Introduction to High Performance Computing with GPUs

Lorne Whiteway
lorne.whiteway@star.ucl.ac.uk

Astrophysics Group
Department of Physics and Astronomy
University College London

29 January 2019

Where to find this presentation

Find the presentation at https://tinyurl.com/yckg9mqa.

On this page click on 'Download' to get a copy of the presentation.

Motivation

- Based in part on material from the course *CUDA Programming on NVIDIA GPUs* by Mike Giles, Oxford.
  `http://people.maths.ox.ac.uk/~gilesm/cuda/`

- Opinions expressed are my own...

Two paradigms

- Intel (market cap \$215B[1]): One-chip-fits-all. Use the same chip (a CPU) for all tasks e.g. both word processing and high-performance computing (HPC).
- NVIDIA (market cap \$98B[1]): Create a specialised chip (the GPU) specifically tailored for certain tasks (including HPC).

[1] As of 28 January 2019

GPUs

- GPU = graphics processing unit
- Specialised chip
- Has many processors and a specialised memory structure
- Originally designed specifically for high performance via parallelisation when doing graphics rendering.
- Single instruction, multiple data (SIMD).
- This design also makes them suitable for parallelizable problems in HPC.
- Not-standalone - needs a CPU ‘host’.

GPU

Image credit: NVIDIA https://www.nvidia.com/en-us/data-center/tesla-v100/

GPUs are displacing CPUs for HPC

- GPUs offer better value (FLOPS/$) and energy usage (FLOPS/J) than CPUs.
- 4 of the world's top 7 supercomputers use GPUs, including the top supercomputer (*Summit*, Oak Ridge National Lab, 122.3 PFLOPS).[2]
- All else being equal, we should probably use GPUs for HPC...

[2] TOP500 list, November 2018 `https://www.top500.org/lists/2018/11/`

But...

- Installed base of CPUs (e.g. splinter).
- Takes time to develop GPU expertise.
- Program code needs to be altered to run (effectively) on GPUs.

GPUs on splinter

- New (July 2018) GPU ‘Tesla V100’ (in addition to old ‘K80’).
- Thanks to Ofer (funding) and Edd (implementation)!
- How can we use these cards effectively?

Tesla V100 specifications

Image credit: NVIDIA http://www.nvidia.com/content/PDF/Volta-Datasheet.pdf

	Tesla V100 PCIe	Tesla V100 SXM2
GPU Architecture	NVIDIA Volta	
NVIDIA Tensor Cores	640	
NVIDIA CUDA® Cores	5,120	
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS

Tesla V100 specifications

Image credit: NVIDIA http://www.nvidia.com/content/PDF/Volta-Datasheet.pdf

GPU Memory	16 GB HBM2	
Memory Bandwidth	900 GB/sec	
ECC	Yes	
Interconnect Bandwidth	32 GB/sec	300 GB/sec
System Interface	PCIe Gen3	NVIDIA NVLink
Form Factor	PCIe Full Height/Length	SXM2
Max Power Comsumption	250 W	300 W
Thermal Solution	Passive	
Compute APIs	CUDA, DirectCompute, OpenCL™, OpenACC	

Next problem...

- When you solve your biggest problem you simply get a new biggest problem.
- Large number of processors should slash computation wallclock time for parallelizable problems.
- But in fact we simply discover that speed of memory read/write becomes the new bottleneck.

GPU memory hierarchy

- GPUs have a *memory hierarchy* (moving down the list: faster, smaller, more expensive):
  - CPU ('host') core memory [but the GPU can't see this directly]
  - GPU ('device') core memory
  - Level 2 ('L2') cache
  - Level 1 ('L1') cache
  - Registers
- GPU-destined software must keep this hierarchy in mind.

CPU $\leftrightarrow$ GPU relatively slow

- Data transfer links as fast as possible. But e.g. CPU host $\leftrightarrow$ GPU device is relatively slow.
- Some problems are simply not worth doing on the GPU as it would be slower to copy the data to the GPU than it would be to do the calculation non-parallel on the CPU.
- Not worth copying a number to the GPU unless you are going to do roughly 100 FLOPS with it.
- But e.g. matrix multiplication is certainly worth doing on the GPU as copying data is $\mathcal{O}(N^2)$ while the calculation is $\mathcal{O}(N^3)$.

Parallel Programming - threads

- The code that runs on the GPU should create multiple threads; one thread will get attention from one processor.
- Generally you want many, many threads at the same time (to keep all the processors busy); if the problem is parallelizable then you should be able to organise the code to achieve this (e.g. each thread deals with part of the data).
- The threads will get batched into sets of 32 threads called *warps* and you want many such warps.

Parallel Programming - warps

- The 32 threads in a warp run in absolute lockstep - they all execute the same code at the same time on different data (SIMD).

- Consider

  `threadID < 8 ? f(x) : g(x);`

  here the first 8 threads calculate f while 24 threads twiddle their thumbs, **then** 24 threads calculate g while 8 threads twiddle their thumbs. Suboptimal.

GPU programming: CUDA

- CUDA is a language in which you can write code that will execute efficiently on a GPU.
- CUDA is `C`, plus templates, plus a few CUDA-specific language instructions. Use extention `.cu`.
- There is a CUDA compiler `nvcc` (I suspect that it's just a `C` preprocessor...)
- Can link with other C, C++, Fortran object files.

CUDA example - see handout

Introduction to GPUs cuda_example_code.cu

```
#include <stdlib.h>
#include <cuda.h>

// This is a kernel - it executes on the device (GPU), once per thread. Programmer's
// job is to write the kernel code from the standpoint of a single thread. The special
// variable 'threadIdx.x' is automatically provided - it tells you what your thread
// number is. In this example we call example_scalar_function on a single data array
// element.
__global__ my_kernel(float* d_data) {
      int tid = threadIdx.x;
      d_data[tid] = example_scalar_function(d_data[tid]);
}

// This is calling code that executes on the host (CPU). The purpose of this example
// function is to evaluate in parallel 'example_scalar_function' in place on every
// element of h_data.
void example(float* h_data, int data_size) {

  // Allocate memory on the device (GPU), and copy the data there.
  float* d_data;
  cudaMalloc((void **)&d_data, data_size*sizeof(float));
  cudaMemcpy(d_data, h_data, data_size*sizeof(float), cudaMemcpyHostToDevice);

  // Call kernel function.
  // This will create many threads and call 'example_kernel' in each thread.
  // In this example we create one thread for each element in the data array.
  int num_threads = data_size;
  my_kernel<<<1, num_threads>>>(d_data);

  // Copy back results from device memory to host memory, and free device memory.
  cudaMemcpy(h_data, d_data, data_size*sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(d_data);

  // CUDA exit - needed to flush printf write buffer.
  cudaDeviceReset();
}
```

CUDA example - host code (green)

- Need to copy data to device, so `malloc` then `memcpy`. But the host cannot see the device memory explicitly, so use special CUDA functions to do this.
- The 'kernel' is the code that will run on the GPU. Special syntax $\langle\langle\langle$`1, num_threads`$\rangle\rangle\rangle$ to specify number of threads; also has a normal argument list.

CUDA example - device code (yellow)

- Special keyword `__global__` identifies kernel function.
- Written from the point of view of one thread (like MPI, not like OpenMP).
- Special variable `threadIdx.x` automagically tells the kernel code what thread it is running in.

Organisation of threads

- 32 threads in a warp - they move in absolute lockstep.
- Up to 32 warps in a *block*. All the threads in a block can share some L1 cache memory.[3]
- As many blocks as you want (running on one GPU). All threads in the GPU can share some L2 cache memory.
- There are also methods for running across multiple GPUs...

[3] Maximum threads in a block depends on the GPU architecture

Organisation of threads

- Thus thread organisation is hierarchical (just as it was with memory organisation).
- This reflects a preference for ‘fine control at multiple levels’ (in the persuit of maximum performance) at the cost of simplicity of design.

CUDA code with multiple thread blocks

- In the caller:

```
int threads_per_block = 32;
int n_blocks = data_size / threads_per_block;
my_kernel<<<n_blocks, threads_per_block>>>(d_data);
```

- In the kernel:

```
int tid = threadIdx.x + blockDim.x * blockIdx.x
```

Synchronisation

- Threads in a warp move in lockstep but otherwise no guarantees of synchronicity - thread 145 might finish before thread 13 begins.
- There are functions to implement synchronisation (across threads in a block, or across all blocks).
- Can set up some shared memory for all the threads in a block.
- There are some *atomic* functions e.g. for modifying a global variable; your thread will briefly get a lock on the global variable.
- Very fast functions to allow threads in a warp to exchange information (warp shuffles).

Cache lines

- Data is read in a *cache line* of 64 bytes (8 doubles) - if you ask for only one double, you will still get 8. Only a few cache lines can be held in fast memory at a time.
- You should already be taking this into account when programming CPUs. For example, it makes sense to process an array sequentially, so as not to 'churn' the cache lines.
- Still an issue with GPUs, but here you need sensible interaction with the cache lines across *all* the threads in a warp.

Single or double precision?

- Should you use single or double precision? GPUs support both, but there's a performance difference (of the factor that you would expect).
- Of course this is also a concern with CPUs.
- CUDA supports templates, so it's easy to experiment.
- GPU also supports half-precision data type.

Streams

- The GPU can transfer data into and out of memory at the same time as it is doing calculations.
- Thus it makes sense to organise the code so that both operations happen at the same time (on different data sets, obviously).
- *Streams* are a CUDA construct to allow this.

Tuning

- The performance of an algorithm on a GPU often depends sensitively to CUDA parameters. (How many threads? How many threads per block? How many streams?)
- Can get e.g. ten-fold performance improvement with correct choice of parameters..
- Therefore crucial to experiment with different parameters.

Parallel algorithms - summation

- Summing N numbers can be done in $\log_2(N)$ (parallel) steps.
  In the first step, nodes 0 and 1 exchange data (so then each knows $x_0 + x_1$); simultaneously 2 and 3 exchange, 4 and 5 exchange, etc.
  In the second step, nodes 0 and 2 exchange subtotals (so then each knows $x_0 + x_1 + x_2 + x_3$), etc.;
  Etc.

- Warp shuffles make this easy and fast (at least for the first five steps).

- Similar logic used for the Fast Fourier Transform (FFT).

Parallel algorithms - other examples

- Many other problems (apparently sequential in nature) actually can be solved in parallel. How would you parallelise the following?
  - Partial summation (a.k.a. *scan* operation): calculate $x_0, x_0 + x_1, x_0 + x_1 + x_2, \ldots$
    This plays a role in parallel sorting algorithms.
  - Solve tridiagonal linear system.
- The answers to these questions have been known for decades (parallel computation is not a new idea...)

Parallel algorithms - FFT

- CUDA makes a parallel FFT algorithm available; the interface to this is `C`, not CUDA. Thus your `C` or `C++` code can calculate FFT on GPUs without you needing to write CUDA code.
- There are two interfaces: one mimics the FFTW interface (for easy transition) while the other is 'the right one'.

Interfaces

- Many users would probably be content to use interfaces that shielded them from the CUDA details (as with FFT).
- But the devil is in the detail; e.g. need some awareness of how CUDA works to ensure algorithm operates at peak efficency.
- CUDA FFT algorithm ‘self-tunes’ its parameters when first installed on a new system.

CUDA ecosystem

- SDK with many examples.
- Good documentation (except perhaps for the compiler).
- Online community.
- *Nsight*, a visual development environment (plug-in for Eclipse and Visual Studio).
- Excellent tools for debugging and for profiling.

Does CUDA have a long-term future?

- `C`, `C++` and `FORTRAN` have remained popular since they are close-to-ideal for abstractly expressing sequential algorithms. They have survived despite huge changes in computer architecture.
- Is CUDA similarly close-to-ideal for abstractly expressing parallel algorithms?
- Or is it too tied up with peculiarities of the GPU card? Will CUDA code go out-of-date as hardware changes?
- Also, CUDA is a proprietary language; NVIDIA appears to be 'nice' but its concerns are commercial ones.

CUDA alternatives

- *OpenCL* is an open standard that competes with CUDA. Used for smartphone apps, but otherwise CUDA appears to be winning.
- *Thrust* hides CUDA code behind a C++ STL-like interface.
- *Kokkos* is another C++ template library for GPU code.
- *OpenACC* appears to be an attempt to merge CUDA and OpenMP.
- Unclear which of these if any will win out.

Tensor cores

- Machine learing applications are commercially important enough for NDVIDIA to have built special processors ('tensor cores') onto the Tesla V100 card.
- These are optimised to perform matrix multiplications of 16-bit floats, but then accumulate the product as 32-bit doubles.
- This hardware can then be used by TensorFlow.
- NVIDIA will provide specialised hardware for your application only if you can guarantee billions of dollars of sales...

GPU users

- Gamers (and bitcoin miners, who have similar computational needs) are one category - the 'GForce' card is aimed at them.
- Data centres and machine learning is another category - the 'Tesla' card is aimed at them.
- Scientific HPC also uses 'Tesla' technology. My impression is that NVIDIA takes this customer base seriously, not because of sales volumes, but to encourage R&D exchange.

On the horizon

- Faster data exchange between GPUs.
- Direct flow of incoming real-time data onto the GPU; self-driving cars will do this.
- Both of these make the CPU less central.