PA1: Matrix Addition

Hannah Munoz

1/28/18

Introduction

In this project, we implemented the sequential and GPU algorithms for matrix addition and the compare the results. Both algorithms are very similar and by comparing the execution times, we can calculate speedup, throughput, and timing of the GPU implementation.

Method

Sequential Algorithm

Both algorithms are very similar to one another. In the sequential algorithm, the user inputs a desired matrix size through command line arguments. Three matrices --- A, B, and C, are created. Matrices A and B filled with numbers. The execution timer starts. Each item of arrays A and B are added together and the result is put in Matrix C as seen in Eq. 1.

Equation 1: $$MatC[i] = MatA[i] + MatB[i]$$

The execution timer stops are the results are outputted.

GPU Algorithm

The GPU algorithm starts with the user inputting the matrix dimension, the desired number of blocks per dimension, and the desired number of thread per dimension. The number of thread and blocks should cover the entire matrix, as in Eq. 2.

Equation 2: $$MatDim^2 = BlockDim^2 * ThreadDim^2$$

The user inputs are checked to see if they are within valid boundaries. If they are not, the program exits with a warning. A grid of BlockDim x BlockDim is created. Each block is create with ThreadDim x ThreadDim threads. Matrices A, B, and C of size MatDim x MatDim are created and filled. Three pointers the size of the matrices are created and GPU memory is allocated to them. The execution timer starts. Matrices A and B are copied to the GPU using the previously allocated pointer.

On the GPU, each core computes it global position using Eq. 3.

Equation 3: $$blockId = blockIdx.x + blockIdx.y * gridDim.x$$
$$GlobalPos = blockId * (blockDim.x * blockDim.y) + (threadIdx.y * blockDim.x) + threadIdx.x$$

The core adds together matrices A and B at this location and puts its result in matrix C. Matrix C is then copied back to the CPU and the execution timing stops.

Results and Conclusion

Sequential

The sequential algorithm's execution times in Table 1 and Fig. 1 increase dramatically as matrix size increases. This is expected with a $O(n^2)$ algorithm, as when the matrix sizes increase, the algorithm will have to spend more time iterating and adding.

Table 1: The sequential execution times in ms

Matrix Size (NxN)	Execution Time (ms)
1000	11.7402
5000	187.663
10000	1326.35
15000	1663.32
20000	6160.97
25000	8503.42
30000	12568
35000	21439.2
40000	26314.6
45000	388853.5

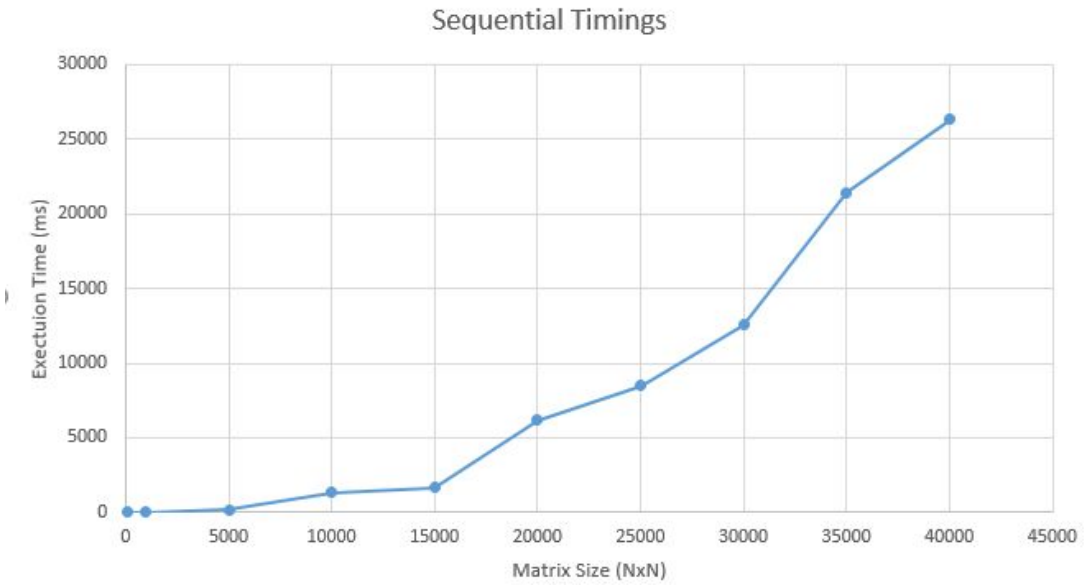

Figure 1: A graph of the sequential execution times found in Table 1.

GPU

The GPU algorithms execution times are in Table 2 and Fig. 2. The graph in Fig. 2 shows timing over different thread counts, as each matrix size can have the same number of threads, but not the same block sizes per Eq. 1.

Table 2: The GPU execution times

Number of Thread / Matrix Size(NxN)	1	5	10	20	25
1000	9.78003	7.86637	7.14454	7.4417	7.1137
5000	169.977	140.61	166.331	150.015	148.627
10000	709.199	634.628	602.51	646.317	605.331
15000	1377.62	1404.1	1402.41	1311.32	2332.5
20000	2986.71	2489.39	2393.84	2536.26	2392.35
25000	9510.41	9031.44	8206.93	8399.77	8352.12

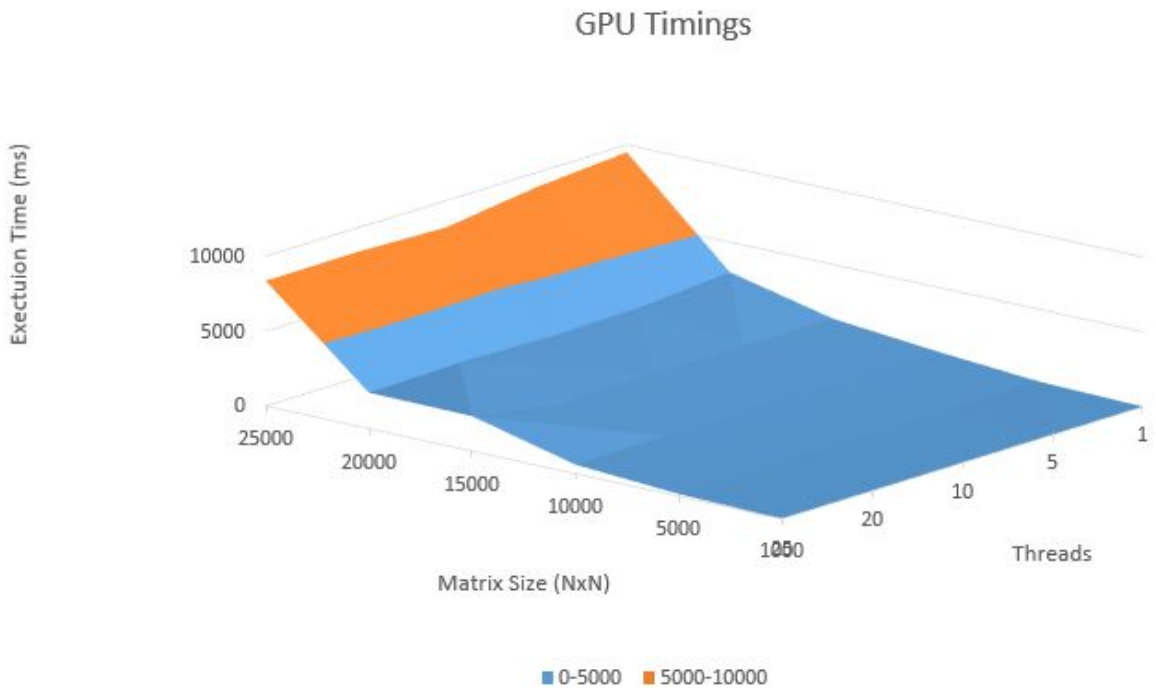

Figure 2: The GPU execution times over varying matrix sizes and threads per block.

This algorithm should have a time complexity of O(1) as no iterating is done. This should make the GPU algorithm run extremely fast, however, there is also the added overhead of copying the matrices to the GPU and back to the CPU. In a 25,000 x 25,000 matix, the cost of sending data to and from the GPU is so expensive that it has a larger execution time than the sequential algorithm.

The equation used to calculate speedup in Table 3 and Fig.3 is given in Eq. 4

Equation 4: $$speedup = \frac{sequential\ time}{GPU\ time}$$

Table 3: The speedup factor of different matrix sizes and thread counts

Thread per block / Matrix Size (NxN)	1	5	10	20	25
1000	1.2004	1.4925	1.6432	1.5776	1.6504
5000	1.1040	1.3346	1.1283	1.2510	1.2626
10000	1.8702	2.0900	2.2014	2.0522	2.1911
15000	1.2074	1.1846	1.1860	1.2684	0.7131
20000	2.0628	2.4749	2.5737	2.4292	2.5753
25000	0.8941	0.9415	1.0361	1.0123	1.0181

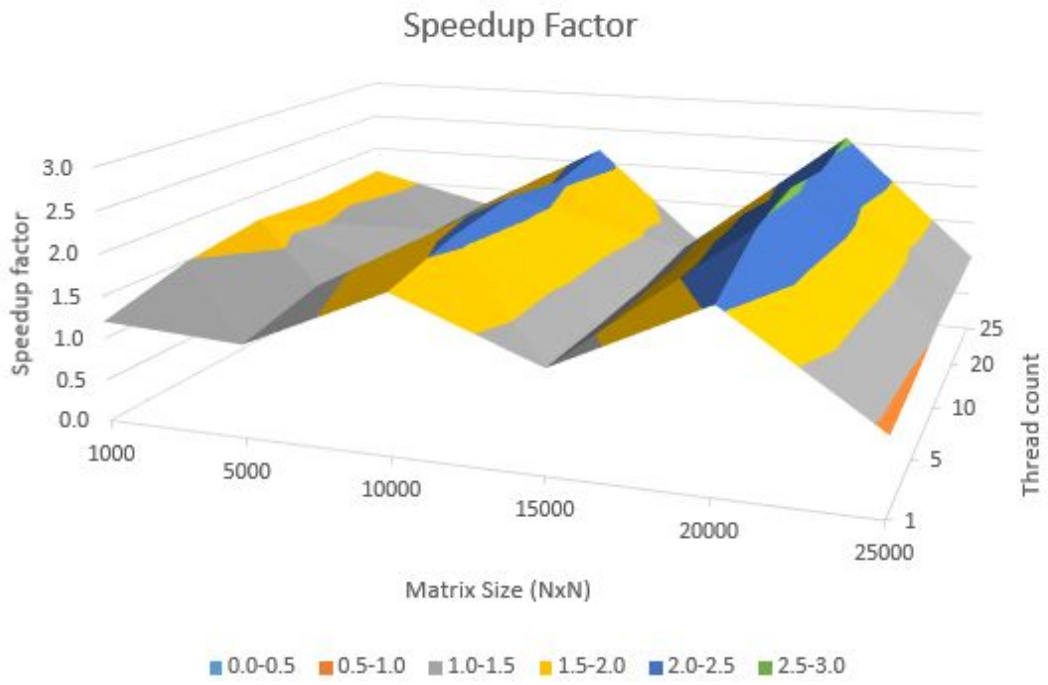

Figure 3: The speedup factor of the GPU algorithm

Max speedup was found to be 2.5753 when the Matrix was 20,000 x 20,000 with 800 blocks and 25 threads per block. Average speedup was found to be 1.5542. Changing the amount of threads per block does not greatly affect the speedup; however, it's interesting to note the changes in speedup which occur as the matrix size expands. This might be due to how CUDA memcpy is storing the matrix array in memory.

The equation used to calculate throughput in Table 4 and Fig.4 is given in Eq. 5

Equation 5:

$$throughput = \frac{Matrix\ Size\ (N\ x\ N)}{GPU\ execution\ time\ (s)}$$

Table 4: The throughput of different matrix sizes and threads per block in bit/sec

Thread per block / Matrix Size (NxN)	1	5	10	20	25
1000	102.2492	127.1234	139.9670	134.3779	140.5738
5000	147.0787	177.7967	150.3027	166.6500	168.2063
10000	141.0041	157.5726	165.9723	154.7228	165.1989
15000	163.3252	160.2450	160.4381	171.5828	96.4630
20000	133.9266	160.6819	167.0955	157.7125	167.1996
25000	65.7175	69.2027	76.1552	74.4068	74.8313

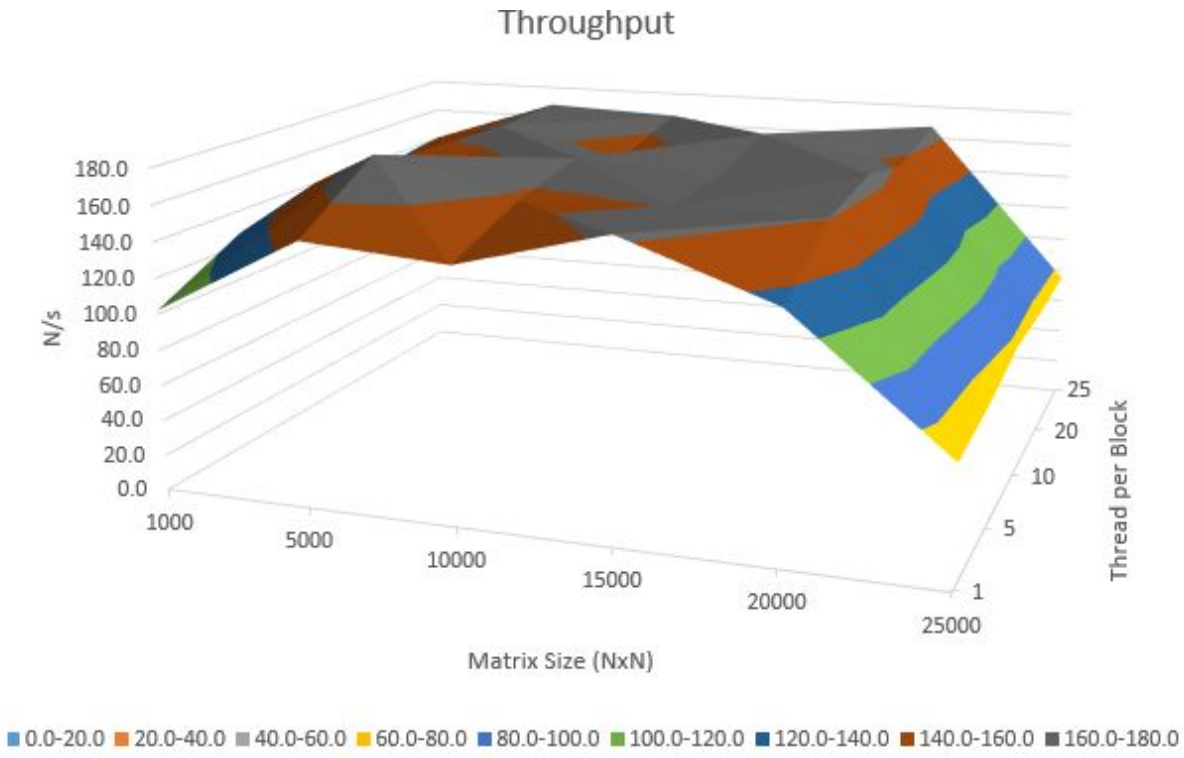

Figure 4: The throughput in Table 3 graphed.

Max throughput was found to be 177.79 bits/sec when the Matrix was 5,000 x 5,000 with 1,000 blocks and 5 threads per block. This is slightly higher than the average throughput, which is 137.92 bits/sec. As soon as the matrix starts to approach the max memory of the GPU throughput drops sharply.