CUDA Programming on NVIDIA GPUs
Mike Giles

Practical 3: Laplace solver

The main objectives in this practical are to learn about:

- a simple "naive" implementation of a finite difference solver
- how improved performance can be achieved by using shared memory to enable data re-use, but it requires very careful attention to detail to achieve memory coalescence

What you are to do is as follows:

1. Using the Makefile, compile and run the code `laplace3d_naive`.
2. Read through `laplace3d_naive.cu`, `laplace3d_naive_kernel.cu` and `laplace3d_gold.cpp` (the CPU reference code).

   In particular, note:

   - The total grid size is `NX`×`NY`×`NZ`, where `NX, NY, NZ` are parameters set in the host code.
   - The grid is cut into pieces of size `BLOCK_X`×`BLOCK_Y` in the $x-y$ direction, and each thread block uses `BLOCK_X*BLOCK_Y` threads, with each thread processing one point in each 2D plane. The parameters `BLOCK_X, BLOCK_Y` are defined as literal constants in the host code.
   - The blocks and the threads are both identified with 2D indices, unlike the 1D indices used in Practicals 1 and 2.
   - In the kernel code, `IOFF, JOFF, KOFF` give the memory offsets in the three coordinate directions.
   - A variable `active` is used to limit computation to points within the grid.
   - There is a "gold" computation on the CPU to check that the results produced by the GPU are correct. This kind of validation is used in most of the CUDA SDK examples.

   The code is relatively short, so try to understand it completely. Please ask questions if anything is not clear.

3. Why does `active` need to be defined and used the way it is? Can you identify particular threads in certain blocks which lie outside the computational grid? Hint: `NX` is not a multiple of `BLOCK_X`.

4. Try varying the values of `BLOCK_X, BLOCK_Y` to see if you can get the code to run faster.

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

6. 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 approximately a factor 3 improvement in performance, but in cases involving more computational effort the "naive" version will probably be almost as efficient and involve much simpler programming.

   Also note the `cudaMallocPitch` memory allocation in the main code which 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 reading from the `u1` array and writing to the `u2` array, but it complicates things by requiring the use of the `pitch` variable to get the correct memory offsets.

7. Try commenting out the `__syncthreads();` instructions in `laplace3d_kernel.cu`. See what happens to the error which is computed as the difference between the GPU results and the CPU results.

I think the difference in performance bewteen the "naive" and the optimized implementations will disappear on the new Fermi GPUs. In fact the "naive" version will probably be faster because it is simpler and involves fewer integer / indexing operations.