CUDA Programming on NVIDIA GPUs
Mike Giles

Practical 12: Streams

The main objectives in this practical are to learn about:

1. how streams work
2. how different CUDA kernels can run in parallel
3. how to overlap host-device transfers and computations

What you are to do is as follows:

1. Read through the `kernel_overlap.cu` source file.

   Note how in the first case there are `nblocks` kernels launched, each with a single block of 1024 threads. Due to the strict FIFO ordering of individual streams (including the default stream), there can be no overlap between the `do_work` kernels, even though no single kernel can saturate the device.

   Review the use of CUDA streams in the second part of the code, specifically how kernels are scheduled to execute in different streams by specifying it in the `<<<...>>>` construct.

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

3. See how kernels overlap by profiling the application in the NVIDIA Visual Profiler. You can launch the profiler by typing `nvvp &`, then under "New session", you specify the compiled executable, and leave all other options at their defaults. After running your application, the profiler will show you a timeline - try zooming in and selecting individual kernels to see their details.

4. Read through the `work_streaming.cu` source file and understand what it does.

   The code represents an important use-case; data is being streamed to the GPU, then processed on the GPU, and once done, the results are copied off of the GPU. The biggest issue is that it takes a long time to copy data and to process it, and while data is being copied, no useful computations are happening and vice versa.

   With your knowledge of CUDA streams, try breaking up the copies and the processing into smaller pieces, so that while a host-to-device copy is ongoing, a kernel in a different stream can process data that was already copied, and

in yet another stream a device-to-host copy can execute, moving data that was already processed.

Remember to use pinned memory on the host (`cudaMallocHost`), and asynchronous copies (`cudaMemcpyAsync`).