Lab 6

In this lab, we’ll revisit the framework for reductions laid out in the book’s discussion of shared memory.

Reductions

Copy `dot.cu` from the course directory. This is the dot product code from chapter 5, modified only slightly to make the file self-contained. In this lab, we’ll focus on the part of this program that performs a reduction (lines 55–66), adding the values calculated by each thread. A reduction takes an array (or list etc) of data values and combines them somehow into a single result. They are a common parallel operation so they are well worth investigating.

In the sample program, the reduction part is fairly uncontroversial; it reduces the total amount of data that must be transferred back from the GPU. What’s less clear is whether the entire dot product operation is worth doing on the GPU at all—after all, doing so incurs a hefty data movement cost to move the data to the GPU. Begin by investigating this; time the book’s approach to the dot product (not including creating the vectors but including the data transfers and all the calculations until the final result was finished). Compare this to a CPU-only approach; again time from when the vectors are created until the final result is computed. Is the dot product worth using the GPU for? Does the vector size matter? Note that you cannot use events for this part because part of what you are timing occurs on the CPU.

The comparison gets better (at least for the GPU) if we notice that the vectors can be created on it as well. Modify the kernel to also create the initial vectors and compare the running time of this version with the equivalent CPU-only version (one that must create the vector and then compute the dot product).

The `threadsPerBlock` parameter is also an important factor in the reduction time. Increasing it means that the reduction will require more stages, but also means that fewer blocks will be needed. The latter lowers the CPU’s part of the running time since each block becomes a value that must be transferred and then combined by the CPU. Try different values for `threadsPerBlock` and see how the running time is affected. Use GPU events to compute the GPU part of this and also time the entire process. Are the results what you expected?

If you have additional time, try out different reductions. Common reductions are computing the max, min, or the number of occurrences of a particular value. More complicated (requiring multiple cache values per thread) would be finding the location of the max/min or “binning” the values by counting the number within each of several possible ranges.