CUDA programming on NVIDIA GPUs

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford-Man Institute for Quantitative Finance

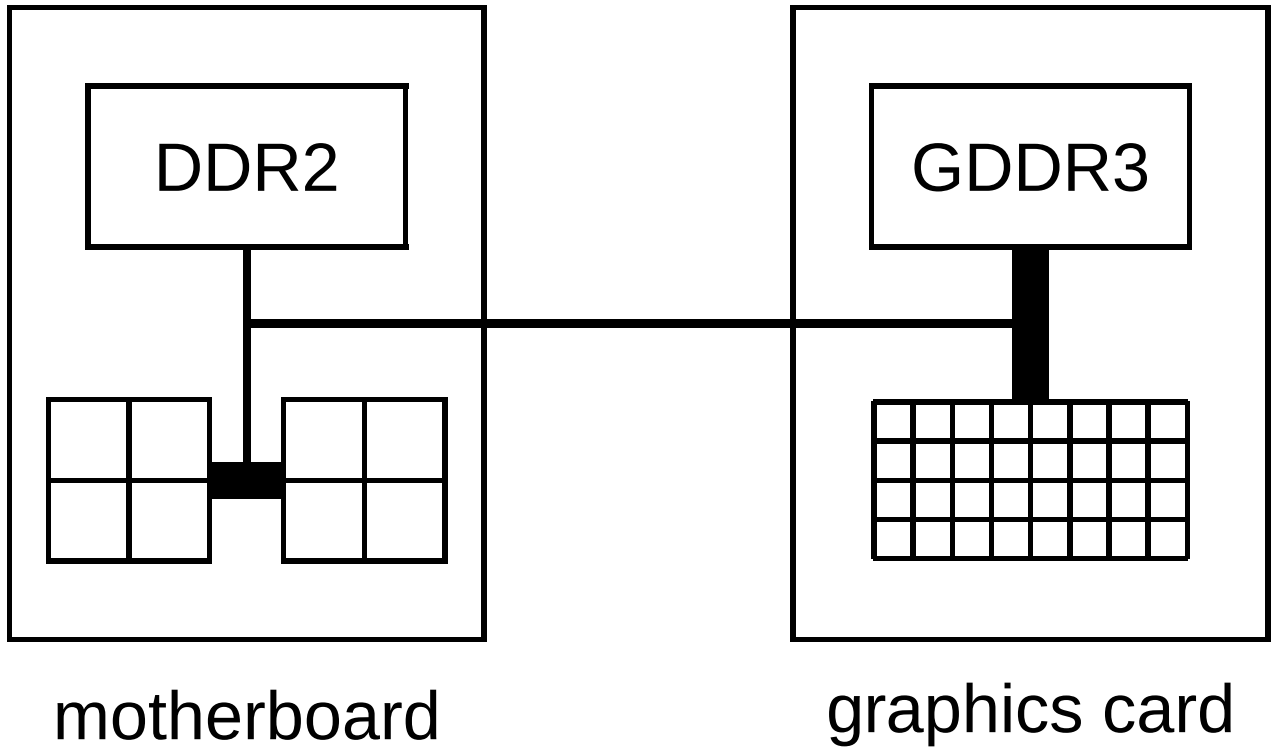
Oxford eResearch Centre

Overview

- hardware view
- software view
- CUDA programming

Hardware view

At the top-level, there is a PCIe graphics card with a many-core GPU sitting inside a standard PC/server with one or two multicore CPUs:



Hardware view

At the GPU level:

- basic building block is a “multiprocessor” with
  - 8 cores
  - 8192 registers (16384 on newest chips)
  - 16KB of shared memory
  - 8KB cache for constants held in graphics memory
  - 8KB cache for textures held in graphics memory
- different chips have different numbers of these:

product	multiprocessors	bandwidth	cost
8800 GT	14	58GB/s	£100
9800 GT	14	58GB/s	£100
9800 GX2	32	128GB/s	£250

Hardware view

Key hardware feature is that the 8 cores in a multiprocessor are SIMD (Single Instruction Multiple Data) cores:

- all cores execute the same instructions simultaneously, but with different data
- similar to vector computing on CRAY supercomputers
- minimum of 4 threads per core, so end up with a minimum of 32 threads all doing the same thing at (almost) the same time
- natural for graphics processing and much scientific computing
- SIMD is also a natural choice for massively multicore to simplify each core

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 copies of execution kernel on device
5. copies data from device memory to host
6. repeats 2-4 as needed
7. de-allocates all memory and terminates

Software view

At a lower level, within the GPU:

- each copy of the execution kernel executes on one of the “multiprocessors”
- if the number of copies exceeds the number of multiprocessors, then more than one will run at a time on each multiprocessor if there are enough registers and shared memory, and the others will wait in a queue and execute later
- all threads within one copy can access local shared memory but can’t see what the other copies are doing (even if they are on the same multiprocessor)
- there are no guarantees on the order in which the copies will execute

CUDA programming

CUDA is NVIDIA's program development environment:

- based on C with some extensions
- FORTRAN support coming in near future
- multicore x86 back-end also coming soon to make CUDA code portable
- may evolve into proposed OpenCL standard which may be supported also by AMD/ATI
- lots of example code and good documentation
  – 2-4 week learning curve for those with experience of OpenMP and MPI programming
- growing user community active on NVIDIA forums

CUDA programming

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

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

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

CUDA programming

In its simplest form it looks like:

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

where

- `gridDim` is the number of copies of the kernel (the “grid” size”)
- `blockDim` is the number of threads within each copy (the “block” size)
- `args` is a limited number of arguments, usually mainly pointers to arrays in graphics memory

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 copy of the kernel is started on a multiprocessor 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)
- constants held in device memory
- uninitialised shared memory and private registers/local variables
- some special variables:
  - `gridDim` size (or dimensions) of grid of blocks
  - `blockIdx` index (or 2D/3D indices)of block
  - `blockDim` size (or dimensions) of each block
  - `threadIdx` index (or 2D/3D indices) of thread

CUDA programming

The kernel code involves operations such as:

- read from/write to arrays in device memory
- write to/read from arrays in shared memory
- read constants
- use a texture array for a lookup table
- perform integer and floating point operations on data held in registers

The reading and writing is done implicitly – data is automatically read into a register when needed for an operation, and is automatically written when there is an assignment.

CUDA programming

Simple Monte Carlo example:

- simplest possible example
- involves the calculation of a large number of “paths”
- each path calculation is completely independent of the others
- each should have its own set of random numbers – still waiting for NVIDIA to develop a random number generation library so just set all of them to 0.3
- `path_calc` calculates the “paths” and `portfolio` calculates the “portfolio” value, which is then written into a device array
- The host code copies the results back and averages them

CUDA programming

Monte Carlo LIBOR application:

- ideal because it is trivially parallel – each path calculation is independent of the others
- timings in seconds for 96,000 paths, with 1500 blocks each with 64 threads, so one thread per path
- executed 5 blocks at a time on each multiprocessor, so 40 active threads per core
- remember: CUDA results are for single precision

	time
original code (VS C++)	26.9
CUDA code (8800GTX)	0.2

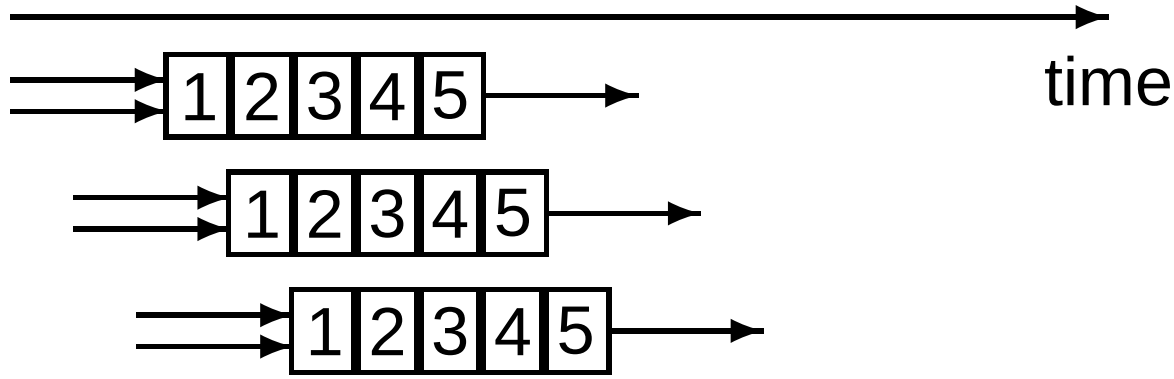
CUDA multithreading

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 execute in “warps” of 32 threads per multiprocessor (4 per core) – execution alternates between “active” warps, with warps becoming temporarily “inactive” when waiting for data

CUDA multithreading

- for each thread, one operation completes long 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 400-600 cycles; with 40 threads this is equivalent to 10-15 operations and can be managed by the compiler

CUDA programming

What have I not discussed:

- “memory transfer coalescence”: bandwidth to device memory is maximised by a half-warp of 16 threads loading 16 contiguous `floats` or `ints` with correct alignment – this is often the most important/tedious aspect of getting good performance
- use of shared-memory – essential for PDE applications to minimise the amount of data loaded from device memory
- use of textures – rather specialised, I’ve used them only for a random-access lookup table to avoid penalty of non-coalesced transfers

CUDA programming

Finite difference application:

- recently started work on simple 3D finite difference applications
  - Jacobi iteration for discrete Laplace equation
  - CG iteration for discrete Laplace equation
  - ADI time-marching
- conceptually straightforward for someone who is used to partitioning grids for MPI implementations
  - each multiprocessor works on a block of the grid
  - threads within each block read data into local shared memory, do the calculations in parallel and write new data back to main device memory

CUDA programming

3D finite difference implementation:

- insufficient shared memory to hold whole 3D block, so hold 3 working planes at a time (halo depth of 1, just one Jacobi iteration at a time)
- key steps in kernel code:
  - load in $k=0$ z-plane (inc x and y-halos)
  - loop over all z-planes
    - load $k+1$ z-plane (over-writing $k-2$ plane)
    - process k z-plane
    - store new k z-plane
- $50\times$ speedup relative to Xeon single core, compared to $5\times$ speedup using OpenMP with 8 cores.

Final words

Will GPUs have real impact?

- I think they're the most exciting development since initial development of PVM and Beowulf clusters
- Have generated a lot of interest/excitement in academia, being used by application scientists, not just computer scientists
- Potential for $10-100\times$ speedup and improvement in GFLOPS/£ and GFLOPS/watt
- Effectively a personal cluster in a PC under your desk
- Needs work on tools and libraries to simplify development effort

Webpages

Wikipedia overviews of GeForce cards:
`en.wikipedia.org/wiki/GeForce_8_Series`
`en.wikipedia.org/wiki/GeForce_9_Series`

NVIDIA's CUDA homepage:
`www.nvidia.com/object/cuda_home.html`

Microprocessor Report article:
`www.nvidia.com/docs/IO/47906/220401_Reprint.pdf`

LIBOR test code:
`www.maths.ox.ac.uk/~gilesm/hpc/`