Name: Yixing Guan

Netid: yg1227

Professor Zahran

Parallel Computing

1 May 2018

Lab 3 Report

Note:

1) I tested my program on cuda2-GPU3. I used “cudaSetDevice(3)” to switch from the default GPU to GPU3 on cuda2 server.
2) This is the only GPU that allowed me to run the case X=100,000,000. Other GPUs simply cannot allocate enough GPU memory for me when X=100,000,000. The reason might be that they were shared among my peers when I tested my program.
3) “cudaSetDevice(3)” is not included in the submitted version of my code.

My Setting:

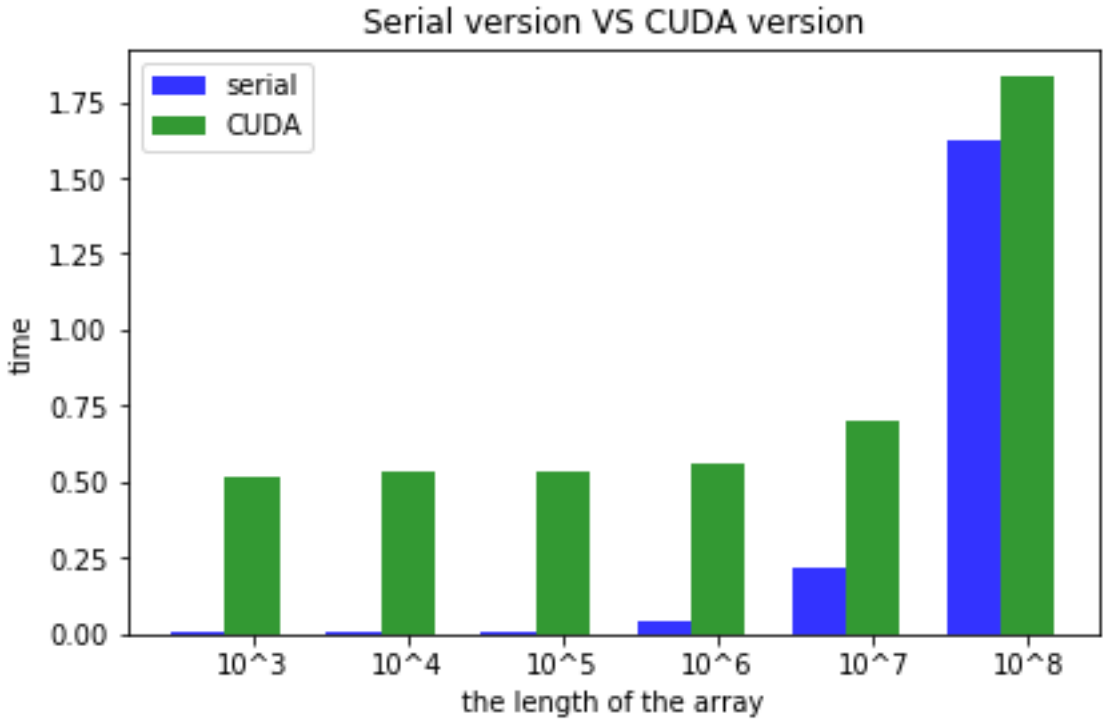
1) The block size is (1024, 1, 1). In other words, each block has 1024 threads. There are two main reasons why I chose this number. First, in my program design, after each thread finds the max in its share of workload, the block will use the reduction tree algorithm to find the max among all threads. However, near the end of this reduction tree algorithm, inevitably only 16, 8, 4, 2, or 1 thread will concurrently execute the next instruction. Since the warp size=32 on cuda2-GPU3, this will surely result in a performance loss. Now that we have to pay the same “price” every time we use this reduction tree algorithm, we’d better assign

more threads to each block in order to cut down the "averaged" cost. Second, the maximum number of threads per block is 1024 on cuda2-GPU3, a strict upper-bound. Hence, it is best to set the number of threads per block to be 1024.

2) The grid size is (64, 1, 1). The total number of blocks is 64. I also experimented many other designs, with the number of blocks ranging from 8 to 512, but the performance of the program measured in run-time did not vary much at all (see Appendix for the detailed experiment data). Therefore, I chose the median 64 among them. In addition, there are 24 multiprocessors on cuda2-GPU3 and the maximum number of threads per multiprocessor is 2048. Since I set the number of threads per block to be 1024, each multiprocessor can take in at most two blocks. Hence, 24 multiprocessors can take in 48 blocks. The power of 2 closest to 48 is 64. Here I did not consider 32 because if there are only 32 blocks, then necessarily some multiprocessors will not be fully utilized. Last but not least, when X=100,000,000, the design of 64 blocks with 1024 threads in each block means that each thread has to find the maximum among at most 1526 elements, which is still an acceptable workload for a thread, given X=100,000,000 is the case with the most workload in this lab.

Command Line used to compile the CUDA version:

```
nvcc maxgpu.cu
```



Explanation for the bar graph above:

1) As can be seen from the graph, the run-time of the CUDA program is at least 0.5s, even when the problem size X is only 10^3. This can be explained by the design of my program. Since the grid size is (64, 1, 1) and block size is (1024, 1, 1) in my program, when the problem size X is small, most of the threads are “idle” in some sense. They don’t have any valid workload. On the other hand, we still need to endure other overheads like launching CUDA, setting up all the blocks and threads, high latency for memory transfer, branch divergence. As a result, the run-time of the CUDA version is larger than 0.5s.

2) As the problem size grows, we see that the gap between the serial version and CUDA version starts to bridge. This indicates that the CUDA version scales better than the serial version. There are two main factors that contribute to this phenomenon. First, as the problem size increases, more threads will be given valid workload and that valid workload also increases, so more work can be done in parallel. Second, the high bandwidth of GPU memory can only be fully utilized when problem size is sufficiently large. As the problem size increases, the CUDA version can exploit more from the high bandwidth while hiding more memory latency. The "price" we have to pay in order to outsource the task to GPU starts to pay off here. I expect that the CUDA version will eventually surpass the serial version if the problem size continues to increase.

Appendix

Table 1

	sequential program running time					
X	case 1	case 2	case 3	case 4	case 5	average
1000	0.006	0.005	0.002	0.005	0.005	0.0046
10000	0.006	0.006	0.006	0.006	0.002	0.0052
100000	0.014	0.007	0.010	0.009	0.006	0.0092
1000000	0.042	0.040	0.038	0.040	0.039	0.0398
10000000	0.216	0.218	0.212	0.218	0.220	0.2168
100000000	1.641	1.624	1.695	1.575	1.595	1.626

Table 2

	GPU program running time (number of blocks=8)					
X	case 1	case 2	case 3	case 4	case 5	average
1000	0.536	0.533	0.518	0.553	0.511	0.5302
10000	0.531	0.534	0.571	0.529	0.532	0.5394
100000	0.546	0.527	0.532	0.542	0.534	0.5362
1000000	0.568	0.561	0.567	0.528	0.558	0.5564
10000000	0.736	0.690	0.719	0.683	0.706	0.7068
100000000	1.851	1.812	1.777	1.812	1.811	1.8126

Table 3

	GPU program running time (number of blocks=16)					
X	case 1	case 2	case 3	case 4	case 5	average
1000	0.524	0.511	0.505	0.509	0.519	0.5136
10000	0.556	0.579	0.534	0.524	0.554	0.5494
100000	0.565	0.547	0.511	0.502	0.524	0.5298
1000000	0.565	0.525	0.536	0.527	0.587	0.548
10000000	0.718	0.722	0.781	0.693	0.694	0.7216
100000000	1.830	1.811	1.816	1.777	1.780	1.8028

Table 4

	GPU program running time (number of blocks=32)					
X	case 1	case 2	case 3	case 4	case 5	average
1000	0.573	0.515	0.549	0.505	0.503	0.529
10000	0.552	0.519	0.502	0.542	0.501	0.5232
100000	0.574	0.532	0.516	0.522	0.506	0.53
1000000	0.560	0.566	0.533	0.552	0.522	0.5466
10000000	0.712	0.694	0.686	0.674	0.747	0.7026
100000000	1.846	1.868	1.833	1.822	1.826	1.839

Table 5

	GPU program running time (number of blocks=64)					
X	case 1	case 2	case 3	case 4	case 5	average
1000	0.506	0.508	0.534	0.501	0.540	0.5178
10000	0.573	0.529	0.536	0.501	0.511	0.53
100000	0.523	0.515	0.529	0.543	0.547	0.5314
1000000	0.590	0.563	0.592	0.533	0.537	0.563
10000000	0.727	0.662	0.752	0.709	0.664	0.7028
100000000	1.835	1.787	1.842	1.827	1.873	1.8328

Table 6

	GPU program running time (number of blocks=128)					
X	case 1	case 2	case 3	case 4	case 5	average
1000	0.535	0.506	0.511	0.513	0.547	0.5224
10000	0.533	0.507	0.522	0.523	0.531	0.5232
100000	0.561	0.554	0.511	0.552	0.515	0.5386
1000000	0.547	0.530	0.558	0.538	0.549	0.5444
10000000	0.744	0.681	0.729	0.661	0.743	0.7116
100000000	1.899	1.803	1.817	1.858	1.841	1.8436

Table 7

	GPU program running time (number of blocks=256)					
X	case 1	case 2	case 3	case 4	case 5	average
1000	0.512	0.578	0.568	0.552	0.576	0.5572
10000	0.534	0.554	0.542	0.529	0.544	0.5406
100000	0.515	0.523	0.530	0.522	0.546	0.5272
1000000	0.558	0.543	0.548	0.556	0.550	0.551
10000000	0.723	0.671	0.693	0.680	0.719	0.6972
100000000	1.805	1.812	1.832	1.804	1.859	1.8224

Table 8

	GPU program running time (number of blocks=512)					
X	case 1	case 2	case 3	case 4	case 5	average
1000	0.530	0.537	0.541	0.561	0.525	0.5388
10000	0.532	0.511	0.569	0.541	0.523	0.5352
100000	0.526	0.518	0.526	0.531	0.523	0.5248
1000000	0.551	0.537	0.530	0.536	0.557	0.5422
10000000	0.675	0.683	0.674	0.664	0.687	0.6766
100000000	1.897	1.841	1.790	1.846	1.853	1.8454