PA2: Matrix Multiplication

Hannah Munoz

2/04/18

Introduction

In this project, we implemented the sequential and GPU algorithms for matrix multiplication and the compare the results. The speedup, throughput, and timing of the parallel implementation is calculated.

Method

Sequential Algorithm

For the sequential, the the matrices A and B are created for multiplication and our results matrix C. We fill the matrices with iterating numbers. The timer starts. There are 3 loops to iterate through. First, we iterate through the rows of matrix A (i), then the columns of matrix B (j), and finally, the individual cell of each matrix (k). Each cell from the matrices is multiplied and added to an ongoing sum for the current column resulting in Eq. 1.

[Equation 1] $$MatC\ [i * size\ +\ j] = \sum_{0}^{size} (MatA[i * size\ +\ k] * MatB[k * size + j])$$

After each entire column iteration, one value of matrix C has been calculated. After each entire row calculation, one row of matrix C has been calculated. Once all iterations have completed, matrix C has been fully calculated and the timer can be stopped.

Parallel Algorithm

The parallel algorithm is set up the same as the matrix addition algorithm was. Once the timer has started, we can begin multiplication. Thanks to the unified memory, no memory needs to be passed. The row and column of the thread is calculated using Equations 2 and 3 respectively.

[Equation 2] $$row = blockId.y * blockDim.y + threadIdx.y$$

[Equation 3] $$col = blockId.x * blockDim.x + threadIdx.x$$

Each thread in the GPU is assigned an element of the matrix. This thread iterates through it's element's row in Matrix A and added to everything in the element's column in Matrix B. The final sum is put into Matrix C. Once all elements have been calculated, the timer 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 $O(n^3)$ algorithm. Because large matrices took so long, execution times were only calculated out to a matrix size of 5000x5000.

Table 1: The sequential execution times in ms

Matrix Size (NxN)	Execution Time (ms)
1000	5255.86
2000	46389.6
2500	113769
3000	176230
3500	339639
4000	484418
4500	789080
5000	1,019,360.00

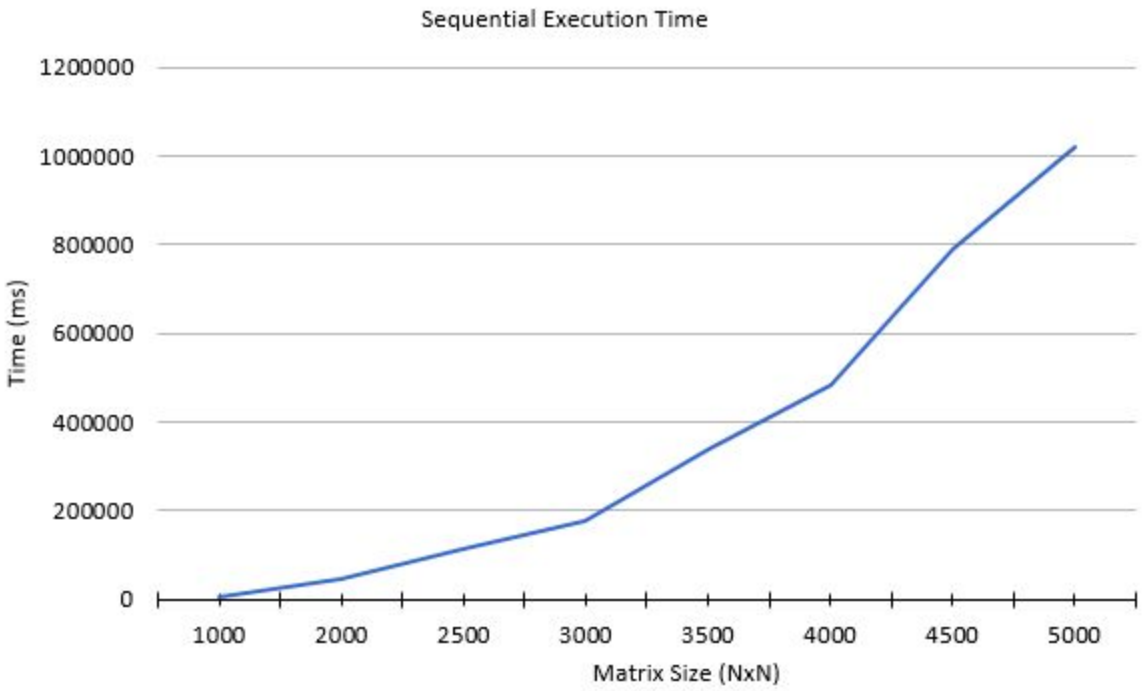

Figure 1: A graph of the sequential execution times found in Table 1.

Parallel

The parallel algorithms execution times are in Table 2 and Fig. 2. The log of the times in Table 2 have been graphed in Fig. 2 to better show the difference in execution times.

Table 2: The parallel execution times

Threads per block / Matrix Size (NxN)	1	5	10	20	25
1000	179.984	23.2861	16.2999	12.2319	13.0635
2000	2536.55	138.011	138.302	70.7697	68.7906
2500	2274.8	262.972	293.022	169.341	141.119
3000	3889.11	737.486	511.615	359.848	289.835
3500	13292.8	457.702	873.313	664.759	507.387
4000	20003.2	1511.1	1272.81	970.756	723.657
4500	25793.8	2231.54	2034.36	1667.36	1227.33
5000	17870.2	3305.27	2756.74	2434.85	1762.17
5500	52872.7	4168.74	3960.07	3353.8	2433.15
6000	160166	6264.54	5293.96	4568.24	3292.36
6500	229730	7827.62	6439.47	5904.45	4321.89
7000	357128	9863.62	8652.65	7542.36	5547.06
7500	203503	12680.4	10966.3	9677.22	7020.6
8000	495732	21432.3	13347.2	11675	8640.04
8500	753540	23430.6	16256.4	14287	10559.8
9000	540252	28748.4	19442	17350.5	12804
9500	465133	38043.9	22968.2	20584.1	15217.4
10000	574463	29509.6	27485.3	24555.5	18077.3

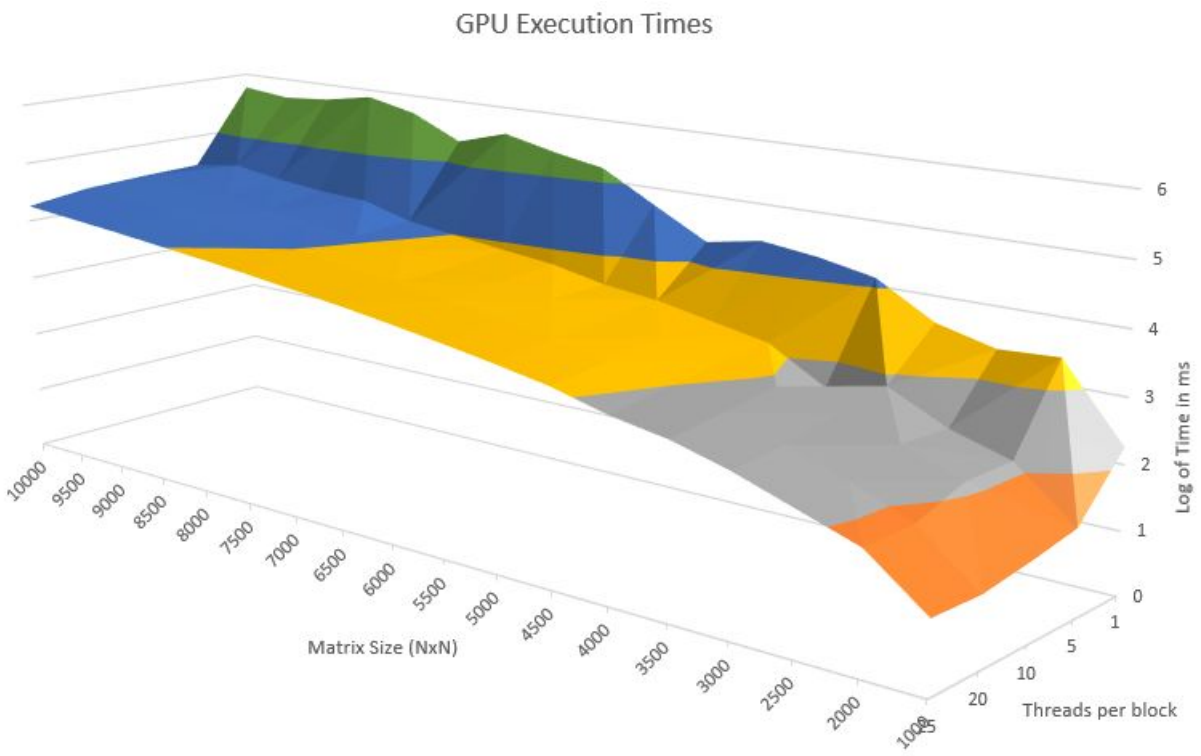

Figure 2: The GPU execution times over varying matrix sizes and threads per block.

The parallel algorithm is still $O(n^3)$, meaning that large matrix sizes will still take a significantly long time. Another interesting thing to note is how much longer a single thread execution takes than any other amount per threads. 1 thread per block is still significantly faster, as we will see. Most often the more threads available, the faster the execution occurred.

The equation used to calculate speedup in Table 3 and Fig.3 is given in Eq. 4. The speedup was only calculated out to 5000, as that's as high as could be executed in a timely matter with the sequential timings.

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	29.202	225.708	322.447	429.685	402.332
2000	18.288	336.130	335.422	655.501	674.360
2500	50.013	432.628	388.261	671.834	806.192

3000	45.314	238.960	344.458	489.735	608.036
3500	25.551	742.053	388.909	510.920	669.388
4000	24.217	320.573	380.589	499.011	669.403
4500	30.592	353.603	387.876	473.251	642.924
5000	57.042	308.404	369.770	418.654	578.469

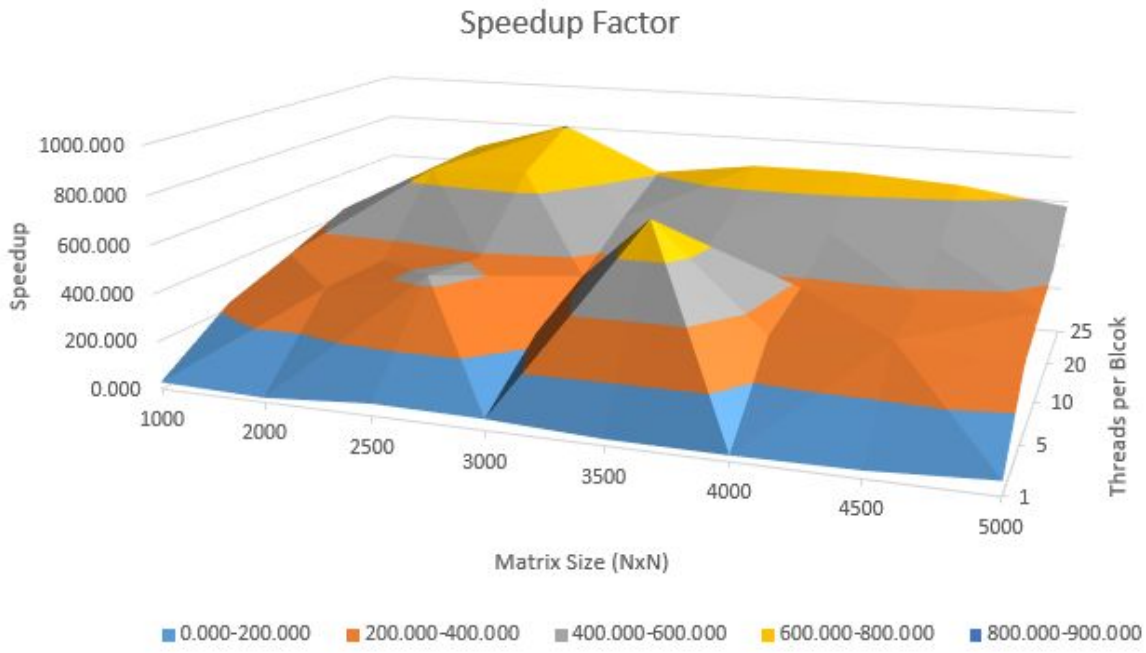

Figure 3: The speedup factor of the parallel algorithm

Max speedup was found to be 806.19 when the Matrix was 2,500 x 2,500 with 10 blocks and 25 threads per block. Average speedup was found to be 389.333.

The speedup for matrix multiplication was immense. Because the matrix can be divided into sections that are run in parallel, it's as though you are doing a much smaller sequential matrix. The peaks in Fig. 3 show where paging became much more efficient and the cache was used correctly.

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	5556.049	42944.074	61350.070	81753.448	76549.164
2000	1576.945	28983.197	28922.214	56521.364	58147.479
2500	2747.494	23766.789	21329.456	36907.778	44288.863
3000	2314.154	12203.621	17591.353	25010.560	31052.150
3500	921.552	26764.139	14027.044	18427.731	24143.307
4000	799.872	10588.313	12570.611	16482.000	22109.922
4500	785.072	9074.451	9953.990	12144.948	16499.230
5000	1398.977	7563.679	9068.683	10267.573	14187.053

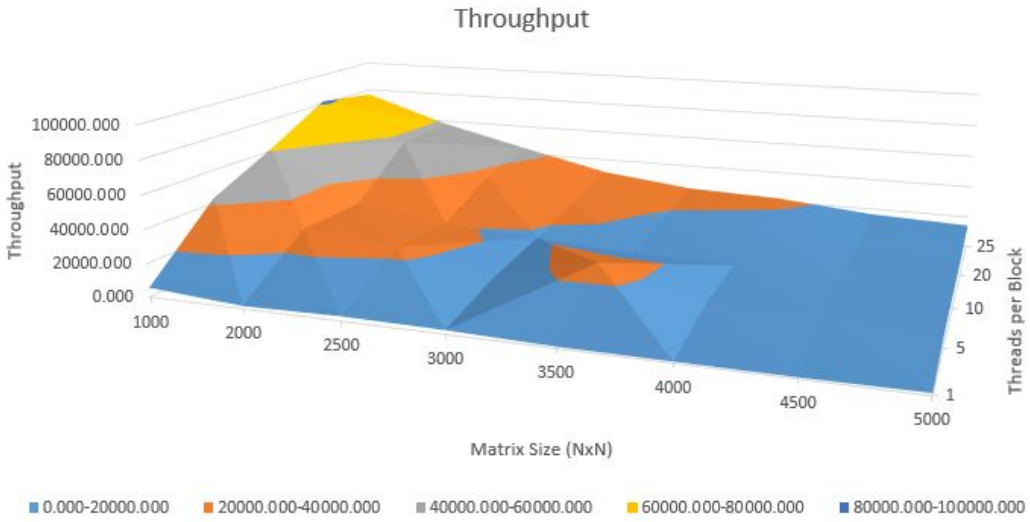

Figure 4: The throughput in Table 3 graphed.

Max throughput was found to be 81753.448 bits/sec when the Matrix was 10,000 x 10,000 with 500 blocks and 20 threads per block. Average throughput was 22612.043 bits/sec.

We see once again that throughput is most efficient where more threads are doing work. As the matrix size becomes very large, there is a decrease in thoughput. The GPU algorithm does better still with smaller matrices.