PROGRAMMING ASSIGNMENT REPORT

PA1: Matrix Addition

Steven Fisher

CS 791

January 30, 2018

1 Introduction

1.1 Preliminary

Before the implementation of the two matrix addition methods, we will needed to obtain and allocate dynamically, the size of the matrices. In addition, we needed loops in order to allow the user to specify the number of blocks for the GPU computations and the the of threads per block.

1.2 GPU

In the GPU portion of this assignment we are requried to implement a method that adds two $N \times N$ matrices. We will be implementing this utilizing CUDA from NVIDIA, in conjunction with C++. During the calculation, we need to determine the time it takes to perform the copying and addition on the GPU. We will also need to calculate the throughput of the GPU. In addition, we need to perform the addition using striding and without using striding. Once we have our calculations for runtime, we will then need to determine the speedup as compared to running the sequential version.

1.3 Sequential

In the sequential part, we will need to implement the method in order to add two $N \times N$ matrices. We will be implementing this using C++ and we will be comparing the results with the calculation obtained with the GPU.

2 Methods

2.1 GPU

In implementing the matrix addition with the GPU, we needed to understand how the parallel computations occur. Here we needed to implement the addition for the matrices without using striding and with using striding. Since, our matrices are dynamically allocated, I found it easier to just define the arrays as 1-d, instead of two. This way we could use a similar method of calculation for both the sequential and GPU. This forced me to make sure that upon the addition, we needed to make sure that we were adding the correct elements. The most trouble that I had during this assignment, was in trying to determine when I was and wasn't using striding. When dealing with striding, I am still not sure if I have implemented it correctly. My implementation did perform what I would expect, as far as, having a better speedup over the sequential. However, we were able to implement the algorithm correctly to add the two matrices and then compare them with the calculation performed on the sequential. One thing that I did notice, is that for smaller matrices, there gpu computing is actually slower than the cpu. This could be due to the overhead of transferring the data from host to device and back from device to host.

2.2 Sequential

The sequential portion is contained within the same as the GPU. This was written using C++. The matrix addition code for the sequential is defined in the matcpuadd.cu file. The matrices

are dynamically allocated based on the size inputed by the user. Was the matrices are filled with arbitrary elements, they are then added together using the formula:

$$C_{ij} = A_{ij} + B_{ij}$$

However, in the code since we are using 1-D arrays, the addition is done by

c[rows * N + columns]= a[rows * N + columns] + b[rows * N + columns]

in order to add the correct elements together

3 Data

3.1 GPU

Our program allows us to specify any matrix from 10x10 to 1000x1000.

Table 1: Calculations for the number of calculations per second in the GPU.

Matrix Size(NxN)	Throughput
10x10	1507.2
50x50	112560
100x100	950400
200x200	3.56E+06
300x300	8.35E+06
400x400	6.68E+07
500x500	4.13E+07
600x600	7.97E+07
700x700	1.46E+08
800x800	2.29E+08
900x900	4.59E+08
1000x1000	5.35E+08

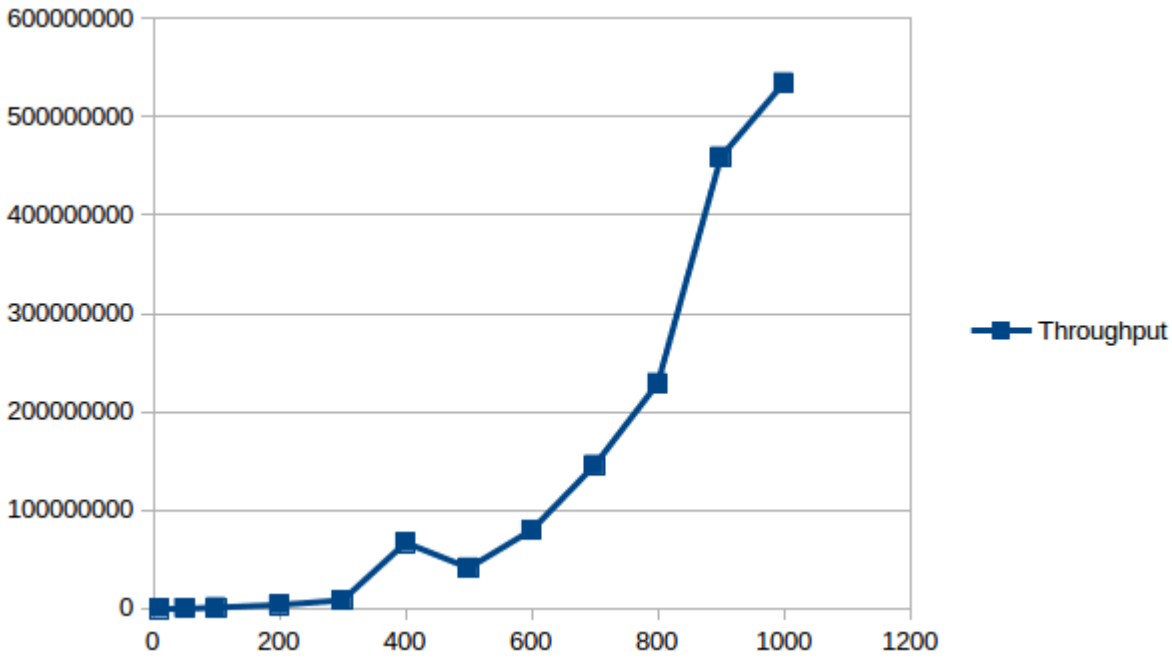

Figure 1: Graph of the throughput for the gpu calculations for each of the recorded sizes.

Table 2: Values from PA1 for GPU Calculations both with and without striding. In most cases striding outperfomed the in calculation over those ran without striding.

Matrix Size(NxN)	Gpu time w/o stride	Gpu time with stride
10x10	0.044768	0.02096
50x50	0.256992	0.317664
100x100	0.265824	0.19744
200x200	0.338336	0.18784
300x300	0.522496	0.343104
400x400	1.23654	0.7088
500x500	1.0513	0.78424
600x600	1.60829	0.945344
700x700	2.20528	1.35024
800x800	2.52794	1.70957
900x900	3.08115	1.68717
1000x1000	3.1065	2.48784

Once, we had implemented the functions to deal with the matrix addition both with and without striding, we then proceeded to collect data on various different aspect. Note, the run includes the overhead for copying the data from the host to the device and for copying back from the device to the host. I used a separate timer to calculate the copying overhead from the host to device and them combined that with the runtime for both implementations on the GPU. As, seen in the preceding table, it would appear that the striding implementation outperforms the implementation without striding. The following two graphs represent the data from the preceding table.

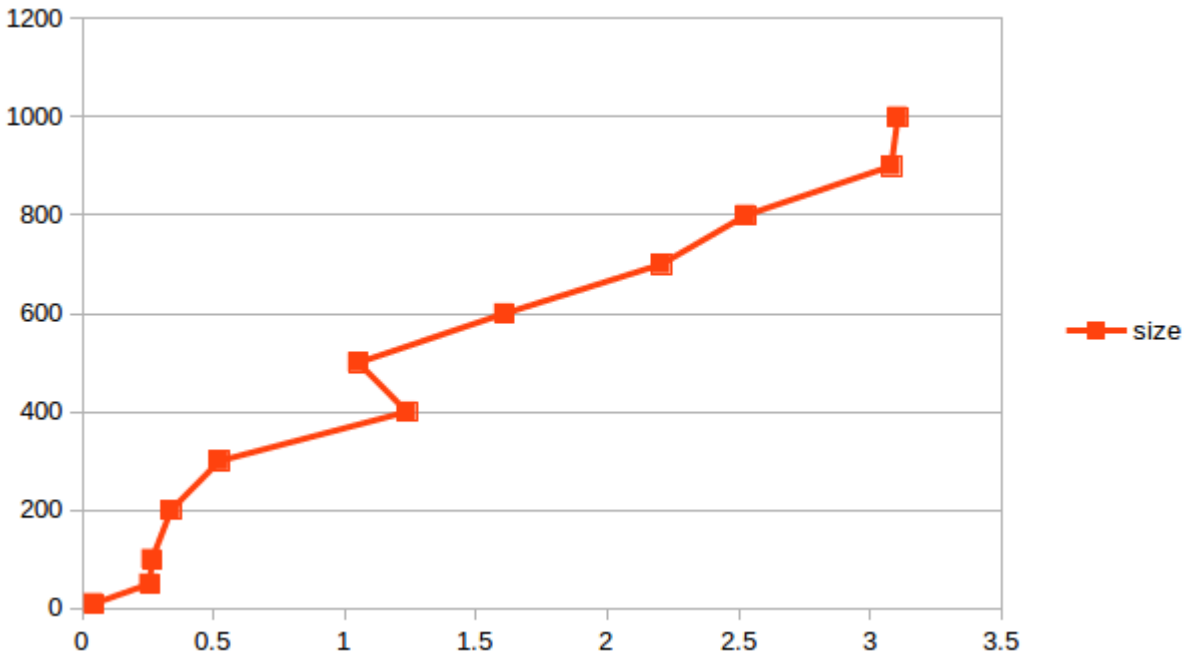

Figure 2: Graph of the runtimes for the GPU calculation without striding for all recorded sizes.

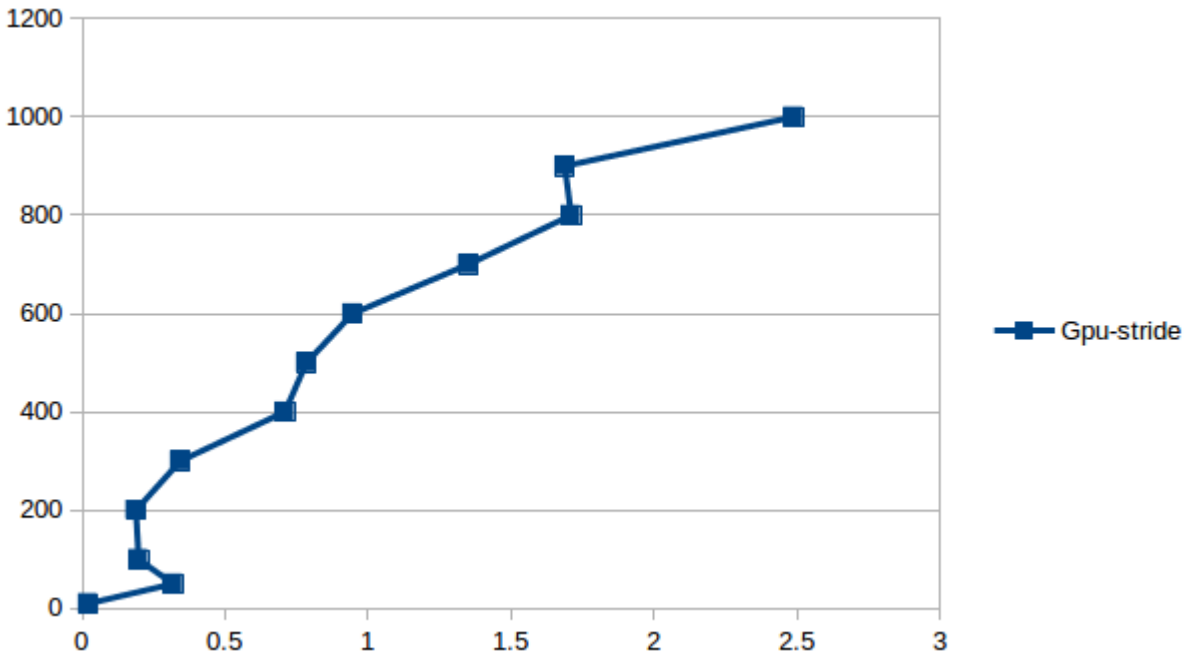

Figure 3: Graph of the runtimes for the GPU calculation with striding for all recorded sizes.

Table 3: Values from PA1 for GPU Speedup both with and without striding. In most cases there exhibited a noticeable difference. These are subjected to choosing the appropriate block size and number of threads.

Matrix Size(NxN)	Gpu Speedup w/o stride	Gpu Speedup with stride
10x10	0.0364546	0.0778626
50x50	0.0100859	0.00815956
100x100	0.445889	0.600324
200x200	1.51868	2.73543
300x300	2.08164	3.17002
400x400	1.52572	2.66172
500x500	1.40164	1.87134
600x600	1.0152	1.72713
700x700	1.11709	1.82448
800x800	1.06351	1.57261
900x900	0.902914	1.64893
1000x1000	1.00385	1.25348

Here we used the formula:

elapsed_cpu / elapsed_gpu

to calculate the speedup of the GPU calculations over the CPU calculations. In most instances both GPU implementations outperforms the CPU. Note in the smaller matrices, I believe that it could be the copying overhead, that is impacting the speed for the GPU and causing it to appear slower than the CPU. Also, note that these calculations and based on how well I was able to ascertain what the best number of blocks to use and the best number of threads per block. Based on my experiments, these were the best results that I was able to obtain. The following two graphs represent the data from the preceding table.

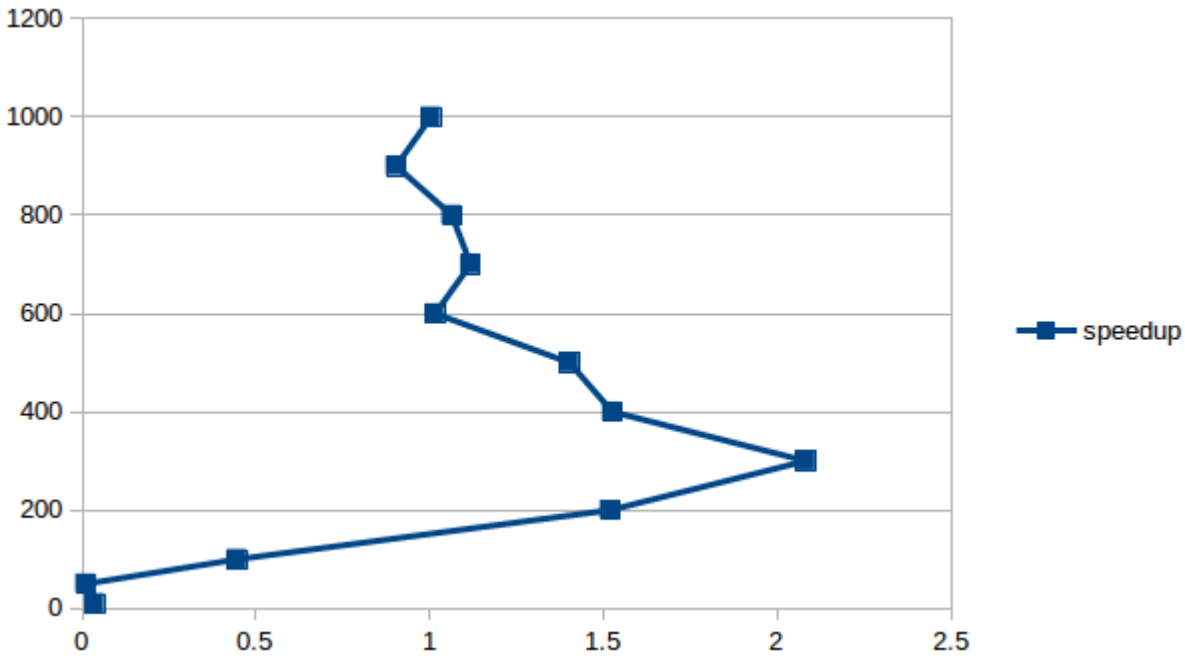

Figure 4: Graph of the speedup for the GPU without striding for all recorded sizes.

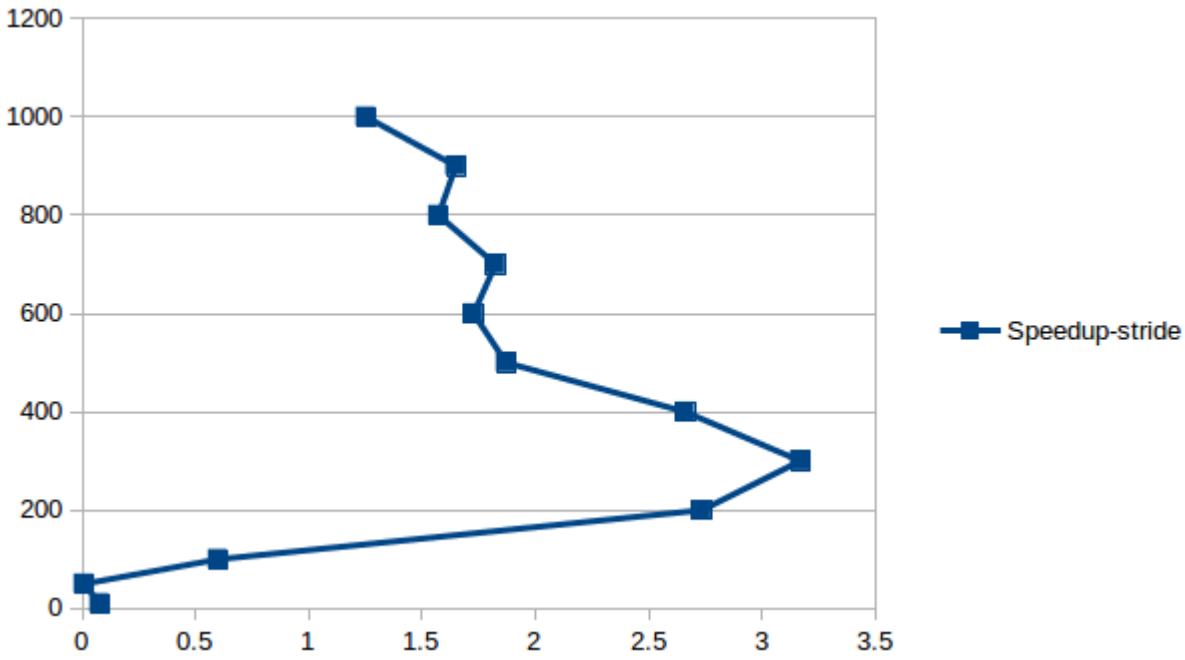

Figure 5: Graph of the speedup for the GPU without striding for all recorded sizes.

3.2 Sequential

As, expected the CPU underperformed when comparing to the GPU parallel calculations. The following table is a collection of different sized matrices for the CPU run-times. As noted earlier; even though, these are fairly quick, they still was a noticeable difference between the run-times observed for the CPU and the run-times observed for the GPU.

Table 4: Collected runtimes for the sequential calculations.

Matrix Size(NxN)	CPU runtime
10x10	0.001632
50x50	0.002592
100x100	0.118528
200x200	0.513824
300x300	1.08765
400x400	1.88662
500x500	1.47354
600x600	1.63274
700x700	2.46349
800x800	2.68848
900x900	2.78202
1000x1000	3.11846

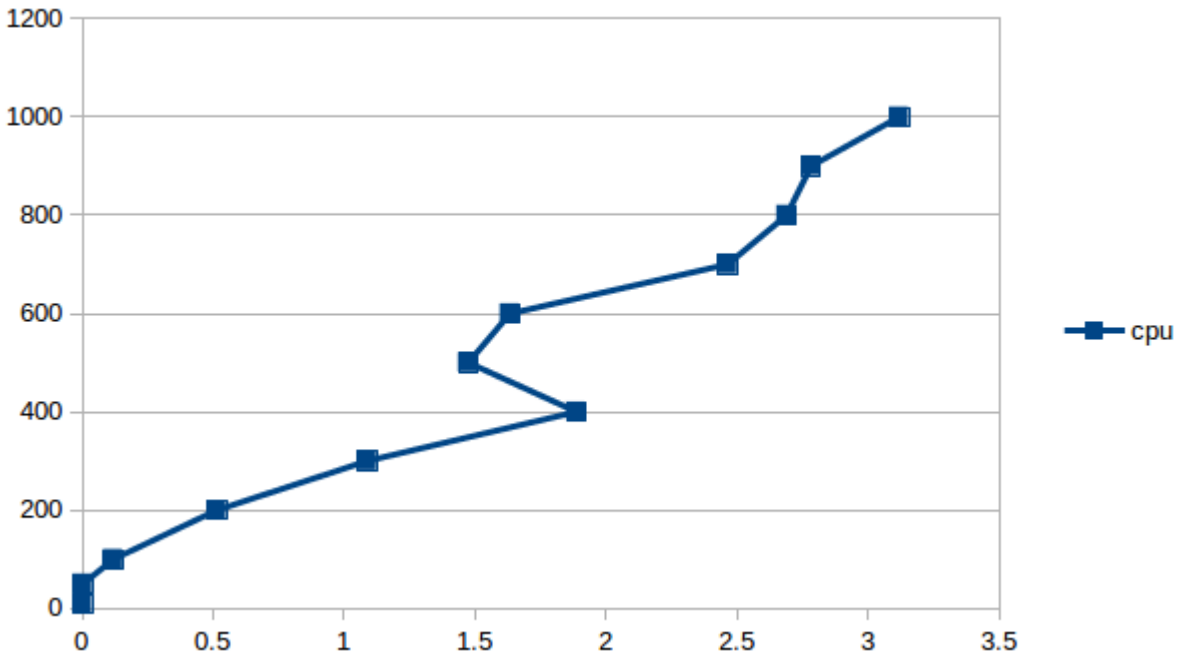

Figure 6: Graph of the runtimes for the sequential calculations, for the recorded sizes.

4 Conclusion

In writing the code for the implementation of adding two matrices. I did run into issues with the string. From the collected data, I will continue to try and determine better values for both the number of blocks and number of threads per block. This will allow me to grasp these concepts even better. From what we observed in our data, we will notice that in most cases, the GPU implementations outperformed the CPU implementations. In the cases where this was not true; it was probably due to my inability to determine the appropriate number of blocks or number of threads. Similarly, we noticed that there was a performance increase with striding over without striding.