Parallel Prefix Sum
(Download ZipFile)

This MP is an implementation of a parallel prefix sum. The algorithm is also called "scan", and will be referred to as "scan" in this description. Scan is a useful parallel building block for many parallel algorithms, such as radix sort, quicksort, tree operations, and histograms. Scan applied to an array A will produce an array A', where

  A'[i] = A'[i-1] + A[i-1] : A[0] = 0

Or:

  A'[i] = sum(j = 0 to i-1) { A[j] } : A[0] = 0

While scan is an appropriate algorithm for any associative operator, we will be using addition. Please read Mark Harris's report "Parallel Prefix Sum (Scan) with CUDA" to learn the algorithmic background for this assignment. Note that you need to replicate any of their algorithms exactly, but this assignment does require a "work-efficient" algorithm.

1) Unzip the lab template into your SDK projects directory.

2) Edit the source files to complete the functionality of the array scan on the device. Use a tiled implementation that can process a very large array.

3) The modes of operation for the application are described here.

   No arguments: Randomly generate input data and compare against the host's result.

   One argument: Randomly generate input data, and write the result to file, name specified by the first argument.

   Two arguments: The first argument indicates the size of the array. Randomly generate input data, and write INPUT data to the second argument. (Good for generating test arrays).

   Three arguments: The first argument is a file name, the contents of which will be read for the array size. The second argument will be a file containing the input array. The third argument is a file name to write the resulting array.

   Note that if you wish to use the output of one run of the application as an input, you must delete the first line in the output file, which displays the accuracy of the values within the file. The value is not relevant for this application.

4) Report.

1. Near the top of scan_largearray.cu, set #define DEFAULT_NUM_ELEMENTS to 16000000, or 16777216 if your implementation does not handle arrays not a power of two in length. Set #define MAX_RAND to 3.

2. Copy and paste the performance results when run without arguments, including the host CPU and CUDA GPU processing times and the speedup.

3. Describe how you handled arrays not a power of two in size, and how you minimized shared memory bank conflicts. Also describe any other performance-enhancing optimizations you added.

4. How do the measured FLOPS rate for the CPU and GPU kernels compare with each other, and with the theoretical performance limits of each architecture? For your GPU implementation, discuss what bottlenecks your code is likely bound by, limiting higher performance.

Grading:

Your submission will be graded on the following parameters.

Demo/knowledge: 25%
- Computation runs on a G80, producing correct outputs files.

Functionality: 40%
- Correct handling of boundary conditions
- Uses shared, constant or texture memory to cover global memory latency.

Report: 35%