Memory Reuse through shared memory in CUDA

The stenciling program provided exhibits reuse of memory in 3 dimenions meaning each thread will be re-reading the same block of memory multiple times in order to carry out the same computation. This can be optimized through the use of caching in shared memory. Shared memory exists as a cache within each SM and therefore, is much less expensive to access than the off chip global memory. By having each thread read their data into shared memory and performing future access from shared memory rather than global memory, faster access speeds can be achieved.

It is important to note that loading into shared memory is not free, especially around the fringes of computation. At the edge, a number of branch instructions take place which cause latencies to be exposed in the program. In our example, we have a small stencil size (6 reads, 2 in each direction) which may be realtively small compared to the branch overheads.

From the timing however, we still see that the `shared_stencil` implementation runs slightly faster than the naive implementaiton.

timing

The use of tiling, seperate from shared memory is also interesting to explore. The above nvprof shows the timing from a 16x16x1 tiled implementation where tiles exist in 2 dimensions. The profiler shows that the average timing for these the two implemenations was on the order of 300-400 us. By comparison, tiling with a tile size of 1x1x1 (i.e. no tiling) runs on the order of 7ms!. A significant slow down.

timing 1x1x1

Tiling through the use of large block sizes helps speed up the runtime for a number of reasons. Individual blocks correspond to a physical SM. Since SMs run the batches of threads groups of 32 (a unit known as a warp), running a block with less than 32 threads is just an inefficient use of hardware. Furthermore, when programs attempt to access similar pieces of data, (temporal locality), that data is primed in the data pipeline and left in the cache.