CUDA Programming on NVIDIA GPUs
Mike Giles

**Thalesians workshop: Monte Carlo and Finite Difference examples**

Morning session:

1. Log into the Oxford University "skynet" cluster and install the CUDA 2.3 module by following the instructions provided, and then use the command

   `qsub -IVX`

   to create an interactive session on one of the compute nodes.

2. Use the command

   `cp -r ~mgiles/cuda_course/prac2 ~/prac2`

   to copy the directory `prac2` from my account to yours.

3. Read through the `prac2.cu` source file and note the following:

   - The way the code is split into one part which runs on the host processor, and another part which runs on the GPU, with explicit transfer of data between the two.
   - The use of the NAG CUDA RNG library for random number generation.
   - The use of `__constant__` memory defined to have global scope for all kernel routines (i.e. it is defined for the lifetime of the entire application, not just the lifetime of a single kernel routine, and it can be referenced by any kernel routine) and the way in which the data is initialised by copying values over from the host.
   - The use of `hTimer` to time the execution of various parts of the code.

4. Use the `Makefile` to compile the code (no debug or emulation) and then run the code in `bin/release` and see the timings it gives.

5. In the source file, uncomment the "Version 2" lines of code, and comment out the "Version 1" lines. Re-compile and re-run the code to see the effect of this on the kernel execution time.

6. Use the command

   `cp -r ~mgiles/cuda_course/libor ~/libor`

   to copy the directory `libor` from my account to yours.

7. Use the `Makefile` to compile the code (no debug or emulation) and then run the code in `bin/release` and see the timings it gives.

8. Read through the source files, and note in particular the way the computation is performed on both the CPU and GPU, and the results compared to check that the GPU computation is correct.

9. Try using the visual profiler to profile the GPU execution and check that the memory transfers are coalesced.

Afternoon session:

1. Use the command

   `cp -r ~mgiles/cuda_course/laplace3d ~/laplace3d`

   to copy the directory `laplace3d` from my account to yours. This has two codes to solve a 3D Laplace equation, a "new" code which uses texture mapping, and an "old" version which does not. Both perform the calculation on both the GPU and the CPU to check that they give the same answers, and they also time how long they take.

2. Using the Makefile, compile and run the code `laplace3d_new`.

3. By modifying the Makefile (removing all `_new` bits) compile and run the code `laplace3d`.

4. Read through `laplace3d_new.cu`, `laplace3d_new_kernel.cu` and `laplace3d_gold.cpp` (the CPU reference code).

   In particular, note:

   - The grid is cut into pieces of size $32 \times 4$ in the $x - y$ direction, and each thread block uses 128 threads, with each thread processing one element in each 2D plane.
   - The `cudaMallocPitch` memory allocation in the main code rounds up the memory allocation for each row in the first ($x$) coordinate direction so that each row starts on a multiple of 16; this ensures memory coalescence later on when writing to the `u2` array.
   - In the kernel code, `IOFF, JOFF, KOFF` give the memory offsets in the three coordinate directions.

5. Have a look also at `laplace3d.cu` and `laplace3d_kernel.cu` and the notes in `laplace3d.pdf` which are also available at
   http://people.maths.ox.ac.uk/~gilesm/codes/laplace3d/laplace3d.pdf

   The main thing to note is how much more complex the programming is. In this simple example which involves very little computation it gives a factor 2 improvement in performance, but in cases involving more computational effort the texture mapping version will probably be almost as efficient and involves much simpler programming.