An introduction to GPU programming

Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford-Man Institute of Quantitative Finance

Oxford eResearch Centre

Overview

The first lecture covered the basics; now we look at the extra complexities

- warps and conditional warp divergence
- memory transfer coalescence
- local, constant, shared and texture memory
- thread synchronisation
- finite difference application
- further reading

Warps

Already said that within a block of threads, the threads are executes in groups of 32 called a warp

- if the block size is not divisible by 32, some of the threads in the last warp don't do anything
- if the block is 2D or 3D, the threads are ordered by first dimension, then second, then third – then split into warps of 32
- all threads in a warp do the same thing at the same time
- different warps are executed independently (except for explicit synchronisation) by run-time scheduler
- it's possible each warp executes until it needs to wait for data (from device memory or a previous operation) then another warp has a turn

Warp divergence

What happens if different threads in a warp need to do different things?

```
if (x<0.0)
  z = x-2.0;
else
  z = sqrt(x);
```

In a simple case like this, all threads will compute a logical *predicate* and two predicated instructions

```
p = (x<0.0);
if (p)  z = x-2.0;     // single instruction
if (!p) z = sqrt(x);
```

Warp divergence

This is called *warp divergence*

The treatment is almost identical to the logical merge operation on CRAY vector systems:

```
z =  p ? x-2.0: sqrt(x);
```

Note also that:

- `sqrt(x)` will produce a `NaN` when `x<0`, but this doesn't matter as the `NaN` will not be stored
- all threads execute both conditional branches, so execution cost is sum of both branches
- in the worst case, if one thread per warp takes an expensive branch, then lose a factor 32 in performance

Warp divergence

If the branches are big, `nvcc` compiler inserts code to check if all threads in the warp take the same branch (*warp voting*) and if so branches accordingly.

In some cases, can determine at compile time this must happen. e.g. if `case` is a run-time argument

```
if (case==1)
  z = x*x;
else
  z = x+2.3;
```

Note: doesn't matter what is happening with other warps – each warp is treated separately.

Warp divergence

Warp divergence can lead to a big loss of parallel efficiency – one of the first things I look out for in a new application.

Typical example: PDE application with boundary conditions

- if boundary conditions are cheap, loop over all nodes and branch as needed for boundary conditions
- if boundary conditions are expensive, first kernel does interior points, second kernel does boundary points

Warp divergence

Another example: processing a long list of elements – depending on run-time values a few require very expensive processing

GPU implementation:

- first process list to build two sub-lists of "simple" and "expensive" elements
- then process two sub-lists separately

Note: none of this is rocket science, or new – this is what we did 20 years ago on CRAY and Thinking Machines systems.

What's important is to understand hardware behavior and design your algorithms / implementation accordingly

Bandwidth Issues

Back-of-the-envelope calculation for single precision:

- GPU: 1 TFlops
- GPU bandwidth: 100 GB/s = 25 `Gfloat`/s
- PCIe bandwidth: 4 GB/s = 1 `Gfloat`/s
- (CPU bandwidth: 25 GB/s = 5 `Gfloat`/s)

First conclusion: need at least 1000 flops per data access to get good performance by offloading small pieces of an application – only good example is dense linear algebra

Usually better to transfer whole application to GPU, and work on it there to completion

Bandwidth Issues

Second conclusion: even with application data sitting in GPU device memory, need at least 20 flops per data access to get good performance, and that assumes max bandwidth is achieved

Bandwidth on current Tesla GPU:

- coalesced: 100 GB/s = 25 `Gfloat`/s
- contiguous, but misaligned: 50 GB/s = 12.5 `Gfloat`/s
- random scatter/gather: 10 GB/s = 2.5 `Gfloat`/s

Current CUDA codes work hard to achieve coalesced transfers – probably much less important with the new Fermi GPU because of L1/L2 caches

Bandwidth Issues

Explanation of terms:

- Contiguous access: each element of the warp accesses a neighbouring array element. e.g. `x[tid]` where `tid = threadIdx.x + blockIdx.x*blockDim.x`
- Coalesced access: contiguous access in which first element is correctly aligned (usually means a multiple of 16 or 32)
- Random access: each thread in the warp accesses entirely different parts of memory

Bandwidth Issues

What's going on?

- memory transfers are performed in blocks, just like cache lines in a CPU – this is how high bandwidth is achieved at hardware level
- widely scattered data means lots of blocks have to be transferred
- current hardware discards all data after completing the operation which needs it – if another operation needs the same data, it has to be reloaded
- in Fermi, it's kept in the cache, so can re-use same data *and* its neighbours without additional transfer

Different variables

Global variables:

- held in device (graphics) memory allocated by host
- pointer passed into kernel routine
- read/write by either host or kernel
- exist until de-allocated

Local variables:

- private variables for each thread in kernel
- scalars usually assigned to registers by compiler
- vectors usually stored in device memory (because registers not addressable)
- exist only for lifetime of kernel

Different variables

Constant variables:

- global scope within a `*.cu` CUDA file
- defined by a `__constant__` prefix
- value set by host, read-only by kernels
- exist for lifetime of entire application
- current GPUs have a 16KB constant cache
- very useful to avoid wasting precious registers or shared memory on essential constants

(Note: literal constants are kept in the code)

Different variables

Shared variables:

- defined by a `__shared__` prefix in kernel code
- (can be sized dynamically but it's trickier)
- limited by 16KB size of shared memory in SM (going up to 48KB in Fermi)
- useful for
  - re-use of data used by more than one thread (e.g. finite difference code)
  - communication/cooperation between threads (e.g. computing a sum)
  - addressable private arrays (so not stored in device memory)

Different variables

Texture variables:

- read-only variables stored in device memory, cached onto GPU
- intended for texture mapping in computer graphics, but also useful for applications like random access lookup tables
- syntax is a bit unusual, confusing
- maybe no longer important when Fermi comes out?
- for simple example, see Practical 4 in my CUDA course: `http://people.maths.ox.ac.uk/~gilesm/cuda/`

Synchronization

Synchronization can be forced at two levels, within a kernel or within the main host code.

Within a kernel the instruction

`__syncthreads();`

forces all warps to wait until rest have reached that point

Essential when need to make sure one task
(e.g. reading data into shared memory)
is completed before starting the next
(e.g. using the shared memory values in a computation)

Synchronization

At the host level, important to understand that the kernel launch is *asynchronous* – it starts the GPU kernel but the CPU code is then free to continue doing other things

Need to look at the documentation carefully to see which CUDA operations are synchronous and asynchronous, and also which force the completion of earlier CUDA operations

The instruction

`cudaThreadSynchronize();`

forces completion of all earlier operations – often used when doing timing

Monte Carlo example

LIBOR market model:

- code available on my webpage `people.maths.ox.ac.uk/~gilesm/libor/`
- simplest possible "real" example from computational finance – trivially parallel
- involves the calculation of a large number of "paths"; each is completely independent of the others but uses its own set of random numbers
- host code copies the results back and averages them

Monte Carlo example

- single precision results
- `gcc` times are spectacularly slow due to `expf` function
- timings below come from someone who is expert with both CUDA and Intel's `icc` compiler

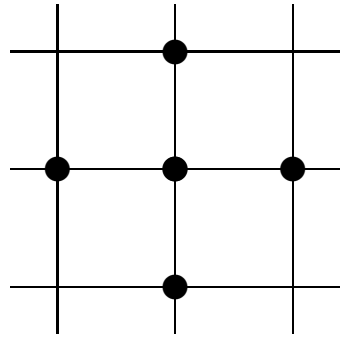
	paths per second
OpenMP on quad-core 3.2GHz Nehalem	0.15 M
OpenMP + SSE vectorisation on same	0.52 M
CUDA on GTX280	4.8 M

Practical 2

Really simple Monte Carlo example:

- random number generation using NAG library (produces numbers on the GPU and stores them in the graphics memory)
- each “path” involves the approximate solution of 2 stochastic differential equations
- demonstrates
  - initialisation and use of `__constant__` variables
  - timing functions
  - importance of memory coalescence

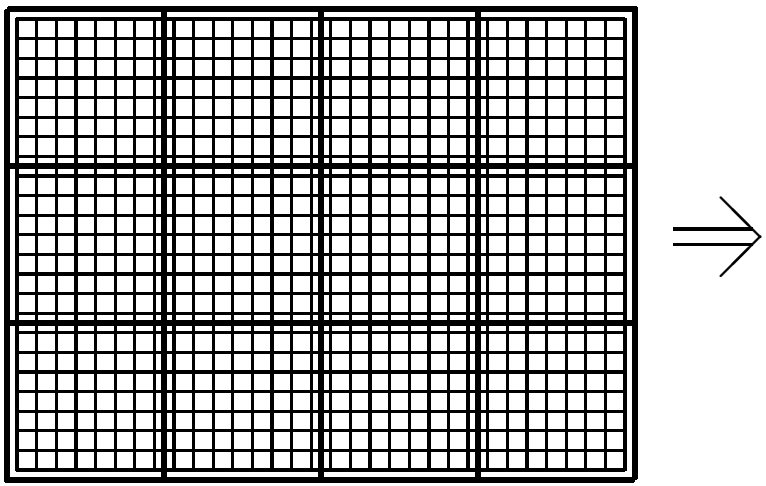
Finite Difference Application



Simple model problem: Jacobi iteration to solve discretisation of Laplace equation

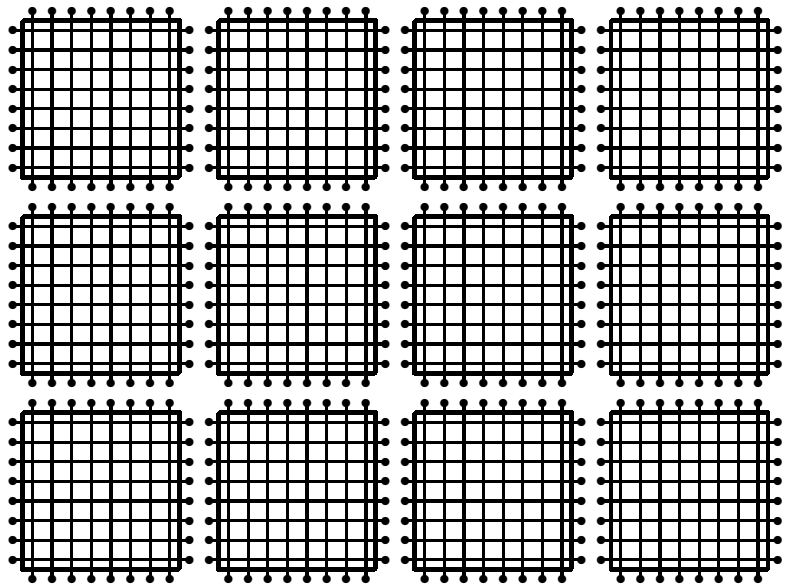
$$V_{i,j}^{n+1} = \frac{1}{4}\left(V_{i+1,j}^{n} + V_{i-1,j}^{n} + V_{i,j+1}^{n} + V_{i,j-1}^{n}\right)$$

Finite Difference Application

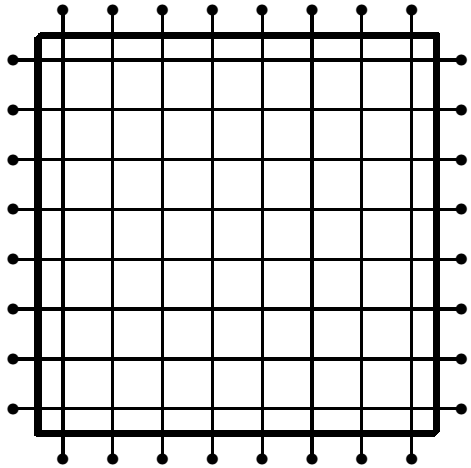


Second idea: take ideas from distributed-memory parallel computing and partition grid into pieces

Finite Difference Application



Finite Difference Application



Each block of threads processes one of these grid blocks, reading in old values and computing new values

Finite Difference Application

How should this be programmed?

First idea: each thread does one grid point, reading in directly from graphics memory the old values at the 4 neighbours (6 in 3D)

Performance is poor on current hardware:

- each old value read in 4 times (6 in 3D) and 2 of these suffer a factor $2\times$ penalty due to misalignment
- including coalesced writing of new value, leads to factor 3.5 (4.5 in 3D) increase in bandwidth compared to ideal
- *however* this version will probably be very efficient on new Fermi GPU because of its L1/L2 caches

Finite Difference Application

Second idea: load old data into shared memory:

- each thread loads in the data for its grid point (coalesced) and maybe one halo point (only partially coalesced)
- need a `__syncthreads();` instruction to ensure all threads have completed this before any of them access the data
- each thread computed its new value and writes it to graphics memory
- not possible to get full coalescence due to *halo* boundary data – probably increases bandwidth by factor 1.5 compared to ideal

Finite Difference Application

2D finite difference implementation:

- good news: $30\times$ speedup relative to Xeon single core, and $7\times$ speedup relative to 2 quad-core Xeons using OpenMP
- bad news: grid size has to be 1024^2 to have enough parallel work to do to get this performance
- unlikely to have real applications of this size; an alternative is to perform multiple 2D calculations at the same time, perhaps with different parameter values

Finite Difference Application

3D finite difference implementation:

- insufficient shared memory for whole 3D block, so hold 3 working planes 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
    - process k z-plane
    - store new k z-plane
- $50\times$ speedup relative to Xeon single core, and $10\times$ speedup relative to 2 quad-core Xeons

Practical 3

- has both "naive" and efficient implementations for finite difference application described above
- has "gold" CPU version to check results are correct – this follow the approach used by the CUDA SDK examples and is good programming practice
- also demonstrates use of timing functions

Checklist

What I think about for a new application:

- lots of inherent parallelism?
- viable to offload small compute-intensive bits?
- any potential problems with warp divergence?
- very compute-intensive, or need to be careful to minimise bandwidth requirements?
- tricky bits (e.g. reduction, scan)?
- use shared memory to improve data re-use?
- (on Fermi, use shared memory or rely on cache?)
- how to minimise registers per thread?
- is there a relevant example in CUDA SDK? or forums? or `gpucomputing.net`?

Further reading

- new book by David Kirk and Wen-Mei Hwu: `www.elsevierdirect.com/morgan_kaufmann/kirk/`
- UIUC course by Wen-Mei Hwu and others: `courses.ece.illinois.edu/ece498/al/Syllabus.ht`
- my course: `people.maths.ox.ac.uk/~gilesm/cuda/`
- CUDA SDK examples: `www.nvidia.com/object/cuda_sdks.html`
- CUDA Programming Guide and Best Practice Guide: `www.nvidia.com/object/cuda_develop.html`

Further reading

Most important things I've *not* mentioned:

- CUBLAS, CUFFT and CUDPP libraries
  – see documentation in Toolkit
- how to implement parallel reduction and scan
  – see SDK examples `reduction` and `scan`
- coping with a maximum of 64 registers per thread
- code optimization tricks – see SDK examples and CUDA C Programming Best Practices Guide

Final Words

- I think GPU computing will be big for next 5–10 years
- right now NVIDIA is well in the lead, but AMD might get back in the game with OpenCL
- I think Intel will respond through increasing AVX vector unit size alongside "standard" CPUs
- within 10 years, we will have GPUs with 20k cores and 1M threads!
- will the interconnect (e.g. Infiniband) become the bottleneck in big clusters?

- Good luck – have fun!