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 ensuring "coalescence" 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 `hTimer` to time the execution of various parts of the code. The line

   ```
   cutilSafeCall( cudaThreadSynchronize() );
   ```

   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.

2. Use the `Makefile` to compile the code (no debug or emulation) and then run the code 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, and we have just 1 timestep (i.e. `N=1`). Work out which random numbers are read in by each thread, to understand why Version 1 has coalesced reads (the 32 threads read in a contiguous block of 32 numbers at the same time) whereas Version 2 does not.

   If in doubt, change the program parameters to reduce the number of paths, and run in debug mode with print statements to print out the array elements being referenced.

5. 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 100 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.

6. Continue browsing the NVIDIA SDK examples. There are several Monte Carlo and random number generation examples.