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 way to handle multi-dimensional PDE applications. It also gives an introduction to two approaches to application profiling.

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. Click on the link in the course webpage to the Google Colab notebook.
2. Carefully follow the instructions in the notebook.
3. Read through `laplace3d.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.
4. Having verified that the GPU code and the Gold code produce the same answers to within machine precision, comment out the running of the Gold code and the comparison of the answers, and increase the grid size to 1024^3.
5. Try changing the thread block size to improve the performance – what are the optimal dimensions?

In a text cell, record the timings taken with different block sizes, and leave the final version of the code in the notebook with the optimal dimensions and the timing this gives.

6. Read through the source code for the new version in `laplace3d_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.

   Again record the timings taken with different block sizes, and leave the final version of the code in the notebook with the optimal dimensions and the timing this gives.

7. Count the number of integer and single precision floating point operations using the NVIDIA command line profiler:

   ```
   ncu --metrics "smsp__sass_thread_inst_executed_op_fp32_pred_on.sum,
   smsp__sass_thread_inst_executed_op_integer_pred_on.sum" laplace3d
   ```

   ```
   ncu --metrics "smsp__sass_thread_inst_executed_op_fp32_pred_on.sum,
   smsp__sass_thread_inst_executed_op_integer_pred_on.sum"
   laplace3d_new
   ```

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

8. 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. You can assume that the array `u2` is first transferred to the GPU before being written to which leads to it being transferred back to the device memory.

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

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

   (It is not clear to me whether the memory bandwidths quoted by NVIDIA are bi-directional, i.e. if they say the bandwidth is 300GB/s, does that mean the hardware can simultaneously achieve close to 300GB/s in each direction? If not, maybe it is more appropriate to sum the read and write bandwidths and compare the combined bandwidth to the hardware number.)

9. Submit your completed notebook to Emmanuel.