Lecture 7: tackling a new application

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Initial planning

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)

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”

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

  (optional lecture 8 on how best to solve tri-diagonal equations on GPUs – involves doing more computation to reduce the amount of communication)
- 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:

- 128KB of shared memory == 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

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

Parallel Cyclic Reduction (PCR): starting from

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

with $a_m \equiv 0$ for $m<0$, $m \geq N$, subtract a_n times row $n-1$, and c_n times row $n+1$ and re-normalise to get

$$a^*_n\, x_{n-2} + x_n + c^*_n\, x_{n+2} = d^*_n$$

Repeating this $\log_2 N$ times gives the value for x_n (since $x_{n-N} \equiv 0, x_{n+N} \equiv 0$) and each step can be done in parallel.

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

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?
- 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 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

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 40 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
- 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

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)

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!

Initial planning

Read NVIDIA documentation on performance optimisation:

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

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: use `cuda-memcheck --tool racecheck` to check for race condition

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

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 $2\times$ 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)

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

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

Going further

In some cases, a single GPU is not sufficient

Shared-memory option:

- single system with up to 16 GPUs
- single process with a separate host thread for each GPU, or use just one thread and switch between GPUs
- can also 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

NVIDIA

Keep an eye on what is happening with new GPUs:

- Volta came out in 2017/18:
  - V100 for HPC
  - 32GB HBM2 memory
  - special “tensor cores” for machine learning (16-bit multiplication + 32-bit addition for matrix-matrix multiplication) – much faster for TensorFlow
- Ampere came out in 2020:
  - A100 for HPC
  - 108 SMs
  - 40-80 GB HBM2 memory
  - wider range of “tensor cores” capabilities

NVIDIA

- NVIDIA DGX Station A100
  `https://www.nvidia.com/en-us/data-center/dgx-station-a100/`
  - 4 NVIDIA A100 GPUs, each with 80GB HBM2
  - 64-core AMD CPU
  - 512 GB DDR4 memory, 10 TB SSD
  - 600GB/s NVlink interconnect between the GPUs
- NVIDIA DGX A100 Deep Learning server
  `https://www.nvidia.com/en-us/data-center/dgx-a100/`
  - 8 NVIDIA A100 GPUs, each with 80GB HBM2
  - 2 $\times$ 64-core AMD “Rome” CPUs
  - 2 TB DDR4 memory, 30 TB SSD
  - 600GB/s NVlink interconnect between the GPUs

NVIDIA

- Hopper is coming out in 2023:
  - H100 for HPC
  - 228-264 SMs
  - 80GB HBM2 memory
  - 40MB L2 cache
  - NVlink improvements – up to 50% faster
  - PCIe v5.0 – $2\times$ improvement
- Grace CPU also coming out in 2023:
  - Arm-based
  - up to 72 cores
  - 550GB/s bandwidth to LPDDR5X memory
  - 3 TB/s NVlink connection to Hopper GPU

AMD

- over past decade AMD has had excellent CPUs and GPUs (and pioneered chiplet packaging) but has not invested enough in software
- "Genoa" Zen4 EPYC CPUs:
  - up to 64 cores, each with vector units
  - up to 384MB L3 cache
  - now getting about 20% server market share
- Instinct GPUs:
  - up to 104 Compute Units, each with 64 stream procs
  - 64 GB HBM2e memory, up to 1.6 TB/s bandwidth
  - programmed using OpenCL, or HIP (Heterogeneous computing Interface for Portability) with translation to/from CUDA

    `https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html`

Intel

- current "Ice Lake" Xeon-SP CPUs:
  - up to 40 cores, each with one or two 512-bit AVX-512 vector units per core (512 bits = 16 floats)
  - 60MB L3 cache (shared), 1.25MB L2 cache per core
  - up to 200 GB/s memory bandwidth
- new "Sapphire Rapids" Xeon-SP CPUs:
  - due to ship in January 2023
  - CPU Max variants have up to 64 GB HBM2e memory
- "Ponte Vecchio" a.k.a. Data Center GPU Max:
  - 128 Xe cores, each with 16 vector units
  - 400MB L2 cache, 64GB HBM2 memory
  - due to ship soon

Outlook

My current software assessment:

- CUDA is dominant in HPC, because of
  - ease-of-use
  - NVIDIA dominance of hardware, with big sales in games/VR, machine learning, supercomputing
  - extensive library support
  - support for many different languages (Fortran, Python, R, MATLAB, etc.)
  - extensive eco-system of tools
- OpenCL is the multi-platform standard, but currently only used for low-end mass-market applications (games, phones, tablets) – AMD and Intel have not invested enough in software support

Final words

- it continues to be an exciting time for HPC
- 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
  - better auto-vectorising compilers
- confident prediction: GPUs and other accelerators such as vector units will remain dominant in HPC for next 10 years, so it's worth your effort to re-design and re-implement your algorithms