Introduction to CUDA Programming on NVIDIA GPUs
Mike Giles

Practical 3: finite difference equations

The main objectives in this practical are to learn about thread block optimisation and the optimal layout of memory for multi-dimensional applications.

This practical is based on a code which uses Jacobi iteration to solve a finite difference approximation of the 3D Laplace equation.

It performs the calculation on both the GPU and the CPU to check that they give the same answers, and also times how long it takes.

What you are to do is as follows:

1. Use Nsight (*build in Release mode!*) or the Makefile to produce the executable, `laplace3d`. Run it to see the results produced, and the times taken.

2. Read through `laplace3d.cu`, `laplace3d_kernel.cu` and `laplace3d_gold.cpp` (the CPU reference code).

   In particular, note:

   - The grid is cut into pieces of size 16×8 in the $x - y$ direction, and each thread block uses 128 threads, with each thread processing one element in each 2D plane.
   - In the kernel code, `IOFF, JOFF, KOFF` give the memory offsets in the three coordinate directions.

   The code is relatively short, so try to understand it completely, including the `u1, u2` pointer swapping in the main code, and the construction of the execution grid `(bx,by)`.

   Please ask questions if anything is not clear.

3. Try changing the thread block size to improve the performance – what are the optimal dimensions?

4. Read through the source code for the new version in `laplace3d_new.cu` and `laplace3d_kernel_new.cu`. Note that in the new version each thread handles a single grid point.

   Optimise its 3D thread block size, and compare its optimum running time to the original version.

5. For both the old code and the new code, reduce the number of iterations to 10, and count the number of integer and single precision floating point operations using the command line NVIDIA profiler:

   ```
   nvprof --metrics "inst_fp_32,inst_integer" laplace3d
   nvprof --metrics "inst_fp_32,inst_integer" laplace3d_new
   ```

   (The number of integer operations is surprisingly high – I think this is due to index arithmetic for the array references.)

6. Try changing the computational grid size from 256^3 to 255^3.

   You may find that this takes longer to execute! The reason is that the start of each line in the grid is no longer aligned with the start of a cache line. This can be "fixed" by "padding" the array so that the storage is of size $256 \times 255 \times 255$ with `JOFF=256` and `KOFF=255*JOFF`.

   Try this – you should now get better performance.

7. Estimate how much data is moved from the device memory into the GPU, and from the GPU back to the device memory, in each iteration.

   Given the execution time per iteration, what device memory bandwidth does this imply?

   Is this a good fraction of the peak bandwidth capability of the hardware?