Homework 3
EECS 117

Stefan Cao (#79267250)
Andrew Yu (#23041544)

Part 1: Naive Parallel Reduction

Trial #	Input Vector Size (N)	Time to execute kernel (sec)	Effective Bandwidth (GB/s)
1	8388608	0.003381	9.92
2	8388608	0.003352	10.01
3	8388608	0.003358	9.99
4	8388608	0.003422	9.81
5	8388608	0.003354	10.00

$$Effective\ Bandwidth = \frac{Input\ vector\ size * Size\ of\ datatype}{Time\ to\ execute\ \ kernel * 10^9}$$

In other words, effective bandwidth is the message size over the total latency. The message size is the input vector size time the data type. The latency is the time to execute the kernel * 10^9 because we want to convert bytes/s to GB/s.

Part 2: Strided Access by Consecutive Threads

Trial #	Input Vector Size (N)	Time to execute kernel (sec)	Effective Bandwidth (GB/s)
1	8388608	0.00222	15.11
2	8388608	0.00222	15.11
3	8388608	0.002219	15.12
4	8388608	0.00222	15.11
5	8388608	0.00222	15.11

Average time to execute Naive = 0.0033734 s

Average time to execute Stride = 0.002198 s

$$\frac{0.0033734-0.002198}{0.0033734} * 100 = 34.84\% \textit{ speedup}$$

The Strided is 34.84% faster than the Naive.

Part 3: Sequential Access by Consecutive Threads

Trial #	Input Vector Size (N)	Time to execute kernel (sec)	Effective Bandwidth (GB/s)
1	8388608	0.001757	19.10
2	8388608	0.001768	18.98
3	8388608	0.001768	18.98
4	8388608	0.001756	19.11
5	8388608	0.001767	18.99

Part 4: First Add Before Reduce

Trial #	Input Vector Size (N)	Time to execute kernel (sec)	Effective Bandwidth (GB/s)
1	8388608	0.000958	35.03
2	8388608	0.000958	35.03
3	8388608	0.000963	34.84
4	8388608	0.000954	35.17
5	8388608	0.00096	34.95

Part 5: Unroll the Last Warp

Trial #	Input Vector Size (N)	Time to execute kernel (sec)	Effective Bandwidth (GB/s)
1	8388608	0.000637	52.68

2	8388608	0.000675	49.71
3	8388608	0.000657	51.07
4	8388608	0.000656	51.15
5	8388608	0.000659	50.92

Part 6: Algorithm Cascading

Trial #	Input Vector Size (N)	Time to execute kernel (sec)	Effective Bandwidth (GB/s)
1	8388608	0.000371	90.44
2	8388608	0.000396	84.73
3	8388608	0.000369	90.93
4	8388608	0.000392	85.60
5	8388608	0.000391	85.82

Extra Credit:

In the matrix transpose operation, we start with the naive algorithm implementation which we assign one element to one Cuda thread. Each thread will read from the input array and write onto the output array.

The next step we take is to create small a shared memory tile for each thread. Instead of calculating one element per thread, we calculate 32*32 elements. Then we write the data back to the output array. To make our job easier, the code can only runs on square matrix where its' width is a multiple of 32.

We test both version of algorithm with a 1024*1024 matrix and below is the result:

trial#	GPU transpose (billion elements/second)	
	Naive	Shared Memory
1	2.01649	5.92416

2	1.98219	5.92416
3	1.93108	5.89088
4	1.96731	5.95782
5	1.97845	5.54802