# Introduction to GPU hardware and to CUDA

Philip Blakely

Laboratory for Scientific Computing, University of Cambridge

# Course outline

- Introduction to GPU hardware structure and CUDA programming model
- Writing and compiling a simple CUDA code
- Example application: 2D Euler equations
- Optimization strategies and performance tuning
- In-depth coverage of CUDA features
- Combining CUDA with MPI.

# Outline


1. CUDA History
2. CUDA Hardware Model
3. Terminology

# References

- http://www.nvidia.com/cuda - Official site
- http://developer.nvidia.com/cuda-downloads - Drivers, Compilers, Manuals
- On CSC network: /lsc/opt/cuda-11.3/doc/pdf
  - https://docs.nvidia.com/cuda/
  - CUDA C Programming Guide
  - CUDA C Best Practices Guide
- Programming Massively Parallel Processors, David B. Kirk, and Wen-mei W. Hwu (2010)
- CUDA by Example, Jason Sanders and Edward Kandrot (2010)
- The CUDA Handbook, Nicholas Wilt (2013)

# Course caveats

- Only applicable to NVIDIA GPUs
- Assume CUDA-enabled graphics card compute capability $\geqslant 3.0$.
- Material relevant to all CUDA-enabled GPUs
- Assume good knowledge of C/C++, including simple optimization
- Some knowledge of C++ (simple classes and templates)
- Assume basic knowledge of Linux command-line
- Some knowledge of compilation useful
- Experience of parallel algorithms and issues is useful
- Linux is assumed (although CUDA is available under Windows and Mac OS)

# The power of GPUs

- All PCs have dedicated graphics cards.
- Usually used for realistic rendering in games, visualization, etc.
- However, when programmed effectively, GPUs can provide a lot of computing power:

**Live demos**

- fluidGL
- particles
- Mandelbrot

# Why should we use GPUs?

- Can provide large speed-up over CPU performance
- For some applications, a factor of 100 (although that was comparing a single Intel core with a full NVIDIA GPU).
- More typical improvement around $\times 10$ when compared to 8-core Intel processor.
- Relatively cheap - a few hundred pounds - NVIDIA are in gamers' market
- Widely available
- Easy to install

# Improved GFLOPS

- Theoretical GFLOPS are higher than Intel's processors (not entirely clear which processor is being compared to)

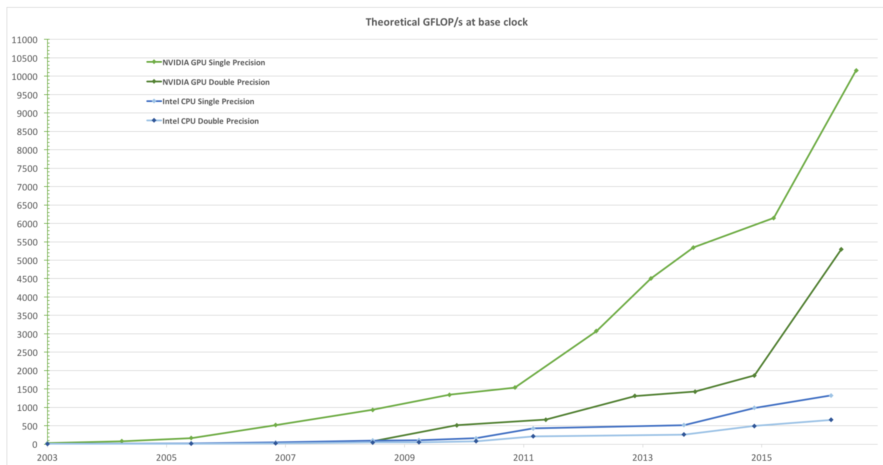



Diagram Copyright NVIDIA

# Improved bandwidth

- In order to get good performance out of any processor, we need high bandwidth as well:

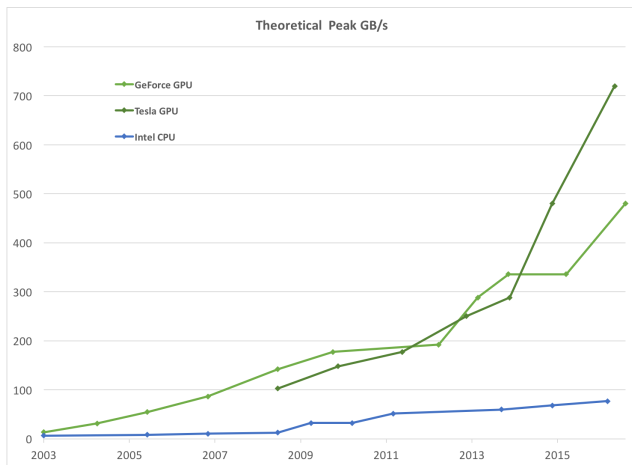



Diagram Copyright NVIDIA

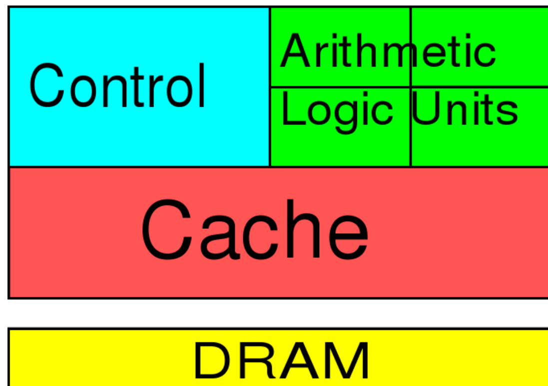
# Why shouldn't we use GPUs?

- Speed up with double precision less good than single
- Factor of up to 32 difference (very GPU-model dependent) in performance as opposed to CPU difference of factor 2
- Not suitable for all problems - depends on algorithm (array or matrix operations good, search/sort operations less suited)
- Can require a lot of work to get very good performance
- May be better to use general multi-core processors with MPI or OpenMP, or even OpenCL, SYCL, DPC++(GPU agnostic)
- Current programming models do not protect the programmer from themselves (even less than C++)
- For some applications, GPU may only give 2.5x advantage over a fully-used multi-core CPU.
- A commercial grade GPU will cost around £2,500, around the same as a 16-core Intel Xeon server.

# Brief history of GPUs

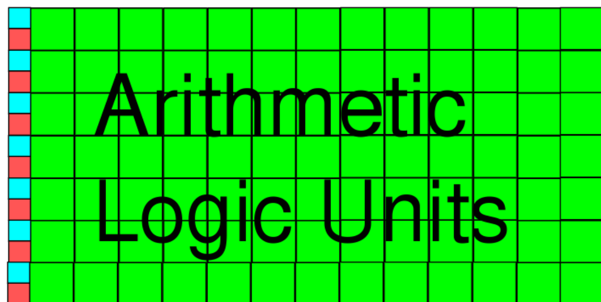
- Graphics cards (for home PCs) first developed in 1990s
  - Handle graphics rendering, leaving CPU to do complex work
  - CPU hands over a scene for rendering to graphics card
- Early 2000s, get more complex graphics ops. and programmable.
  - APIs such as OpenGL and Microsoft's DirectX
  - Scientific Computing researchers start performing calcs. using built-in vector ops to do useful work.
- Late 2000s, Graphics card companies developed ways to generally program GPUs
  - Advent of GPGPUs and general programming languages, e.g. CUDA (NVIDIA - 2006)
  - Super computing with GPUs takes off
- 2010-2021 - NVIDIA progressively improve their GPUs, adding extra instructions, memory caches, dynamic parallelism, better C++ support, etc.
- Meanwhile, Intel, AMD, and others have been investing in multi-core computing, producing the Intel Xeon Phi. OpenCL consortium also started.

# Simplistic hardware structure - typical CPU




- Large number of transistors associated with flow control, cache, etc.
- Handles branch prediction, speculative execution, etc.

# Simplistic hardware structure - typical GPU




- Larger proportion of transistors associated with floating point operations.
- No branch prediction or speculative execution on GPU cores.
- Results in higher theoretical FLOPS as long as we can get the data there fast enough.

# GPU programming

GPU Programming model:

- GPU programming is typically Single Instruction Multiple Data (SIMD) or SIMT(hread)
- Programming for GPUs requires rethinking algorithms and memory layout.
- Best algorithm for CPU not necessarily best for GPU.
- Knowledge of GPU hardware required for best performance

GPU programming works best if we:

- Perform same operation simultaneously on multiple pieces of data
- Organise operations to be as independent as possible
- Arrange data in GPU memory to maximise rate of data access

# CUDA language

- The CUDA language taught in this course is applicable to NVIDIA GPUs only.
- Other GPU manufacturers exist, e.g. AMD.
- CUDA (Compute Unified Device Architecture) is enhanced C++
- Contains extensions corresponding to hardware features of the GPU.
- Easy to use (if you know C++)...
- but contains much more potential for hard-to-trace errors than C++ due to the hardware and threaded access.

# Alternatives to CUDA

Caveat: I have not studied any of these...

- Microsoft's DirectCompute (following DirectX)
- AMD Stream Computing SDK (AMD-hardware specific)
- OpenCL (`opencl.org`) aims to be applicable to all GPUs.
- SYCL (`https://www.khronos.org/sycl/`) aims to be applicable to all multicore processors.
- PyCUDA (Python wrapper for CUDA)
- CLyther (Python wrapper for OpenCL)
- Jacket (MatLab or C++ wrapper - commercial)
- MatLab's Parallel Computing Toolbox
- Various NVIDIA libraries: cuBLAS, cuSPARSE, cuRAND, cuFFT, nvGRAPH, and more

# Installing CUDA Drivers and Toolkit

- Instructions for latest driver can be found at `https://developer.nvidia.com/cuda-downloads`
- For Ubuntu, proprietary drivers are in standard repositories but may not be the absolute latest version
- The Toolkit from NVIDIA provides a compiler, libraries, documentation, and a wide range of examples.
- CSC desktops and laptops have NVIDIA drivers and Toolkit pre-installed.

# Compiling a CUDA program

- NVIDIA have their own compiler, `nvcc` (based on Clang).
- Uses either `g++` or `cl` (Microsoft) for compiling CPU code.
- `nvcc` needed to compile any CUDA code
- CUDA API functions can be compiled with normal compiler (`gcc` / `cl`)

**Compilation**

```
nvcc helloWorld.cu -o helloWorld -O3
```

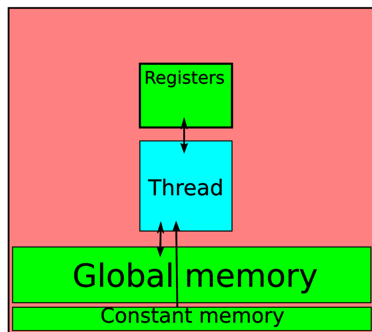
# Outline


1. CUDA History
2. CUDA Hardware Model
3. Terminology

# Programming model

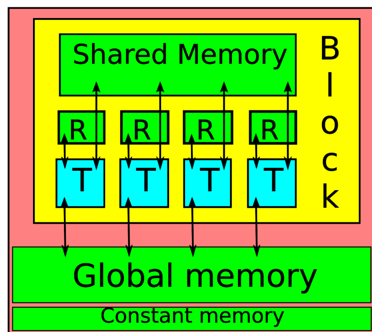
- CUDA works in terms of threads (Single Instruction Multiple Thread)
- Threads are separate processing elements within a program:
  - Each executes exactly the same set of instructions,
  - typically differing only by their thread number,
  - which can lead to differing data dependencies / execution paths.
- Threads execute independently of each other unless explicitly synchronized (or part of same warp)
- Typically threads have access to same memory block as well as some data local to each thread (and perhaps to a subset of threads)
- i.e. shared-memory paradigm with all the potential issues that implies.

# Conceptual diagram



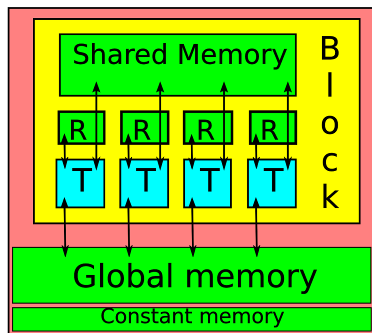

- Each thread has internal registers (local variables) and access to global and constant memory

# Conceptual diagram



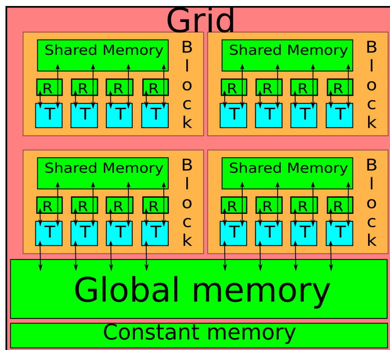

- Each thread has internal registers (local variables) and access to global and constant memory
- Threads arranged in blocks have access to a common block of shared memory. Threads can communicate within blocks

# Conceptual diagram



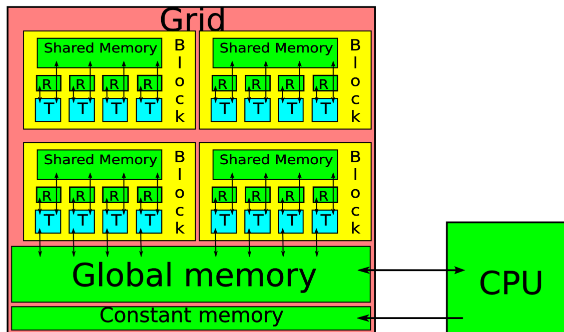

- Each thread has internal registers (local variables) and access to global and constant memory
- Threads arranged in blocks have access to a common block of shared memory. Threads can communicate within blocks
- A block of threads is run entirely on a single streaming-multiprocessor on the GPU.

# Conceptual diagram



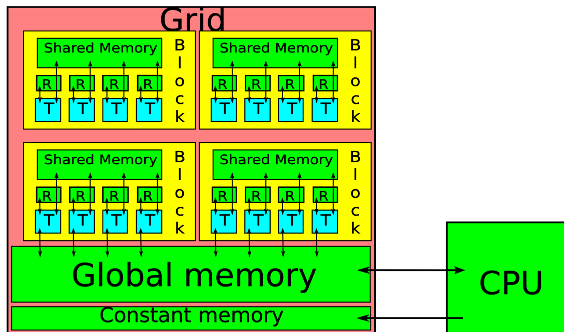

- Blocks are arranged into a grid. Separate blocks cannot (easily) communicate as they may be run on separate streaming multiprocessors.

# Conceptual diagram




- Blocks are arranged into a grid. Separate blocks cannot (easily) communicate as they may be run on separate streaming multiprocessors.
- Communication with CPU is only via global and constant memory.

# Conceptual diagram




- Blocks are arranged into a grid. Separate blocks cannot (easily) communicate as they may be run on separate streaming multiprocessors.
- Communication with CPU is only via global and constant memory.
- The GPU cannot request data from the CPU; the CPU must send data to it.

# Conceptual elaboration

- Work for the GPU is written in separate functions called *kernels*, called from the main program.
- When running a program, you launch one or more blocks of threads, specifying the number of threads in each block and the total number of blocks.
- Each block of threads is allocated to a Streaming Multiprocessor (SM) by the hardware.
- Each thread has a unique index made up of a thread index and a block index which are available inside your code.
- These are used to distinguish threads in a kernel.
- Communication between threads in the same block is possible (via shared memory or special functions).
- Communication between blocks (in the same grid) is not.

# Simple kernel example

```
__global__ void add(float* a, float* b, float* c, int N)
{
  int i = threadIdx.x;
  if( i < N )
  {
    c[i] = a[i] + b[i];
  }
}

int main(){
  // Call kernel from CPU
  add<<<1, N>>>(a,b,c,N);
}
```

# Hardware specifics

- A GPU is made up of several streaming multiprocessors (4 for Quadro P620, 128 for Tesla A100) each of which consists of 32, 48, 128, or 192 cores.
- No. of cores per SM is lower in older cards.
- Each SM runs a warp of 32 threads at once (width 32 vectorized instructions).
- Global memory access relatively slow (200-400 clock cycles)
- Shared memory access quicker (20-40 cycles - higher bandwidth)
- A SM can have many thread blocks allocated at once, but only runs one at any one time.
- The SM can switch rapidly to running other threads to hide memory latency
- Thread registers (local variables) and shared memory for a thread-block remain on multiprocessor until thread-block finished.

# Streaming Multiprocessor

- When launching a kernel each block of threads is assigned to a Streaming Multiprocessor
- All threads run exactly the same kernel code.
- All threads in the block run potentially concurrently; do not assume any particular ordering.
- Within a block threads are split into warps of 32 threads which all execute the same instruction at the same time
- Different warps within a block are not necessarily run simultaneously, but can be synchronized at any point.
- All threads in a block have access to some common shared memory on the SM.
- Multiple thread blocks can be allocated on a SM at once, assuming sufficient memory.
- Execution can switch to other blocks if necessary, while holding other running blocks in memory.

# Hardware limitations

Tesla K20 - Compute capability 3.5 (Release Nov 2012)

- 13 Multiprocessors, each with 192 cores = 2496 cores.
- Per streaming multiprocessor (SM):
  - 65,536 32-bit registers
  - 16-48kB shared memory per SM
  - 16 active blocks
- Global:
  - 64kB constant memory
  - 5GB global memory
- Maximum threads per block: 1024

# Hardware limitations

Pascal P100 - Compute capability 6.0 (Release April 2016)

- 56 Multiprocessors, each with 64 cores = 3584 cores
- Per streaming multiprocessor (SM):
  - 65,536 32-bit registers
  - 96kB shared memory per SM
  - 32 active blocks
- Global:
  - 64kB constant memory
  - 16GB global memory
- Maximum threads per block: 1024

# Hardware limitations

Ampere A100 - Compute capability 8.0 (Release May 2020)

- 108 Multiprocessors, each with 64 cores = 6912 cores
- Per streaming multiprocessor (SM):
  - 65,536 32-bit registers
  - 164kB shared memory per SM
  - 32 active blocks
- Global:
  - 64kB constant memory
  - 80GB global memory
- Maximum threads per block: 1024

# Hardware limitations

- If you have a recent NVIDIA GPU in your own PC, it will have CUDA capability.
- If bought recently, this will probably be 6.0 or better.
- The commercial-grade GPUs differ only in that they have more RAM, more SMs, and Error Correcting Code (ECC) memory.
- The same applications demonstrated here will run on your own hardware, so you can often use a local machine for testing
- See `https://www-internal.lsc.phy.cam.ac.uk/systems.shtml` for CUDA capability details for our machines. (Sadly none $\geqslant 6.0$ as yet.)

# Outline

1 CUDA History

2 CUDA Hardware Model

3 Terminology

# Terminology

## Host

The PC's main CPU and/or RAM

## Device

A GPU being used for CUDA operations (usually the GPU global memory)

## Kernel

Block of code to be run on a GPU

## Thread

Single running instance of a kernel

## Block

Indexed collection of threads

# Terminology continued

## Global memory

RAM on a GPU that can be read/written to by any thread

## Constant memory

RAM on a GPU that can be read by any thread

## Shared memory

RAM on a GPU that can be read/written to by any thread in a block.

## Registers

Memory local to each thread.

## Latency

Time between issuing an instruction and instruction being completed

# Terminology ctd.

## Compute capability

NVIDIA label for what hardware features a graphics card has.

| CC | Hardware name | Release year |
|---|---|---|
| 3.x | Kepler | 2012 |
| 5.x | Maxwell | 2014 |
| 6.x | Pascal | 2016 |
| 7.x | Volta | 2017 |
| 7.5 | Turing | 2019 |
| 8.0 | Ampere | 2020 |

## Warps

A warp is a consecutive set of 32 threads within a block that are executed simultaneously.

If branching occurs within a warp, then code branches execute serially.

# Parallel Correctness

- Remember that threads are essentially independent
- Cannot make any assumptions about execution order
- Blocks may be run in any order within a grid
- Algorithm must be designed to work without block-ordering assumptions
- Any attempt to use global memory to pass information between blocks will probably fail (or at least give unpredictable results)...
- The best approach for parallelism is that each thread reads data from a different array element (and writes to a separate array).

# Simple race-condition

- The simple instruction `i++` usually results in the operations:
  - Read value of `i` into processor register
  - Increment register
  - Write `i` back to memory
- If the same instruction occurs on separate threads, and `i` refers to the same place in global memory, then the following could occur:

Start with $i = 0$

| Thread 1 | Thread 2 |
| --- | --- |
| Read `i` | |
| Increment register | |
| Write `i` | |
| | Read `i` |
| | Increment register |
| | Write `i` |

End with $i = 2$

# Simple race-condition

- The simple instruction `i++` usually results in the operations:
  - Read value of `i` into processor register
  - Increment register
  - Write `i` back to memory
- If the same instruction occurs on separate threads, and `i` refers to the same place in global memory, then the following could occur:

Start with $i = 0$

| Thread 1 | Thread 2 |
|---|---|
| Read `i` | Read `i` |
| Increment register | |
| Write `i` | |
| | Increment register |
| | Write `i` |

End with $i = 1$

# Global memory read/write

## Atomic Instruction

Using an atomic instruction will ensure that the reads and writes occur serially. This will ensure correctness at the expense of performance.

## Programming Guide

If a non-atomic instruction executed by a warp writes to the same location in global or shared memory for more than one of the threads of the warp, the number of serialized writes that occur to that location varies depending on the compute capability of the device and which thread performs the final write is undefined.

In other words: Here be dragons!
Each thread should write to its own global memory location
Ideally, no global memory needing to be read should be written to by the same kernel.

# Summary

- Introduced GPUs - history and their advantages
- Described hardware model
- and how it relates to the programming model
- Defined some terminology for later use