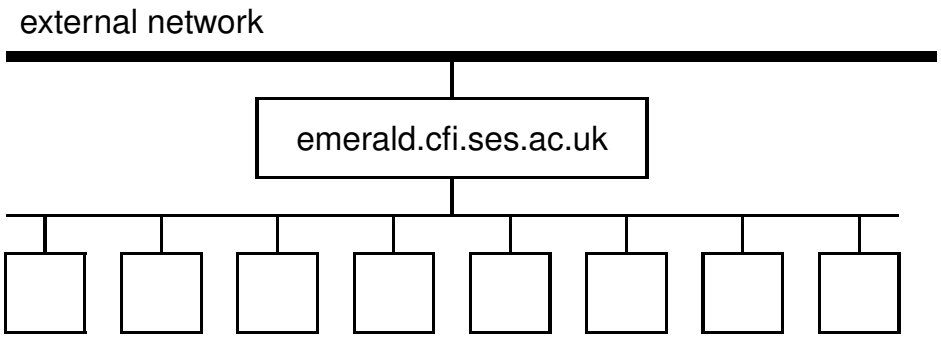
Practical 1

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

- if a kernel uses an array x, then this probably forces a copy from CPU to GPU, even if the data hasn't changed since the previous transfer
- when there is a `cudaDeviceSynchronize();` synchronisation, all managed data is probably transferred back to the CPU, even if it hasn't changed or is not needed

Personally, I prefer to keep complete control over data movement, so that I can maximise performance.

Emerald cluster



The 3 front-end nodes have Quadro 4000 GPUs – Quadro is NVIDIA's series of cards for professional graphics applications (e.g. video editing, CAD, scientific visualisation)

Can do some limited code development on these, but better to work on the back-end compute nodes.

Emerald cluster

Emerald has three kinds of compute nodes:

- `cn3g*.gpu.rl.ac.uk (*=01-08,13-60)`
  have 3 Fermi M2090 GPUs
- `cn8g*.gpu.rl.ac.uk (*=01-24)`
  have 8 Fermi M2090 GPUs
- `cn3g*.gpu.rl.ac.uk (*=09-12)`
  have 3 Kepler K20 GPUs

Read the Emerald notes before starting the practicals

Note that you will need to use the Kepler nodes for practicals with managed memory (Practical 1) or shuffle instructions (Practicals 4, 7, 8)

Key reading

CUDA Programming Guide, version 7.0:

- Chapter 1: Introduction
- Chapter 2: Programming Model
- Appendix A: CUDA-enabled GPUs
- Appendix B, sections B.1 – B.4: C language extensions
- Appendix B, section B.17: `printf` output
- Appendix G, section G.1: features of different GPUs