The “Smooth” challenge

Sarah Tattersall, Jack Stevenson, Wael Aljeshi

March 3, 2013

Introduction

We have been provided with a piece of code that concerns a computational kernel extracted from a research project at Imperial College London on dynamic mesh adaptation[1]. The aim of this series of experiments was to figure out how to run this program as fast as possible on a given architecture.

We have been given access to the HPC's PC cluster to run the results of our experiment on. Since the HPC system uses the PBS Pro queuing system to manage the execution of jobs on the compute resources it can often take some time to run a single result. Since our project predomenantly involved porting the code to the GPU we also developed our code on the lab machines.

Technologies

Hardware

Labs The lab machines we ran our code on have an Intel Core i5 650 3.2GHz processor and a GeForce GT 330 CUDA enabled graphics card with compute capability 1.0.

CX1 The CX1 machine has...

Software

The softwares we have made use of on both lab machines and cx1 are:

- **CUDA Compiler Driver NVCC** which ...
- **CUDA Command Line Profiler**
- **Python** for scripting of both profiled results and benchmark scores

Hypothesis

Our hypothesis is that parallelising our code through coloring and running each color on the GPU simultaneously will provide a massive performance increase.

GPGPU 'is the utilization of a graphics processing unit (GPU), which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit (CPU)'[2]. GPU's are ideal for the smoothing algorithm that we have been provided with since although they can only process independent vertices and framgements they can process many of them in parallel and achieve SIMD performance. The transistors of a GPU are devoted to data processing(mainly integer and floating point)[3].

Since we can colour our mesh and run verticies of the same colour on in parallel the GPU seems like a natural choice for optimisation. Furthemore the smoothing algorithm makes use of many floating point operations and so we should see a very large speed up from just porting to the CPU alone.

The CUDA Toolkit provides a comprehensive development environment and allows for easy migration of code.

Colouring

We cannot independently smooth all verticies in the mesh simultaniously since a vertex depends on it's adjacent verticies. If these were smoothed at the same time the results would not be correct and so these must be locked down. We can however colour verticies of the mesh and run each independent set simultenously on the GPU. Coloring is the first step that must be performed when porting the code across to the GPU. We will experiment with a greedy algorithm that finds the first possible colour a vertex can fall in, and thus may not prove to be an optimal solution but will provide satisfactory results.

Migration to GPU

In order to run the code on the GPU we must serialize 2D data structures. For example consider a vector of vectors:

```
std::vector< std::vector<size_t> > colours = {[1, 5, 9, 2],
                                              [0, 4, 8],
                                              [3, 6, 7]};
```

Since we do not have access to the standard library on a GPU the easiest and most effecient option is to turn this into a 1D array.

```
size_t[] colours = {1, 5, 9, 2,
                    0, 4, 8,
                    3, 6, 7};
size_t[] colour_indicies = {0 , 4, 7, 10};
```

In each case must add an extra indicies array, in this case `colour_indicies`, to exactly where the beginning of each colour is in `colours`. The array must be of size one larger than the number of items so that we can perform size calculations of the individual groups inside for looping purposes.

Next we must copy input data from the CPU memory to GPU memory which can be done with the `cuMemAlloc` function to allocate memory, `cuMemcpyHtoD` to copy from host to device and a similar method to copy data back. All setup code for running on the GPU is in `CUDATools.h`

Finally we must adapt the smoothing algorithm for the GPU in a number of steps:

- **Modify the code to smooth a single vertex rather than looping through them all.**
  We can calculate the vertex id easily using block and thread indexing like so:

  ```
  const size_t threadID = blockIdx.x * blockDim.x + threadIdx.x;
  if(threadID >= NNodesInSet)
    return;
  size_t vid = colourSet[threadID];
  ```

  We must be careful to ensure that we do not access beyond the end of arrays and so the if statement above checks this for us.

- **Modify mesh methods for running on the GPU**.
  Since we will not have access to the mesh object on the GPU we will no longer be able to call methods such as `mesh->isCornerNode(vid);`. All methods needed by the smoothing algorithm must be declared as `__device__` methods and re-written.

It is however, not enough to stop at just running the code on our GPU. We must also then optimise for the GPU to achieve maximum performance. Therefore we propose the following.

Branching

Unlike CPU cores GPU's offer no branch prediction or speculative execution. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps (32 parallel threads), and these execute one common instruction at a time. At branch statements threads can diverge however, the warp will serially execute each branch path taken for all threads (but just disabling them) and when paths complete the threads will converge.[4]. This will kill the performance of the GPU if many threads within a warp.

A major source of branching in the smoothing algorithm comes from the fact that different verticies have different numbers of neighbours. The three main loops (calculating worst coords, calculating the new worst coords, and assembling matricies A & q) all depend upon looping over a verticies neighbours and thus each warp will end up executing the loop the maximum number of neighbours times.

We will therefore investigate the following:

- **Minimising branching** In order to avoid this we will investigate minimising the branches by ordering the individual verticies in order of the number of neighbours they have. This will reduce branching since in most warps all threads will process verticies with the same degree and only a few unlucky warps that have a transition in number of neighborus will do some unnecessary work. For example if the verticies in color 0 were 1, 4, 6, 7, 10, 12 and we have 0, 4, 6, 12 have 6 neighbours, 1 and 7 have 5, and 10 has 4 we would re-order them to be 10, 1, 7 0, 4, 6, 12.

- **Removing corner nodes during coloring** The smoothing algorithm bypasses any corner nodes and so we can avoid the conditional branch:

  ```
  if(isCornerNode(vid)) {
    return;
  }
  ```

  By factoring out corner nodes during the colouring phase it removes the need for this branch entirely and so should hopefully provide a slight speed up.

Pinning

Host data allocations are pageable by default which means that the GPU cannot access data directly from pageable host memory. When a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a pinned host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory[5], as illustrated below.

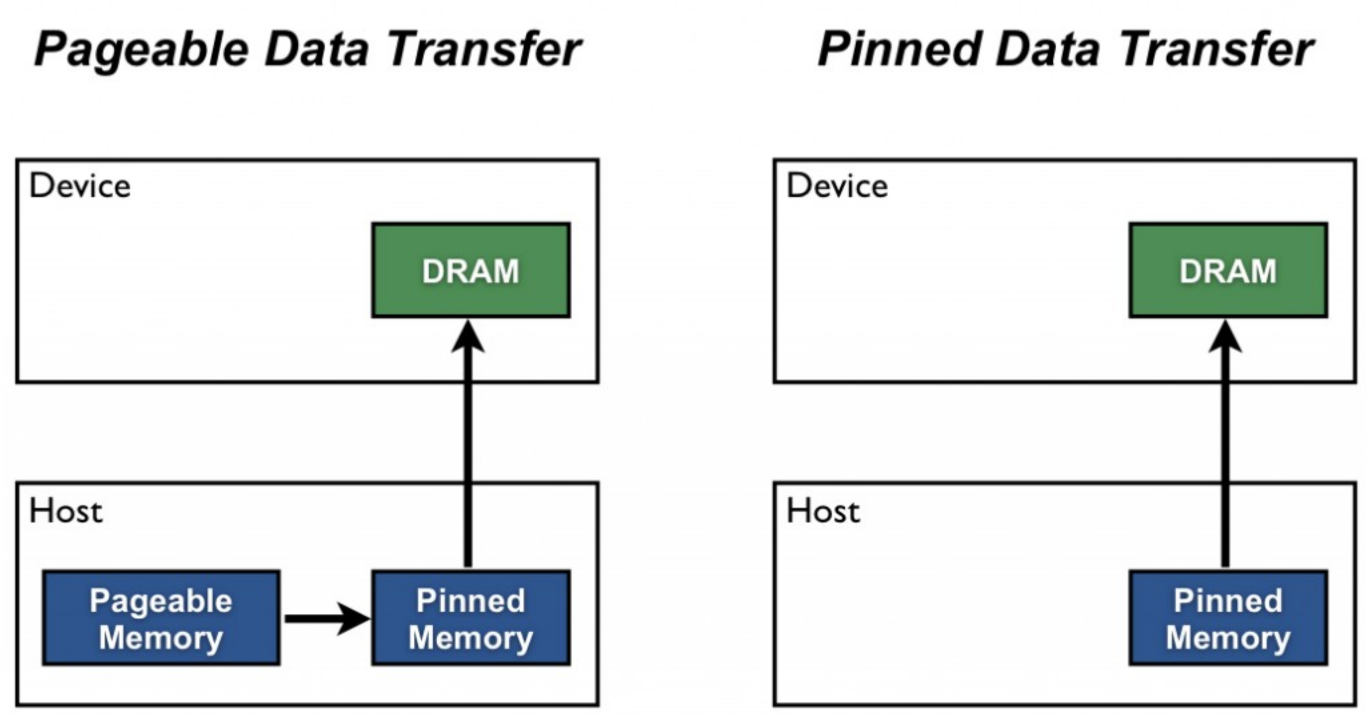

Figure 1: GPU Data Transfers[5]

Pinned memory is used as a straging area for transfers and we can avoid this cost by directly allocating our host arrays (e.g colours) in pinned memory.

Since we do not know the size of our 2D data arrays we will allocate all our data in normal memory using the standard library data structurs and then in a pre-processing stage call a `pin_data` method on the mesh and similarly with the colours. With extra code our data structures could be pinned straight away, however we perform this pre-processing stage as a comprimise to allow us to spend our time on GPU optimisations rather than speeding up the CPU code which does not count in the benchmark timings.

We can easily lift the code we will write for serializing data structures `CUDATools.h` into the Mesh, so this should hopefully reduce time spent on this optimisation.

The Potential Advantage

Effectiveness

Conclusion

Group Work

References

[1] Dynamic Mesh Adaptation Project. `https://launchpad.net/pragmatic`.

[2] Wikipedia GPGPU. `http://en.wikipedia.org/wiki/GPGPU`.

[3] George Rokos. AMD's Cayman Architecture. University Lecture, 2013.

[4] CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[5] How To Optimise Data Transfers In CUDA C/C++. `https://developer.nvidia.com/content/how-optimize-data-transfers-cuda-cc`.