Introduction to CUDA Programming on NVIDIA GPUs
Mike Giles

Practical 2: Monte Carlo

The main objectives in this practical are to learn about:

- how to use `constant` memory on the graphics card, initialising it from the host
- how to use CUDA's timing functions to measure kernel execution times
- the importance of data layout when reading from (or writing to) the main graphics memory

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 the `prac2.cu` source file which find in the notebook.

   Note 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.

   Note also the use of `cudaEvent` operations to time the execution of various parts of the code. The line

   ```
   cudaEventSynchronize(stop);
   ```

   is required to ensure that the previous operations have completed before the timer is stopped. This is because some operations such as kernel launching are asynchronous, i.e. the program starts the operation but doesn't wait for it to complete. This has the potential to give improved performance in some cases by over-lapping execution and communication, but it also has the potential to cause confusion when timing things.

   If you want to learn more about the mathematics of the Monte Carlo simulation in this practical, read these notes:
   https://people.maths.ox.ac.uk/gilesm/cuda/prac2/MC_notes.pdf

4. Follow the directions in the notebook to compile and run the code and see the timings it gives.

5. Make a duplicate copy of the source code and uncomment the "Version 2" lines of code, in 2 places, and comment out the "Version 1" lines. Again compile and run the code to see the effect of this on the kernel execution time.

6. For Version 1, work out why the 32 threads in a warp read in a consecutive block of 32 random numbers at the same time, but they don't in Version 2.

   If in doubt, use print statements to print out the indices of the array elements being loaded in.

7. Work out much data is being loaded from device memory by the kernel, and based on the execution time determine the effective transfer rate in GB/s.

   For the faster version of the kernel, is this close to the peak capability of the hardware (approximately 300GB/s for the T4 GPU) ?

8. Write your own small program to compute the average value of

$$az^2 + bz + c$$

   where z is a standard Normal random variable (i.e. zero mean and unit variance, which is what the random number generator produces) and $a = 1.0, b = 2.0, c = 3.0$ are constants which you should store in `__constant__` memory.

   Use the same number of blocks and threads as the original code for `prac2.cu`. Use each thread to average over 100 values, and then write this average to a device array which gets copied back to the host for the averaging over the contributions from each of the threads.

   (Note: the average value should be very close to $a + c$.)

9. You might like to look at the code `prac2_device.cu` which is alternative version which generates the random numbers within the user kernel as they are required, using CURAND's device API. This avoids the need to store them in the main device memory.