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

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 $128^3$ 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

Lecture 7 – p. 5

Initial planning

2D:
- 64KB of shared memory == 16K 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

Lecture 7 – p. 6
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?

Parallel Cyclic Reduction (PCR): starting from

\[ a_n x_{n-1} + x_n + c_n x_{n+1} = d_n, \quad n = 0, \ldots, N-1 \]

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 on Kepler hardware 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

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 (12GB/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 \(\Rightarrow\) changes approach to implementation

If compute-intensive:
- don’t worry (too much) about cache efficiency
- minimise integer index operations – surprisingly costly (this changes with Volta which has separate integer units)
- 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)

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
- Kepler Tuning Guide
- Maxwell Tuning Guide
- Pascal 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

Lecture 7 – p. 19

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

Lecture 7 – p. 20
Programming and debugging

In developing \texttt{laplace3d}, my approach was to
\begin{itemize}
  \item first write CPU code for validation
  \item next check/debug CUDA code with \texttt{printf} statements as needed, with different grid sizes:
    \begin{itemize}
      \item grid equal to 1 block with 1 warp (to check basics)
      \item grid equal to 1 block and 2 warps (to check synchronisation)
      \item grid smaller than 1 block (to check correct treatment of threads outside the grid)
      \item grid with 2 blocks
    \end{itemize}
  \item then turn on all compiler optimisations
\end{itemize}

Performance improvement

The size of the thread blocks can have a big effect on performance:
\begin{itemize}
  \item often hard to predict optimal size \textit{a priori}
  \item optimal size can also vary significantly on different hardware
  \item optimal size for \texttt{laplace3d} with a $128^3$ grid was
    \begin{itemize}
      \item $128 \times 2$ on Fermi generation
      \item $32 \times 4$ on later Kepler generation
    \end{itemize}
  \item at the time, the size of the change was a surprise
  \item we’re not talking about just 1-2\% improvement, can easily be a factor $2 \times$ by changing block size
\end{itemize}
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, developed a simple flexible automated system to do this – can try it in one of the mini-projects

Lecture 7 – p. 23

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 can provide this kind of information

The CUDA profiler is great

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

Lecture 7 – p. 24
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

Going further

Keep an eye on what is happening with new GPUs:

- Pascal came out in 2016:
  - P100 for HPC with great double precision
  - 16GB HBM2 memory → more memory bandwidth
  - NVlink → 4×20GB/s links per GPU

- Volta came out in 2017/18:
  - V100 for HPC
  - 32GB HBM2 memory
  - roughly 50% faster than P100 in compute, memory bandwidth, and 80% faster with NVlink2
  - special “tensor cores” for machine learning (16-bit multiplication + 32-bit addition for matrix-matrix multiplication) – much faster for TensorFlow
Going further

Two GPU systems:

- NVIDIA DGX-1 Deep Learning server
  - 8 NVIDIA GV100 GPUs, each with 32GB HBM2
  - $2 \times 20$-core Intel Xeons (E5-2698 v4 2.2 GHz)
  - 512 GB DDR4 memory, 8TB SSD
  - 150GB/s NVlink interconnect between the GPUs

- NVIDIA DGX-2 Deep Learning server
  - 16 NVIDIA GV100 GPUs, each with 32GB HBM2
  - $2 \times 24$-core Intel Xeons (Platinum 8168)
  - 1.5 TB DDR4 memory, 32TB SSD
  - NVSwitch interconnect between the GPUs

JADE

Joint Academic Data science Endeavour

- funded by EPSRC under national Tier 2 initiative
- 22 DGX-1 systems (with older Pascal P100s)
- 50 / 30 / 20 split in intended use between machine learning / molecular dynamics / other
- Oxford led the consortium bid, but system sited at STFC Daresbury and run by STFC / Atos
- in operation for a year now

There is also a GPU system at Cambridge.
Going further

Intel:
- latest “Skylake” CPU architectures
  - some chips have built-in GPU, purely for graphics
  - 4–22 cores, each with a 256-bit AVX vector unit
  - one or two 512-bit AVX-512 vector units per core on new high-end Xeons
- Xeon Phi architecture
  - Intel's competitor to GPUs, but now abandoned

ARM:
- already designed OpenCL GPUs for smart-phones
- new 64-bit Cavium Thunder-X2 has up to 54 cores, being used in Bristol's new “Isambard” Cray supercomputer

Going further

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
  - computer games
  - HD video codecs
Final words

- it continues to be an exciting time 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

Lecture 7 – p. 31