CUDA Programming on NVIDIA GPUs
Mike Giles

Practical 11: CPU threads and GPU streams

This practical is an introduction to streams and multi-threading.

- First, make and run `stream_legacy` and `stream_per_thread`.

  Notice the difference in run-time between the two executables.

- These two executables are built from the same source file `stream_test.cu`, but with different compiler flags – see the `Makefile`.

  `stream_legacy` behaves in the way which used to be standard. Kernels are being launched in multiple streams, but as soon as there is a kernel launch or a memory copy on the default stream it introduces a GPU synchronisation block; it doesn't start until all previously launched kernels in all streams have been completed, and any subsequent kernels in any stream cannot start until it has completed.

  `stream_per_thread` uses the compiler flag `--default-stream per-thread`. This application is not multi-threaded, and hence the flag is a little misleading. However, using this flag has an important side-effect even on single-threaded applications; it makes the default stream more similar to other streams, and in particular activity on the default stream does not block (and is not blocked by) activity on other streams.

- Read through the code carefully, and ask questions about anything which is not clear.

- Add print statements so that thread 0 in the compute kernels prints a message when the kernel starts, and finishes.

- Using multiple streams is useful when there are lots of different small kernels to be launched and executed concurrently. i.e. each kernel on its own does not make good use of a full GPU, but they are independent computations and collectively they full up a GPU.

  Experiment with this idea, for example by modifying the code in Practical 3 to perform multiple 3D computations with a range of small grid sizes (e.g. $50^3 - 100^3$).

- A second use, which particularly exploits the new `--default-stream per-thread` capability, is for overlapped computation and communication.

  Suppose that we are dealing with non-stop streaming data arriving in chunks (e.g. from a radio-telescope), and for each chunk we want to send it to the GPU, perform some GPU processing, and then transfer back the results to the CPU.

  Using the capabilities illusrated in this practical, write a code to do this, with a time-stepping loop in which, for the n^{th} step

  - stream 1 is used to process the n^{th} chunk of data
  - the default stream copies back to the CPU the output data from chunk $n-1$, then copies over to the GPU the input data for chunk $n+1$

- The second part of the practical has another source code `multithread_test.cu` which again is compiled in two different ways, with and without the flag `--default-stream per-thread`.

  Again notice the difference in run-time between the two executables.

- This example uses OpenMP multi-threading, with the number of OpenMP threads equal to the size of the `for` loop. Hence, each pass of the loop is handled concurrently by a separate thread, and therefore when the flag `--default-stream per-thread` is set, all of the kernels are executed concurrently because each thread has its own default stream.

  If you are familiar with OpenMP then you might like to explore this example and try modifying it to do other things.