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. Read through the `prac2.cu` source file.

   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.

2. Use the `Makefile` to compile the code and then run it and see the timings it gives.

3. 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.

4. Think about what happens when there is just 1 block of 32 threads. Work out which random numbers are read in by each thread, to understand why in Version 1 the 32 threads read in a contiguous block of 32 numbers at the same time, but they don't in Version 2.

   If in doubt, use print statements to print out the array elements being referenced.

If you have any questions, ask! Understanding how best to access data in device memory is very important for good CUDA performance so it's important you fully understand this.

5. 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?

   (The V100 GPU is capable of 900GB/s according to this NVIDIA datasheet.)

6. 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, b, c are constants which you should store in `__constant__` memory.

   I suggest you use each thread to average over 200 values, and then write this 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 close to $a + c$.)

7. The code `prac2_device.cu` has a second version of the code 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.

   Following the guidance in sections 3.5, 3.6 of the cuRAND documentation, it uses one small kernel to initialise the random number generators for each thread, and then a second kernel which does the computations. In real applications one would do the initialisation once and then usually run the main kernel many times.

   The host code also uses the CUDA functions `cudaGetDevice` and `cudaGetDeviceProperties` to find out the number of SMs in the GPU, and the function `cudaOccupancyMaxActiveBlocksPerMultiprocessor` to determine how many copies of the kernel function `pathcalc` can run at the same time within a single SM. This enables the host code to launch exactly the maximum number of blocks which can run simultaneously within the GPU, without any of them being queued for later execution.

   Compile and run the code to see the improved performance it gives.