Tackling a new CUDA application

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Oxford e-Research Centre

Initial planning

1) Has it been done before?

- check CUDA SDK examples
- check CUDA user forums
- check `gpucomputing.net`
- check with Google

Initial planning

2) Where is the parallelism?

- efficient CUDA execution needs thousands of threads
- usually obvious, but if not
  - go back to 1)
  - talk to an expert – they love a challenge
  - go for a long walk
- may need to re-consider the mathematical algorithm being used, and instead use one which is more naturally parallel – but this should be a last resort

Initial planning

Sometimes you need to think about “the bigger picture”

Consider a 3D finite difference example:

- lots of grid nodes so lots of inherent parallelism
- even for ADI method (which needs a set of tri-diagonal equations to be solved in each direction), a grid of 128^3 has 128^2 tri-diagonal solutions to be performed in parallel so OK to assign each one to a single thread
- but what if we have a 2D or even 1D problem to solve?

Initial planning

If we only have one such problem to solve, why use a GPU?

But in practice, often have many such problems to solve:

- different initial data
- different model constants

This adds to the available parallelism

Initial planning

2D:

- 48KB of shared memory == 12K `float` so grid of 64^2 could be held within shared memory
  - one kernel for entire calculation
  - each block handles a separate 2D problem; almost certainly just one block per SM
- for bigger 2D problems, would need to split each one across more than one block
  - separate kernel for each timestep / iteration

Initial planning

1D:

- can certainly hold entire 1D problem within shared memory of one SM
- maybe best to use a separate block for each 1D problem, and have multiple blocks executing concurrently on each SM
- but for implicit time-marching need to solve single tri-diagonal system in parallel – how?

Initial planning

Cyclic reduction: starting from

$$-a_n\, x_{n-1} + x_n - c_n\, x_{n+1} = d_n, \qquad n = 1, \ldots N$$

for even n, add a_n times row $n-1$, and c_n times row $n+1$ to get

$$\begin{aligned} & -a_n\, a_{n-1}\, x_{n-2} + (1 - a_n c_{n-1} - c_n a_{n+1})\, x_n - c_n\, c_{n+1}\, x_{n+2} \\ = \quad & d_n + a_n\, d_{n-1} + c_n\, d_{n+1} \end{aligned}$$

then normalise to get

$$-a_n^*\, x_{n-2} + x_n - c_n^*\, x_{n+2} = d_n^*, \qquad n = 2, 4, \ldots$$

Note this can be done in parallel

Initial planning

Repeat, halving the number of unknowns at each stage, until 1 left, then do back substitution to get other values.

eqns: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11

eqns: 2, 4, 6, 8, 10

eqns: 4, 8

eqns: 8

soln: 8

soln: 4

soln: 2, 6, 10

soln: 1, 3, 5, 7, 9, 11

Parallel implementation similar to scan

Initial planning

3) Break the algorithm down into its constituent pieces

- each will probably lead to its own kernels
- do your pieces relate to the 7 dwarfs?
  - dense / sparse linear algebra
  - spectral methods
  - N-body methods
  - structured / unstructured grids
  - Monte Carlo
- re-check literature for each piece – sometimes the same algorithm component may appear in widely different applications
- check whether there are existing libraries which may be helpful

Initial planning

4) Is there a problem with warp divergence?

- GPU efficiency can be completely undermined if there are lots of divergent branches
- may need to implement carefully – lecture 3 example:

  processing a long list of elements where, depending on run-time values, a few involve expensive computation:
  - first process list to build two sub-lists of “simple” and “expensive” elements
  - then process two sub-lists separately

- . . . or again seek expert help

Initial planning

5) Is there a problem with host <–> device bandwidth?

- usually best to move whole application onto GPU, so not limited by PCIe bandwidth (5GB/s)
- occasionally, OK to keep main application on the host and just off-load compute-intensive bits
- dense linear algebra is a good off-load example; data is $O(N^2)$ but compute is $O(N^3)$ so fine if N is large enough
- rule of thumb: need at least 100 operations on GPU per byte of data transferred

Heart modelling

Heart modelling is another interesting example:

- keep PDE modelling (physiology, electrical field) on the CPU
- do computationally-intensive cellular chemistry on GPU (naturally parallel)
- minimal data interchange each timestep

Initial planning

6) is the application compute-intensive or data-intensive?

- break-even point is roughly 20 operations (FP and integer) for each 32-bit device memory access (assuming full cache line utilisation)
- good to do a back-of-the-envelope estimate early on before coding $\Longrightarrow$ changes approach to implementation

Initial planning

If compute-intensive:

- don’t worry (too much) about cache efficiency
- minimise integer index operations – surprisingly costly
- if using double precision, think whether it’s needed

If data-intensive:

- ensure efficient cache use – may require extra coding
- may be better to re-compute some quantities rather than fetching them from device memory

Initial planning

Need to think about how data will be used by threads, and therefore where it should be held:

- registers (private data)
- shared memory (for shared access)
- device memory (for big arrays)
- constant arrays (for global constants)
- “local” arrays (efficiently cached on Fermi)

Initial planning

With complex applications, I increasingly worry about "register pressure", i.e. coping with a maximum of 63 registers per thread:

- split big kernels into two – may increase bandwidth requirements but probably reduces register count

  (Example: in Monte Carlo simulations, pre-compute random numbers or do them on-the-fly?)
- if any variables have same value for all threads, put them into shared memory, set by thread 0
- sometimes hard to predict what will work best, may need to experiment later

Initial planning

When working with shared memory, be careful to think about thread synchronisation.

Very important!

Forgetting a

```
__syncthreads();
```

may produce errors which are unpredictable / rare — the worst kind.

Also, make sure all threads reach the synchronisation point — otherwise could get deadlock.

Initial planning

If you think you may need to use “exotic” features like atomic locks:

- look for SDK examples
- write some trivial little test problems of your own
- check you really understand how they work

Never use a new feature for the first time on a real problem!

Initial planning

Read NVIDIA documentation on performance optimisation:

- section 5 of CUDA Programming Guide
- CUDA C Best Practices Guide
- Fermi Tuning Guide

Programming and debugging

Many of my comments here apply to all scientific computing

Though not specific to GPU computing, they are perhaps particularly important for GPU / parallel computing because

debugging can be hard!

Above all, you don't want to be sitting in front of a 50,000 line code, producing lots of wrong results (very quickly!) with no clue where to look for the problem

Programming and debugging

- plan carefully, and discuss with an expert if possible
- code slowly, ideally with a colleague, to avoid mistakes but still expect to make mistakes!
- code in a modular way as far as possible, thinking how to validate each module individually
- build-in self-testing, to check that things which ought to be true, really are true

  (In my current project I have a flag `OP_DIAGS`;
  the larger the value the more self-testing the code does)
- overall, should have a clear debugging strategy to identify existence of errors, and then find the cause
- includes a sequence of test cases of increasing difficulty, testing out more and more of the code

Programming and debugging

In developing `laplace3d`, a 3D finite difference code, my approach was to

- first write CPU code for validation
- next check/debug CUDA code with `printf` statements as needed, with different grid sizes:
  - grid equal to 1 block with 1 warp (to check basics)
  - grid equal to 1 block and 2 warps (to check synchronisation)
  - grid smaller than 1 block (to check correct treatment of threads outside the grid)
  - grid with 2 blocks
- then turn on all compiler optimisations

Performance improvement

The size of the thread blocks can have a big effect on performance:

- often hard to predict optimal size *a priori*
- optimal size can also vary significantly on different hardware
- optimal size for `laplace3d` with a 128^3 grid is
  - 32×4 on Tesla
  - 128×2 on Fermi

  I think I know why now, but it was a surprise at the time
- we're not talking about just 1-2% improvement, can easily be a factor $2\times$ by changing block size

Performance improvement

A number of numerical libraries (e.g. FFTW, ATLAS) now feature auto-tuning – optimal implementation parameters are determined when the library is installed on the specific hardware

I think this is going to be important for GPU programming:

- write parameterised code
- use optimisation (possibly brute force exhaustive search) to find the optimal parameters
- an Oxford student, Ben Spencer, has developed a simple flexible automated system to do this `http://mistymountain.co.uk/flamingo/`

Performance improvement

Use profiling to understand the application performance:

- where is the application spending most time?
- how much data is being transferred?
- are there lots of cache misses?
- on Fermi, there are a number of on-chip counters can provide this kind of information

The CUDA 4.0 profiler is greatly improved

- provides lots of information (a bit daunting at first)
- gives hints on improving performance

Going further

In some cases, a single GPU is not sufficient

Shared-memory option:

- single system with up to 8 GPUs
- single process with a separate host thread for each GPU, or use just one thread and switch between GPUs (new CUDA 4.0 capability)
- in CUDA 4.0 can also now transfer data directly between GPUs

Distributed-memory option:

- a cluster, with each node having 1 or 2 GPUs
- MPI message-passing, with separate process for each GPU

Going further

Keep a watchful eye on what is happening in computing

NVIDIA:

- Kepler due out in 2012
  - $3\times$ improvement in performance per watt
  - PCIe gen 3 ($2\times$ improvement)
- Maxwell due in 2013
  - another $3\times$ improvement in performance per watt
- project Denver due in 2013
  - adds some ARM cores to Maxwell GPU
  - no longer need a CPU or PCIe bus?
- lots of supercomputers, by CRAY and others, built on NVIDIA GPUs

Going further

AMD:

- AMD has had good GPU hardware, but aimed purely at graphics & games, not HPC
- some signs that is changing with increased effort on OpenCL software
- APU (Accelerated Processing Unit) called Llano
  - combined CPU/GPU supporting OpenCL/DirectX 11
  - 29GB/s, 500GFlops single precision
  - aimed at laptops/desktops, not HPC
- new high-end GPU, Radeon HD 7970

Going further

Intel:

- latest “Sandy Bridge” CPU architecture
  - some chips have built-in GPU, purely for graphics
  - 4–10 cores, each with a 256-bit AVX vector unit
- MIC (Many Integrated Core) architecture
  - 32–48 cores, each with a 512-bit vector unit
  - I’m told performance is comparable to Fermi and power consumption (300 watts)
  - new supercomputers based on this (Stampede)

ARM:

- already designed OpenCL GPUs for smart-phones
- will design much more powerful GPUs in the future

Going further

My current software assessment:

- CUDA is dominant in HPC, because of ease-of-use and NVIDIA dominance of hardware
  - PGI has developed a FORTRAN CUDA compiler
  - PGI also developing capability to compile CUDA to generate AVX vector code for Intel CPUs
- OpenCL is the multi-platform standard, but currently only used for low-end mass-market applications
  - computer games
  - HD video codecs

Going further

- Intel is promoting a confusing variety of alternatives for MIC and multicore CPUs with vector units
  - auto-vectoring compiler
  - Ct/ABB (array building blocks)
  - OpenCL
  - vector operations

- Microsoft has just announced a new initiative: C++ Accelerated Massive Parallelism (C++ AMP) which will target GPUs and multicore CPUs

Final words

- exciting times for HPC
- the fun will wear off, and the challenging coding will remain – computer science objective should be to simplify this for application developers through
  - libraries
  - domain-specific high-level languages
  - code transformation
  - better auto-vectorising compilers
- confident prediction: GPUs and other accelerators / vector units will be dominant in HPC for next 5-10 years, so it's worth your effort to re-design and re-implement your algorithms