

Initial planning

Lecture 7: tackling a new application

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Lecture 7 – p. 1/26

1) Has it been done before?

- check with Google
- ask a local expert
- check CUDA sample codes
- sign up to the CUDA Developer Program (free) and check out relevant Video-on-Demand talks from the last GTC (GPU Technology Conference)
- check out the NVIDIA Developer blogs:
<https://developer.nvidia.com/blog>
(very good for info on new hardware architectures as well as new software features)

Lecture 7 – p. 2/26

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!

Lecture 7 – p. 3/26

Initial planning

Sometimes you need to think about “the bigger picture”

Already considered 3D finite difference example:

- lots of grid nodes so lots of inherent parallelism
- even for ADI method, a grid of 256^3 has 256^2 tri-diagonal solutions to be performed in parallel so OK to assign each one to a single warp
- but what if we have a 2D or even 1D problem to solve?

Lecture 7 – p. 4/26

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

Lecture 7 – p. 5/26

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?

Lecture 7 – p. 7/26

Initial planning

2D:

- 128KB of shared memory on Ampere == 32K float so grid of 64^2 could be held within shared memory
 - one kernel for entire calculation
 - each block handles a separate 2D problem; possibly two block per SM
- for bigger 2D problems, might need to split each one across more than one block
 - separate kernel for each timestep / iteration

Lecture 7 – p. 6/26

Initial planning

Parallel Cyclic Reduction (PCR): starting from

$$a_n x_{n-1} + x_n + c_n x_{n+1} = d_n, \quad n = 0, \dots, N-1$$

with $a_0 = c_{N-1} = 0$, subtract a_n times row $n-1$, and c_n times row $n+1$ and re-normalise to get

$$a_n^* x_{n-2} + x_n + c_n^* x_{n+2} = d_n^*$$

with $a_m^* = 0$ for $m < 2$ and $c_m^* = 0$ for $m \geq N-2$.

Repeating this $\log_2 N$ times gives the value for x_n (since the values of the final a 's and c 's will be zero) and each step can be done in parallel.

(Practical 7 uses shared memory, but if $N \leq 32$ it fits in a single warp and can be implemented using shuffles.)

Lecture 7 – p. 8/26

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 in lecture 5?
- 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

Lecture 7 – p. 9/26

Initial planning

5) Is there a problem with host \leftrightarrow device bandwidth?

- usually best to move whole application onto GPU, so not limited by PCIe v4 bandwidth (32GB/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

Lecture 7 – p. 11/26

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

Lecture 7 – p. 10/26

Initial planning

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

- break-even point is roughly 50 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 \implies changes approach to implementation

Lecture 7 – p. 12/26

Initial planning

If compute-intensive:

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

Lecture 7 – p. 13/26

Initial planning

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

- look for NVIDIA sample codes demonstrating use of the feature
- 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!

Lecture 7 – p. 15/26

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)

Lecture 7 – p. 14/26

Initial planning

Read NVIDIA documentation on performance optimisation:

- Section 5 of CUDA C++ Programming Guide
- CUDA C++ Best Practices Guide
- Volta Tuning Guide
- Ampere Tuning Guide
- Hopper Tuning Guide

Lecture 7 – p. 16/26

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

Lecture 7 – p. 17/26

Programming and debugging

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.

Reminder:

```
compute-sanitizer --tool racecheck  
compute-sanitizer --tool synccheck  
to check for race condition and deadlock
```

Lecture 7 – p. 19/26

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 major projects I have a `cpp` flag `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

Lecture 7 – p. 18/26

Programming and debugging

In developing `laplace3d`, 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

Lecture 7 – p. 20/26

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 on different hardware
- with early GPUs, could gain 2× improvement by re-optimising the block sizes
- probably not as much change these days between successive generations

(not so much change in SMs, more a change in the number of SMs, the size of L2 cache, and new features like Tensor Cores)

Lecture 7 – p. 21/26

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 a good idea for GPU programming, though I have not seen it used by others:

- write parameterised code
- use optimisation (possibly brute force exhaustive search) to find the optimal parameters
- an Oxford student, Ben Spencer, developed a simple flexible automated system to do this – can try it in one of the mini-projects

Lecture 7 – p. 22/26

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?
- there are a number of on-chip counters to provide this kind of information

The Nsight Compute profiler is powerful

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

The Nsight Systems profiler gives a top-level view and is relatively easy to use.

Lecture 7 – p. 23/26

Going further

In some cases, a single GPU is not sufficient

Shared-memory option:

- single system with up to 16 GPUs
- GPUs linked by either PCIe (direct or via CPU) or NVlink (much faster)
- single process with a separate host thread for each GPU, or use just one thread and switch between GPUs
- can transfer data directly between GPUs – NVIDIA software will use the fastest route, avoiding the CPU if possible

Lecture 7 – p. 24/26

Going further

Distributed-memory option:

- a cluster, with each system having 1 or 2 GPUs
- systems connected by high-speed Ethernet/Infiniband networking with PCIe network cards
- simplest approach is MPI message-passing, with separate process for each GPU
- modern MPI software has full support for CUDA, with direct data transfers (no intermediate copies in CPU) where possible

<https://developer.nvidia.com/mpi-solutions-gpus>

<https://developer.nvidia.com/gpudirect>

Final words

- HPC continues to be exciting – the performance of the latest hardware keeps advancing
- coding to get a good fraction of peak performance remains challenging – computer science objective should be to simplify this for developers through
 - libraries
 - domain-specific high-level languages
 - code transformation
- GPUs will remain dominant in HPC for next 10 years, so it's worth your effort to re-design and re-implement your algorithms