Lecture 1: an introduction to CUDA

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Overview

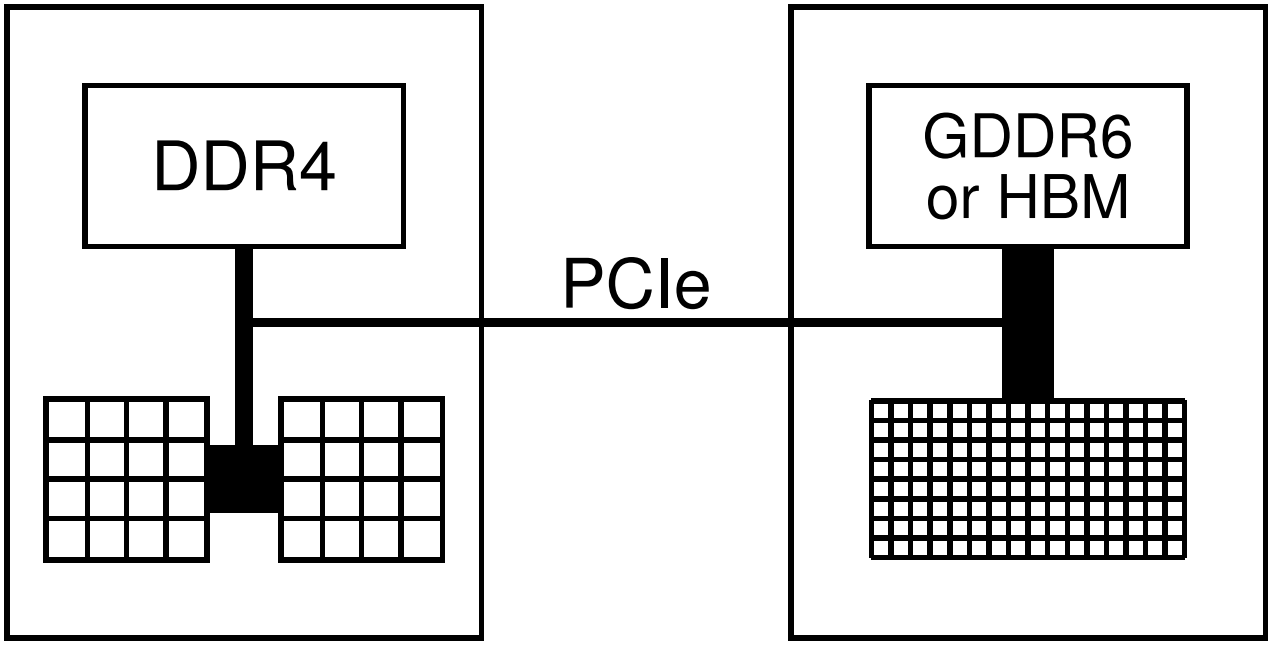
- hardware view
- software view
- CUDA programming
- first practical

Course materials are available at:

https://people.maths.ox.ac.uk/gilesm/cuda/

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

Currently, 3 generations of professional/HPC cards with excellent double precision (DP) capabilities, and special "tensor cores" for AI/ML:

- Volta (compute capability 7.0):
  - V100 released in 2018
- Ampere (compute capability 8.0):
  - A100 released in 2020
    we will use these for the practicals
  - smaller A30 released later
  - A2, A10, A16, A40 (CC 8.6) for inference and Virtual Desktop
- Hopper (compute capability 9.0):
  - H100 announced, due to ship in 2023

Hardware view

In addition there are consumer/gaming cards with excellent single precision (SP) capabilities and ray tracing support, but much poorer on DP and without “tensor cores” for AI/ML

- Ampere (compute capability 8.6):
  - GeForce RTX 3080
  - GeForce RTX 3080 Ti
  - GeForce RTX 3090
  - GeForce RTX 3090 Ti
- Hopper – announced but not yet shipping
  - GeForce RTX 4080
  - GeForce RTX 4080 Ti
  - GeForce RTX 4090

Hardware view

The Ampere A100s come in 4 versions:

- 40GB or 80GB of HBM2e very-high speed memory
- PCIe or SXM packaging: SXM is for special GPU servers with up to 16 A100s, and also supports higher clock frequencies

All 4 versions have a total of 3456 FP64 cores, and 6912 FP32/integer cores

Hardware view

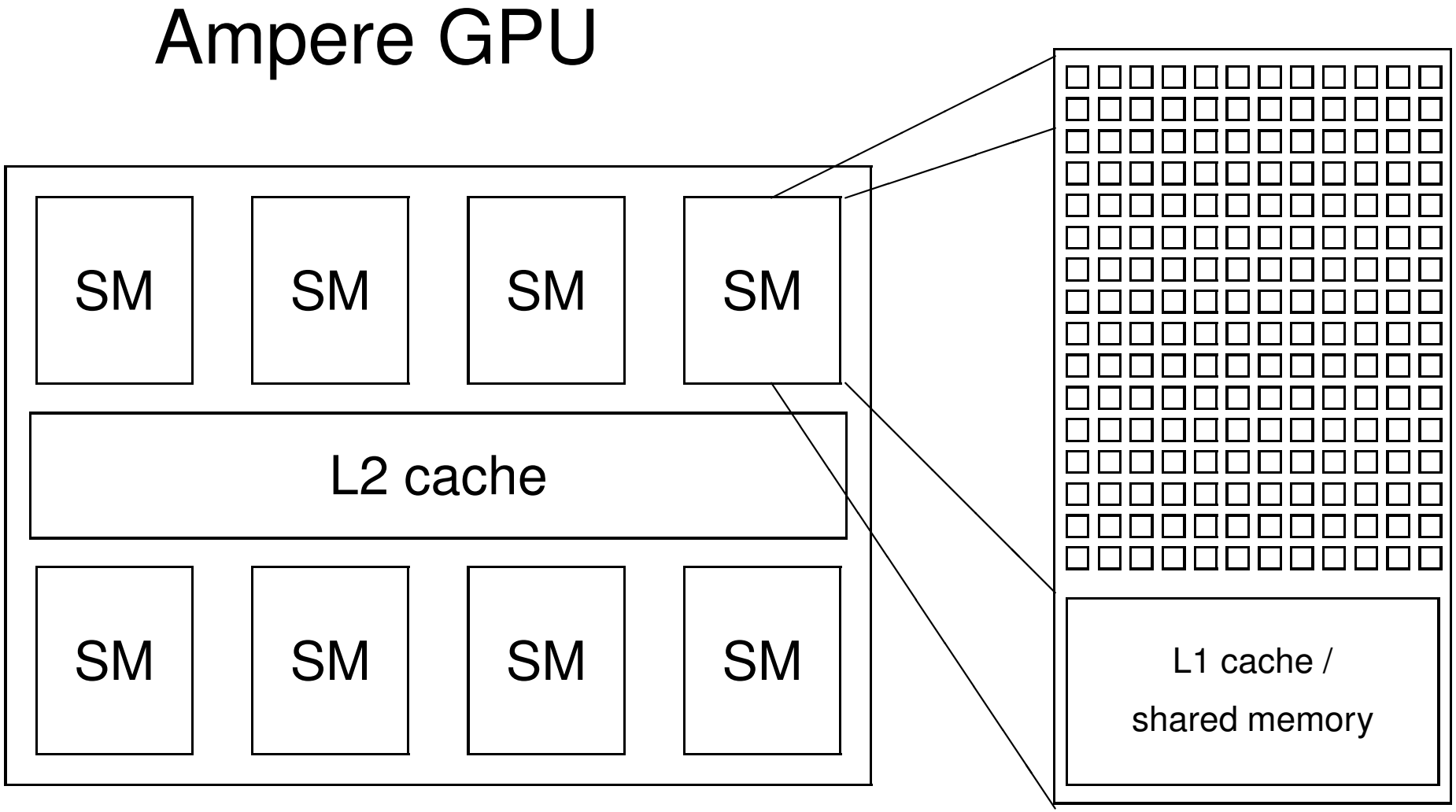
The key building block in an NVIDIA GPUs is a “streaming multiprocessor” (SM) – the A100 has 108 of them each with:

- 32 FP64 cores + 64 FP32 cores + 64 INT32 cores
- 64k registers
- 192KB of shared memory/L1 cache
- up to 2K threads per SM

In addition the A100 has:

- 40MB of L2 cache
- bandwidth of 1.6TB/s to external HBMe memory

Hardware View



Remember it really has 108 SMs!

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 AVX vectorisation on Intel Xeons
- 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

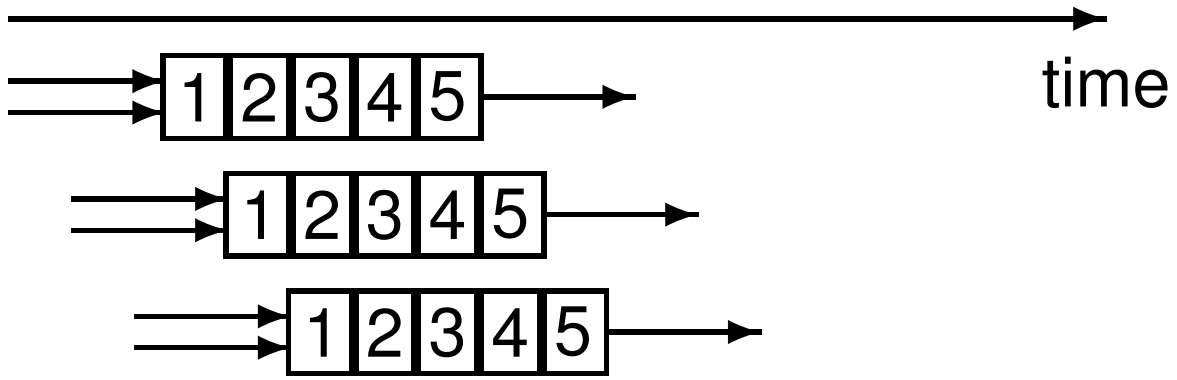
Multithreading

Lots of active threads is the key to high performance:

- no “context switching”; each thread has its own registers (up to 255 of them), 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



however, NVIDIA have now relaxed this, so each thread can have multiple independent instructions overlapping

- 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?

Software view

At the top level, we have a main 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 (or copy) of the 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 (on the GPU) and run 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 is NVIDIA's program development environment:

- based on C/C++ with some extensions
- Fortran support also available
- lots of sample codes 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 (currently on version 11.8)
  - `nvcc` CUDA compiler
  - Nsight plugin for Eclipse or Visual Studio
  - profiling and debugging tools
  - lots of libraries

In addition NVIDIA makes available lots of sample codes in a GitHub repository

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):

- run-time
  - simpler, more convenient
- driver
  - much more verbose, more flexible (e.g. allows run-time compilation), closer to OpenCL

We will only use the run-time 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
  - 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
  - `warpSize` always 32 so far, but could change

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

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 Ampere 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

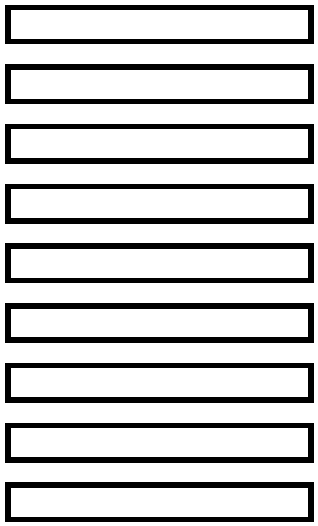
Each clock tick, SM warp scheduler decides which warps to execute next, choosing from those not waiting for

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

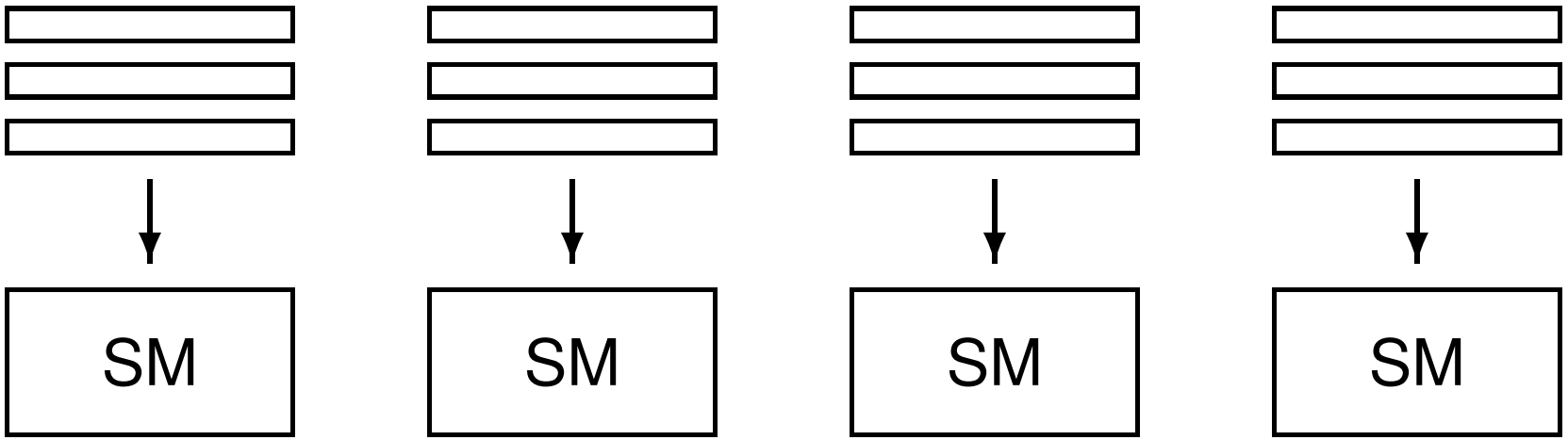
Programmer doesn't have to worry about this level of detail, just make sure there are lots of threads / warps

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)
- 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 samples

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 a 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.

Key reading

CUDA Programming Guide:

- Chapter 1: Introduction
- Chapter 2: Programming Model
- Section 5.4: performance of different GPUs
- Appendix A: CUDA-enabled GPUs
- Appendix B, sections B.1 – B.4: C language extensions
- Appendix B, section B.32: `printf` output
- Appendix K, section K.1: features of different GPUs