COMPSCI 532 S22 - Project 3: GPGPU Programming

Christopher Shi
University of Massachusetts Amherst
140 Governors Dr, Amherst, MA 01002
cshi@umass.edu

1 Section 1: Problem Statement

In this project, we will evaluate and compare the performance of CPU and GPU-accelerated applications on three different tasks:

1. Vector Addition
2. Matrix Multiplication
3. Tiled Matrix Multiplication with Shared Memory

2 Section 2: Implementation

To begin, all CPU and GPU-accelerated implementations were created in the CUDA environment utilizing C. Testing was conducted utilizing one Tesla p100 GPU. As such, 1024 threads per block were able to be utilized. In each test, the input vectors/matrices were initialized in the following manner:

1. Input A: $sin(index)$ where index is the respective location in the vector/matrix
2. Input B: $cos(index)$ where index is the respective location in the vector/matrix

For each of the functions, for both CPU and GPU implementations, the input vectors/matrices A and B as well as the output vector/matrix C were initialized/allocated within the function rather than passed into the function as a parameter. The input arrays also have their values initialized within the functions. Once the input and output vectors/matrices were allocated and initialized with values, the timer was started. For the CPU implementations, the timer was ended immediately following the execution of the desired task. (Copying the output array and freeing memory was conducted after the timer had stopped recording) For the GPU implementations, the timer was started in the same manner as the CPU implementations. The timer was ended following the execution of cudaMemcpy(). (Freeing the GPU memory, copying the output array, and freeing the CPU memory was conducted after the timer had stopped recording) Additionally, to error check, the output vectors/arrays from the CPU and GPU implementations are compared at each index. The total error is the sum of any discrepancies between the CPU output and the GPU output across all indices. For all three tasks, a discrepancy of 0.0 was attained.

2.1 Implementation of Vector Addition

For the CPU implementation of vector addition, a simple for loop was utilized that iterated through each index of A and B and summed them into the corresponding index of C.

For the GPU implementation of vector addition, the chosen number of threads per block was 1024, and the number of blocks in the grid was chosen to be equal to the following: $ceil(N/threadsperblock)$ where N = size of input vectors A and B. The kernel function implemented first performs a check to make sure the chosen index is less than N (where N = size of input vectors A and B), before executing the following: $c[index] = a[index] + b[index]$; where c is the output vector and a and b are the input vectors.

The chosen values of N were selected as follows: $N \in \{1.5^x\}$ where $\{x = Z|1 \leq x \leq 50\}$.

2.2 Implementation of Multiplication of two NxN Matrices

For the matrix implementations, rather than utilizing a 2D array, a 1D array was utilized.

For the CPU implementation of matrix multiplication, three nested for loops are utilized in order to calculate the matrix multiplication operation. The first two for loops iterate through the rows and columns, respectively, while the last for loop sums up the values of the multiplication between the values of the row and column chosen from the first two for loops.

For the GPU implementation of matrix multiplication, the chosen number of threads per block was equal to $(N, N, 1)$ if $N * N <= 1024$ and the number of blocks in the grid was equal to $(1, 1, 1)$. For values of N s.t. $N * N > 1024$, the number of threads was equal to $(32, 32, 1)$ and the number of blocks per grid was equal to $(ceil((N + 31)/threadsperblock.x), ceil((N + 31)/threadsperblock.y), 1)$ where N is equal to the chosen size of the matrix NxN. The kernel function implemented first performs a check on the row and column values for the chosen index, s.t. row ¡ N and column ¡ N. Following that, a simple for loop is utilized that calculates the sum of the multiplied values of the chosen row from A and the chosen column from B. The final sum is put into the designated index of the output matrix C, given that each thread computes one element/index of the block sub-matrix.

The chosen values of N were selected as follows: $N \in \{1.5^x\}$ where $\{x = Z|1 \leq x \leq 20\}$.

2.3 Implementation of Multiplication of two NxN Matrices Utilizing Tiled Matrix Multiplication with Shared Memory

For the matrix implementations, rather than utilizing a 2D array, a 1D array was utilized.

For the CPU implementation of matrix multiplication, three nested for loops are utilized in order to calculate the matrix multiplication operation. The first two for loops iterate through the rows and columns, respectively, while the last for loop sums up the values of the multiplication between the values of the row and column chosen from the first two for loops.

For the GPU implementation of matrix multipication of two NxN matrices utilizing tiled matrix multiplication with shared memory, the number of threads per block was assigned as the following: $(TILE\_DIM, TILE\_DIM, 1)$ where $TILE\_DIM$ is equal to the size of one side of the square tile that is being utilized. The number of blocks in the grid is assigned as follows: $((N + TILE\_DIM - 1)/TILE\_DIM, (N + TILE\_DIM - 1)/TILE\_DIM, 1)$ where N is equal to the chosen size of the matrix NxN and $TILE\_DIM$ is equal to the size of one side of the square tile that is being utilized. The kernel function implemented allows for each thread to be responsible for loading the input elements into the shared memory. Essentially mini-matrix multiplication is being performed with shared memory, temporary results are stored, and then continue summing the temporary results of the next mini-matrix multiplication. When each individual mini-matrix multiplication finishes, each thread would load their corresponding result to the output C element that they are mapped to.

The chosen values of N were selected as follows: $N \in \{1.5^x\}$ where $\{x = Z|1 \leq x \leq 20\}$. $TILE\_DIM$ had the following values: 2, 4, 8, 16, 32.

3 Section 3: Experimental results

When comparing GPU and CPU implementation of vector addition, matrix multiplication of two NxN matrices, and matrix multiplication of two NxN matrices utilizing tiled matrix multiplication with shared memory, the method of evaluation is how long each process takes to complete their task with an input of N/NxN respectively. This will be measured in seconds, and compared.

From Figure 1. and Table 1., we can see that the GPU implementation of vector addition was able to perform a faster computation relative to the CPU implementation once a vector size of 647,160 was reached.

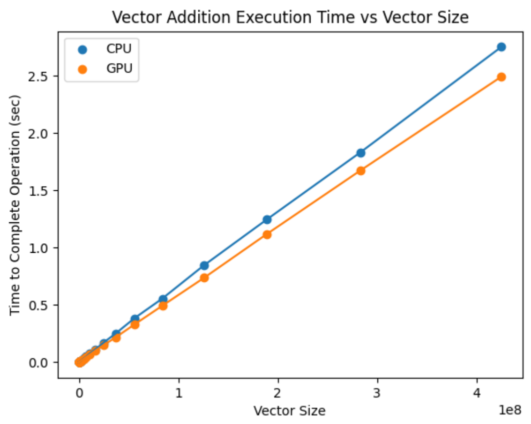

Figure 1: Visualization of Time Taken for CPU and GPU Implementation of Vector Addition.

N	CPU (sec)	GPU (sec)
1	0.000001	0.00018
2	0.000001	0.000166
2	0.000001	0.000229
3	0.000001	0.000174
5	0.000001	0.000178
8	0.000001	0.000244
11	0.000001	0.000184
17	0.000001	0.000134
26	0.000001	0.000142
38	0.000001	0.000136
58	0.000002	0.000137
86	0.000001	0.000136
130	0.000003	0.000162
195	0.000002	0.000141
292	0.000002	0.000155
438	0.000002	0.000213
657	0.000006	0.000144
985	0.000004	0.000145
1478	0.000009	0.000159
2217	0.000009	0.000162
3325	0.00002	0.000185
4988	0.000024	0.000211
7482	0.000041	0.000279
11223	0.000057	0.000222
16834	0.000111	0.000286
25251	0.000141	0.000323
37877	0.000215	0.000391
56815	0.000292	0.000508
85223	0.000437	0.000724
127834	0.000604	0.001054
191751	0.000798	0.001531
287627	0.001248	0.00185
431440	0.002027	0.00249
647160	0.003349	0.002881
970740	0.006582	0.004346
1456110	0.010932	0.006842
2184164	0.015089	0.008826
3276247	0.023093	0.014562
4914370	0.033927	0.020281
7371555	0.050587	0.030343
11057332	0.078345	0.046401
16585998	0.114499	0.068477
24878998	0.174229	0.097192
37318497	0.257071	0.148999
55977745	0.386087	0.226763
83966617	0.602579	0.33943
125949926	0.882715	0.517558
188924889	1.331125	0.774053
283387333	2.055184	1.166795
425081000	3.07154	1.762472

Table 1: Results of Time Taken for Vector Addition.

This is most likely because at this vector size, the threaded implementation of the GPU implementation was able to be cost effective relative to the extraneous factors associated with utilizing GPU code. These factors are prevalent in smaller datasets and include:

1. Copying the data buffer onto the device memory
2. Performing the specific computations
3. Copying the device buffer back to the host memory

Copying the data to device and back to the host are very costly operations. It is only when we have a much larger input size when we are able to have a much larger increase factor of performance.

From Figure 2. and Table 2., we can see that the GPU implementation of matrix multiplication was able to perform a faster computation relative to the CPU implementation once a matrix size of 58x58 was reached.

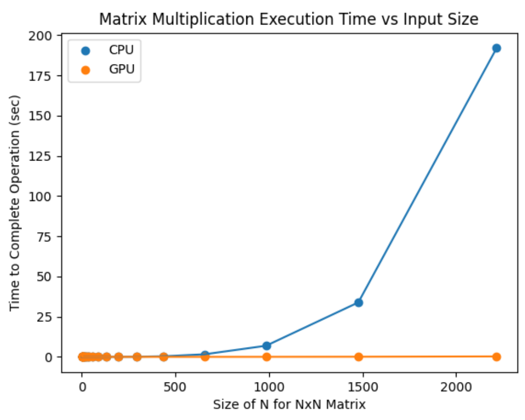

Figure 2: Visualization of Time Taken for CPU and GPU Implementation of Matrix Multiplication of two NxN Matrices.

N	CPU (sec)	GPU (sec)
1	0.000001	0.00018
2	0.000001	0.000143
2	0.000001	0.000123
3	0.000001	0.000118
5	0.000002	0.000126
8	0.000003	0.000132
11	0.000005	0.000158
17	0.000018	0.000136
26	0.000059	0.000145
38	0.000199	0.0002
58	0.000658	0.000236
86	0.002125	0.000321
130	0.007306	0.000428
195	0.025812	0.000856
292	0.090145	0.0017
438	0.380248	0.003741
657	1.615092	0.009771
985	6.923595	0.028201
1478	33.869562	0.087638
2217	192.016584	0.276046

Table 2: Results of Time Taken for Matrix Multiplication of two NxN Matrices.

From Figure 3. and Table 3., we can see that the GPU implementation of matrix multiplication of two NxN matrices utilizing tiled matrix multiplication with shared memory and a tile size of 2 was able to perform a faster computation relative to the CPU implementation once a matrix size of 58x58 was reached.

From Figure 3. and Table 4., we can see that the GPU implementation of matrix multiplication of two NxN matrices utilizing tiled matrix multiplication with shared memory and a tile size of 4 was able to perform a faster computation relative to the CPU implementation once a matrix size of 38x38 was reached.

From Figure 3. and Table 5., we can see that the GPU implementation of matrix multiplication of two NxN matrices utilizing tiled matrix multiplication with shared memory and a tile size of 8 was able to perform a faster computation relative to the CPU implementation once a matrix size of 38x38 was reached.

From Figure 3. and Table 6., we can see that the GPU implementation of matrix multiplication of two NxN matrices utilizing tiled matrix multiplication with shared memory and a tile size of 16 was able to perform a faster computation relative to the CPU implementation once a matrix size of 38x38 was reached.

From Figure 3. and Table 7., we can see that the GPU implementation of matrix multiplication of two NxN matrices utilizing tiled matrix multiplication with shared memory and a tile size of 32 was able to perform a faster computation relative to the CPU implementation once a matrix size of 38x38 was reached.

When comparing Tables 3,4,5,6 and 7, it is interesting to see how the GPU runtime is able to decrease as the tile size increases. Tile sizes of 8, 16 and 32 are all able to attain significant speed improvements as compared to tile sizes of 2 and 4. Once the tile size has been set to one of 8, 16, or 32, the improvements between them seem to be marginal at best.

Additionally, it is interesting to see that tile sizes of 2 and 4 are slower than the non-tiled matrix multiplication implementation. Tile sizes of 8, 16, and 32 were able to marginally improve on the performance compared to the non-tiled matrix multiplication implementation. This may be because it is at these points where the decreases in global memory accesses are compared to non-tiled matrix multiplication are able to overcome the overhead of creating the shared memory arrays and syncing the threads.

N	CPU (sec)	GPU (sec)
2	0.000001	0.00014
2	0.000001	0.000144
3	0	0.000122
5	0.000001	0.000127
8	0.000002	0.000133
11	0.000006	0.000125
17	0.000018	0.000129
26	0.000061	0.000138
38	0.000242	0.00016
58	0.000724	0.000212
86	0.002292	0.00036
130	0.007647	0.000716
195	0.026303	0.001887
292	0.090822	0.005446
438	0.383086	0.017371
657	1.575408	0.05608
985	6.406179	0.178479
1478	33.213427	0.445925
2217	187.250811	1.188255

Table 3: Results of Time Taken for Matrix Multiplication of two NxN Matrices Utilizing Tiled Matrix Multiplication with Shared Memory with Tile Size=2.

N	CPU (sec)	GPU (sec)
2	0.000002	0.00012
2	0.000001	0.000123
3	0	0.000126
5	0.000001	0.000123
8	0.000004	0.000129
11	0.000005	0.000124
17	0.000018	0.000126
26	0.000061	0.000151
38	0.000202	0.00014
58	0.000759	0.000188
86	0.002416	0.000283
130	0.00803	0.000428
195	0.027904	0.000832
292	0.089555	0.002039
438	0.390057	0.005632
657	1.604521	0.016282
985	6.768446	0.05005
1478	32.919626	0.160733
2217	188.543722	0.379173

Table 4: Results of Time Taken for Matrix Multiplication of two NxN Matrices Utilizing Tiled Matrix Multiplication with Shared Memory with Tile Size=4.

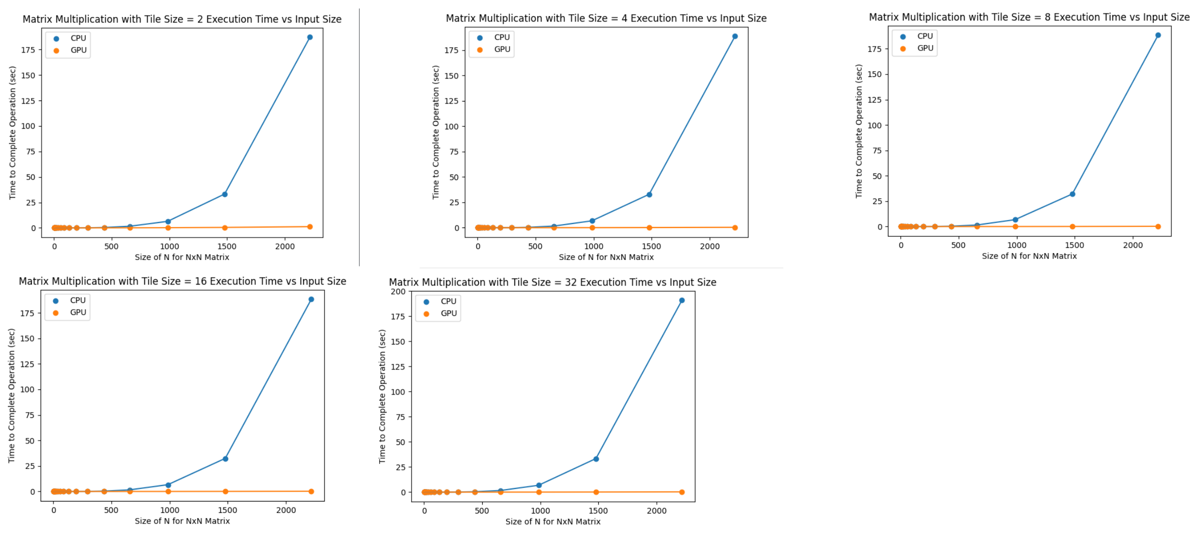

Figure 3: Visualization of Time Taken for for CPU and GPU Implementation of Matrix Multiplication of two NxN Matrices Utilizing Tiled Matrix Multiplication with Shared Memory. Tile sizes of size 2,4,8,16,and 32 are shown.

N	CPU (sec)	GPU (sec)
2	0.000001	0.00013
2	0.000001	0.000132
3	0.000001	0.00013
5	0.000001	0.000146
8	0.000003	0.000115
11	0.000006	0.000121
17	0.000018	0.000124
26	0.000061	0.000167
38	0.000203	0.000147
58	0.00066	0.000189
86	0.002255	0.000256
130	0.007706	0.000365
195	0.026302	0.000665
292	0.086361	0.00151
438	0.379215	0.003603
657	1.534576	0.009719
985	6.780253	0.027991
1478	32.06246	0.085594
2217	188.428903	0.248206

Table 5: Results of Time Taken for Matrix Multiplication of two NxN Matrices Utilizing Tiled Matrix Multiplication with Shared Memory with Tile Size=8.

N	CPU (sec)	GPU (sec)
2	0.000001	0.00013
2	0.000001	0.000139
3	0.000001	0.000157
5	0.000001	0.000137
8	0.000003	0.000137
11	0.000006	0.000121
17	0.000018	0.000132
26	0.000083	0.000225
38	0.000217	0.000171
58	0.000723	0.000181
86	0.002741	0.000326
130	0.008305	0.000439
195	0.026596	0.000733
292	0.094914	0.001547
438	0.373469	0.003736
657	1.594878	0.009866
985	6.662201	0.027764
1478	32.434478	0.085717
2217	188.272476	0.226037

Table 6: Results of Time Taken for Matrix Multiplication of two NxN Matrices Utilizing Tiled Matrix Multiplication with Shared Memory with Tile Size=16.

N	CPU (sec)	GPU (sec)
2	0.000001	0.00016
2	0.000001	0.000162
3	0.000001	0.000151
5	0.000001	0.00015
8	0.000002	0.000151
11	0.000005	0.000148
17	0.000017	0.000155
26	0.000066	0.000175
38	0.000216	0.000199
58	0.000676	0.000246
86	0.00236	0.000331
130	0.007778	0.000447
195	0.026618	0.000875
292	0.090462	0.001763
438	0.386316	0.003865
657	1.623397	0.010823
985	6.813613	0.030027
1478	33.398975	0.093886
2217	190.832926	0.239224

Table 7: Results of Time Taken for Matrix Multiplication of two NxN Matrices Utilizing Tiled Matrix Multiplication with Shared Memory with Tile Size=32.

4 Section 4: Summary and Takeaway

In this experiment we show how utilzing GPU implementations in the CUDA environment are able to surpass the performance of CPU implementations. This occurs when the input size is large enough to overcome the overhead associated with a GPU implementation. Additionally, we saw how utilizing tiled matrix multiplication with shared memory was able to improve upon the performance of non-tiled matrix multiplication though the use of shared memory which entails fewer global memory accesses which in turn makes the implementation more efficient.

5 Section 5: Team Contribution

Christopher Shi finished the whole project by himself.

6 Section 6: Artifact evaluation

The terminal output of 'vectorAddition.cu' was copied to a text file named 'vectorAdd.txt'. The terminal output of 'matrixMult.cu' was copied to a text file named 'matrixMult.txt'. The terminal output of 'tileMult.cu' with tile size = 2 was copied to a text file named 'tileMult2.txt'. The terminal output of 'tileMult.cu' with tile size = 4 was copied to a text file named 'tileMult4.txt'. The terminal output of 'tileMult.cu' with tile size = 8 was copied to a text file named 'tileMult8.txt'. The terminal output of 'tileMult.cu' with tile size = 16 was copied to a text file named 'tileMult16.txt'. The terminal output of 'tileMult.cu' with tile size = 32 was copied to a text file named 'tileMult32.txt'. Executing 'readData.py' compiles the text files and displays the generated input size vs runtime plots.