Matthew VanCompernolle
CS791v 1/28/15

PA1 Report – Vector Addition

Introduction

The purpose of the project is to implement a simple program in parallel on the graphical processing unit (GPU) and to determine how the parallel implementation compares to a sequential implementation on the CPU. The program implements vector addition on the GPU using CUDA, which is Nvidia's parallel computing platform. Vector addition can be implemented very easily using very few lines of code, and is also embarrassingly parallel because the addition of one index two vectors in completely independent from the addition of all of the others. A second primary goal of the project is to practice with CUDA and to determine which combinations of parallelization using blocks and threads yield the best performance.

Theory

Vector Addition

The operation that this project implements in parallel is vector addition for very large vectors. Vector addition is the operation of summing two or more vectors together by adding the corresponding values at each index of vectors being summed into the corresponding index of a result vector. For example, to add two vectors together you would use the following equation:

$$c[i] = a[i] + b[i]$$

For every element i to add the elements of vector and vector b into vector c, which all must have the same magnitude.

Speedup Factor

For this project it is necessary to determine how much faster a parallel implementation of vector addition is than the best sequential algorithm. The speedup factor can be used to measure this and is defined as:

$$S(p) = \frac{t_s}{t_p}$$

S(p), or the speedup factor, determines how many times faster a parallel system using a parallel algorithm is than the best sequential algorithm. In the equation, t_s is the execution time of the best sequential algorithm running on a single processor and t_p is the execution time of the parallel algorithm running on the GPU.

Results

Sequential Implementation

The sequential implementation was executed on an Intel i7 4970k running at 4 GHz with a maximum turbo frequency of 4.4 GHz. The implementation was executed using 3 different vector sizes that were selected because they are large and memory intensive but still small enough to fit on the GPU in the parallel implementation. The 3 sizes selection were 1 million, 25 million, and 250 million. Even at these sizes computation times were extremely quick, with each of the 3 sizes taking less than a second. This is due to the fact that vector addition only has a time complexity of O(n), which can be seen in Figure 1. The run times produced by the sequential implementation for the three vector sizes are shown in Table 1.

Table 1: Sequential Implementation Run Times in Seconds.

Vector Elements (millions)	Run Time (ms)
1	2.1006
25	51.3306
250	509.339

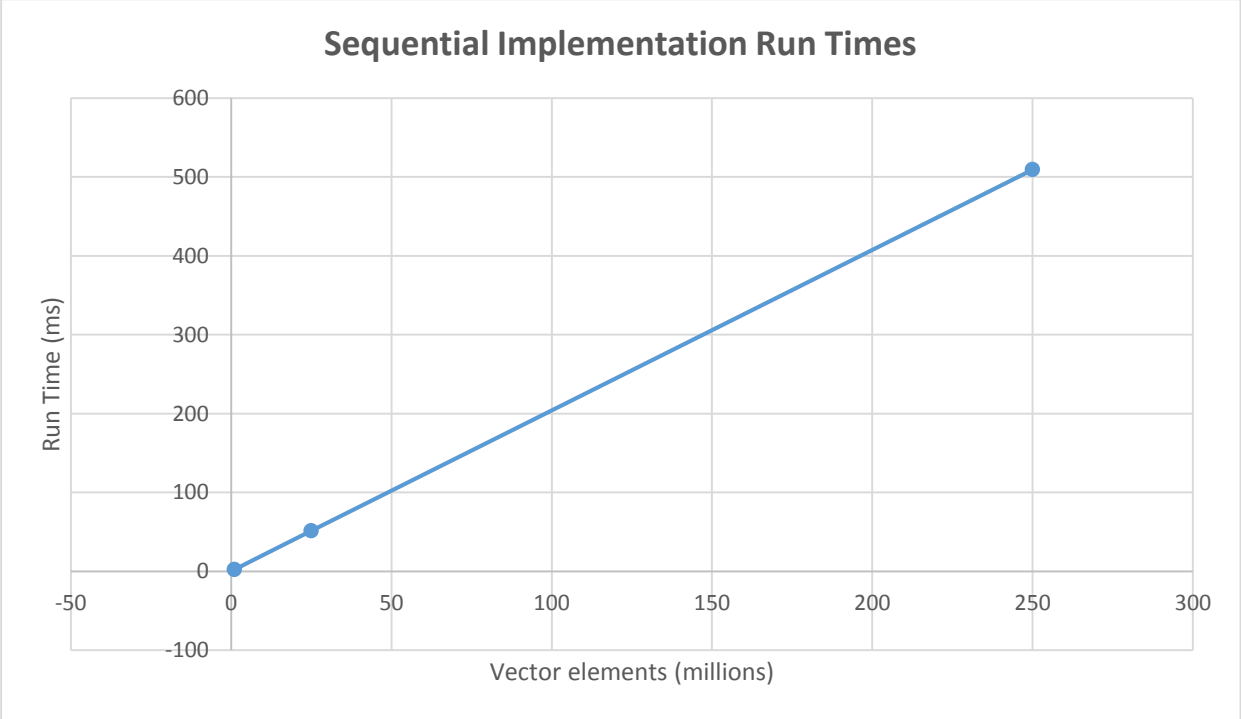

Figure 1: This graph illustrates the linear time complexity of vector addition and how little time it takes to compute.

Parallel Implementation

The parallel Implementation was executed on an Nvidia GTX 970 with 1664 CUDA cores running at a boost clock rate of 1.178 GHz. The GPU has 4 GB of memory, which was determined to limit the size of the vectors being summed to about three hundred million elements. Two different parallel implementations were used in the execution of the project. The first was a parallel implementation using striding, which essentially allows a single kernel to perform one element of vector addition, then repeatedly make constant sized jumps forward to other elements of the vectors until the end is reached. The main benefit of this implementation is that it allows the GPU compute vector addition on sizes larger than the maximum numbers of blocks times the maximum number of threads it can support, which in this case was 65536 times 1024. The second implementation was the non-striding method which only performs one element addition per kernel, limiting it to vector additions of sizes 67,108,864 or less. As a result, the non-striding method has instances of block and thread combinations that were not large enough to compute certain vector sizes. These instances appear as zeros on the graphs, because they were not able to execute the addition. For example, Figure 6 illustrates how almost two thirds of the block and thread combinations were not able to compute vector addition of size 25 million. The non-striding version was also unable perform vector addition of 250 million with any block and thread combo, so the corresponding graphs were omitted.

Figure 2 through Figure 11 illustrate the run times and speedup factors via graphs for both striding and non-striding parallel implementations. The graphs are ordered below by vector size starting at 1 million and ending at 250 million, and alternate between non-striding and striding implementations. Each of the graphs take into the amount of time it took to transfer the vector data both to and from the GPU in the computation, which was determined to take a significant portion of the total time in every case. When analyzing any of the graphs it is import to note that the graphs do not take into account a block size of one running with a thread size of one because that would make the implementation sequential. The position of the graph corresponding to 1 block and 1 thread should be ignored in all cases, as it is always zero and does not use real data.

Data for the parallel versions were collected by running the different variations of parallel implementations and vector sizes with 1 block and 1 thread up to 65531 blocks and 1021 threads with many variations between. The step size selected for block variations was 6553 (10% of the total available blocks) and the step size selected for thread variations was 102 (10% of the total available threads).

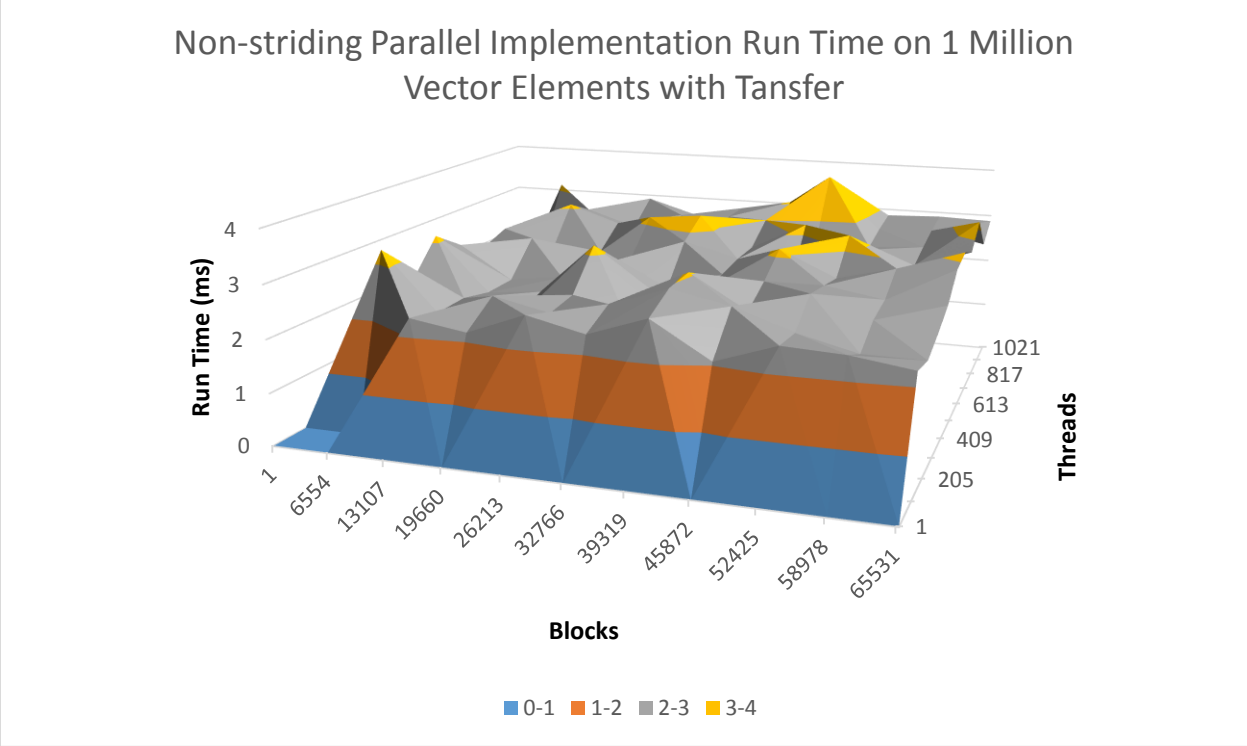

Figure 2: This graph shows the various run times (including I/O transfer time) on 1 million elements the non-striding parallel implementation took using variety of number of threads and blocks.

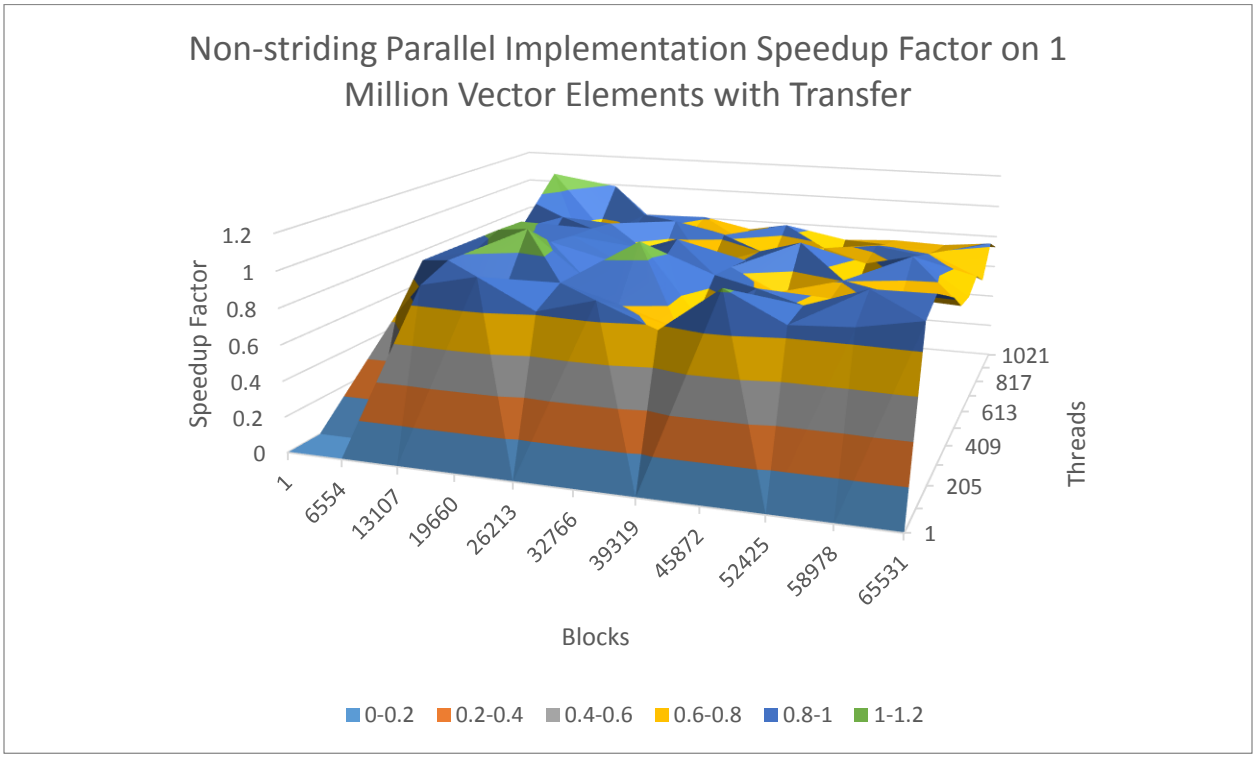

Figure 3: This graph shows the GPU speedup factor of the non-striding parallel implementation using a variety of number of threads and blocks on 1 million elements.

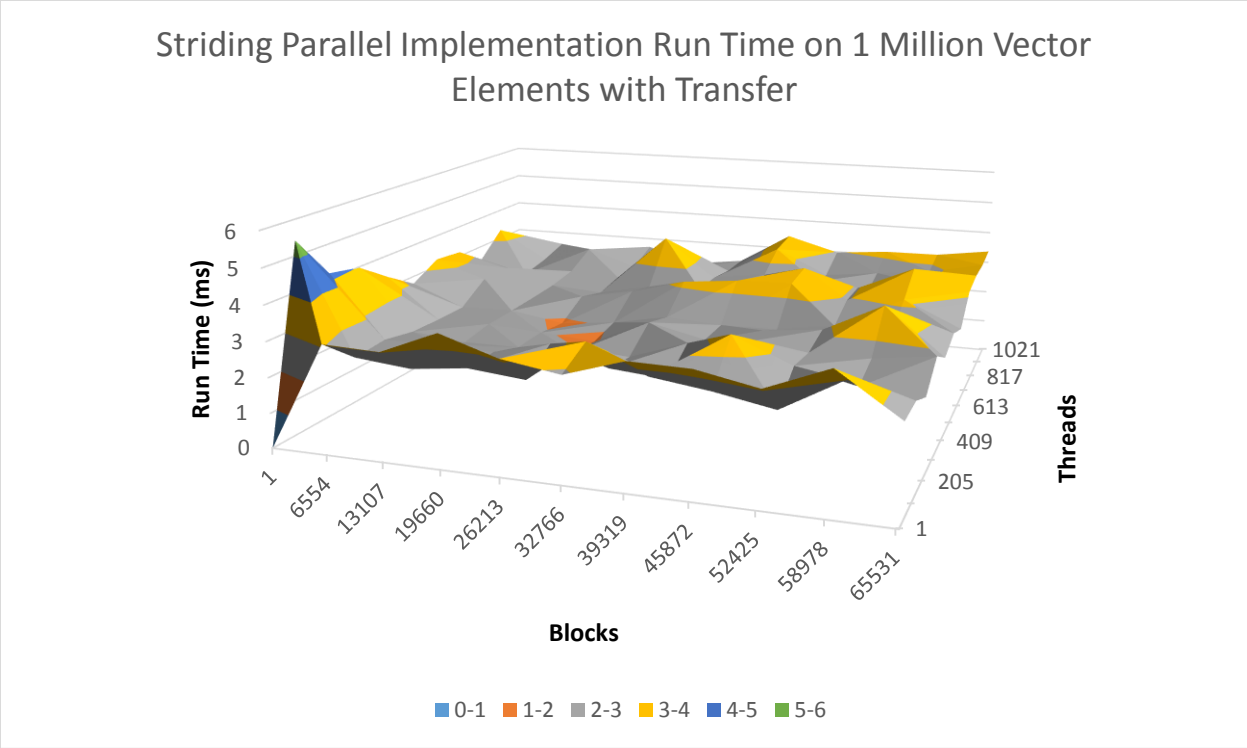

Figure 4: This graph shows the various run times (including I/O transfer time) on 1 million elements the striding parallel implementation took using variety of number of threads and blocks.

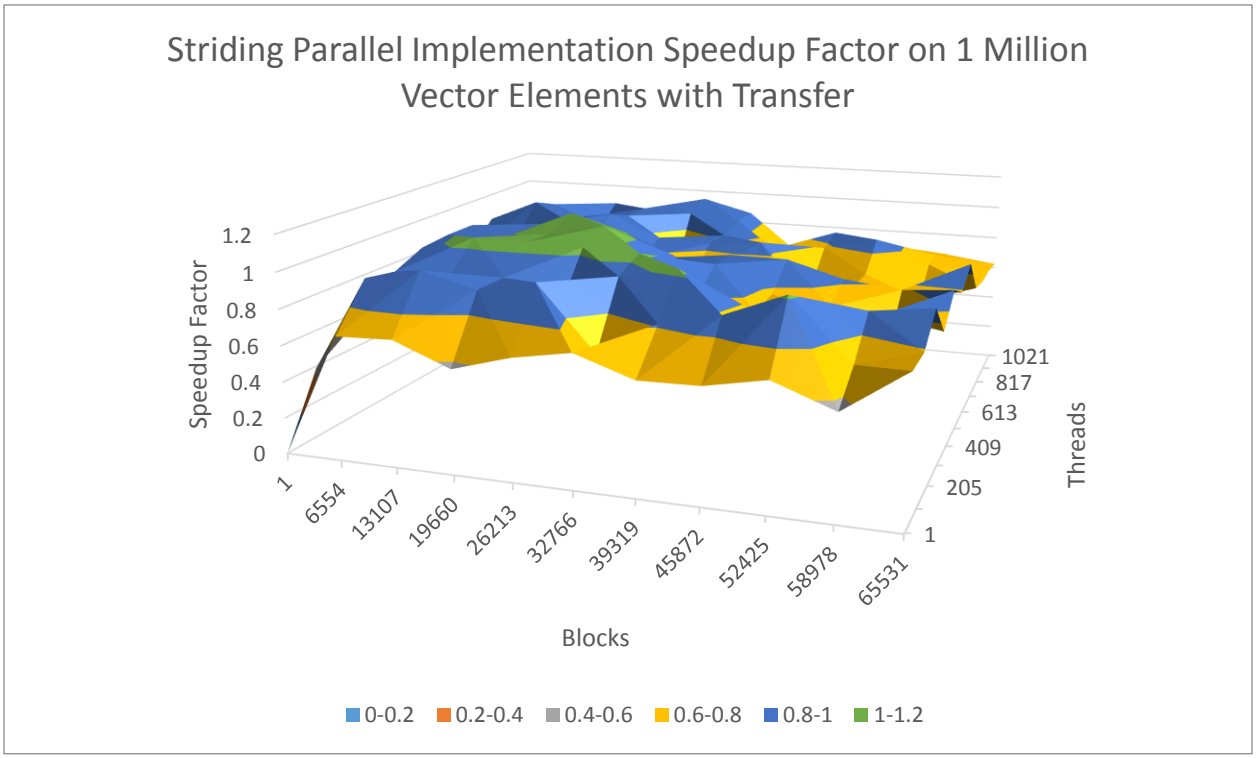

Figure 5: This graph shows the GPU speedup factor of the striding parallel implementation using a variety of number of threads and blocks on 1 million elements.

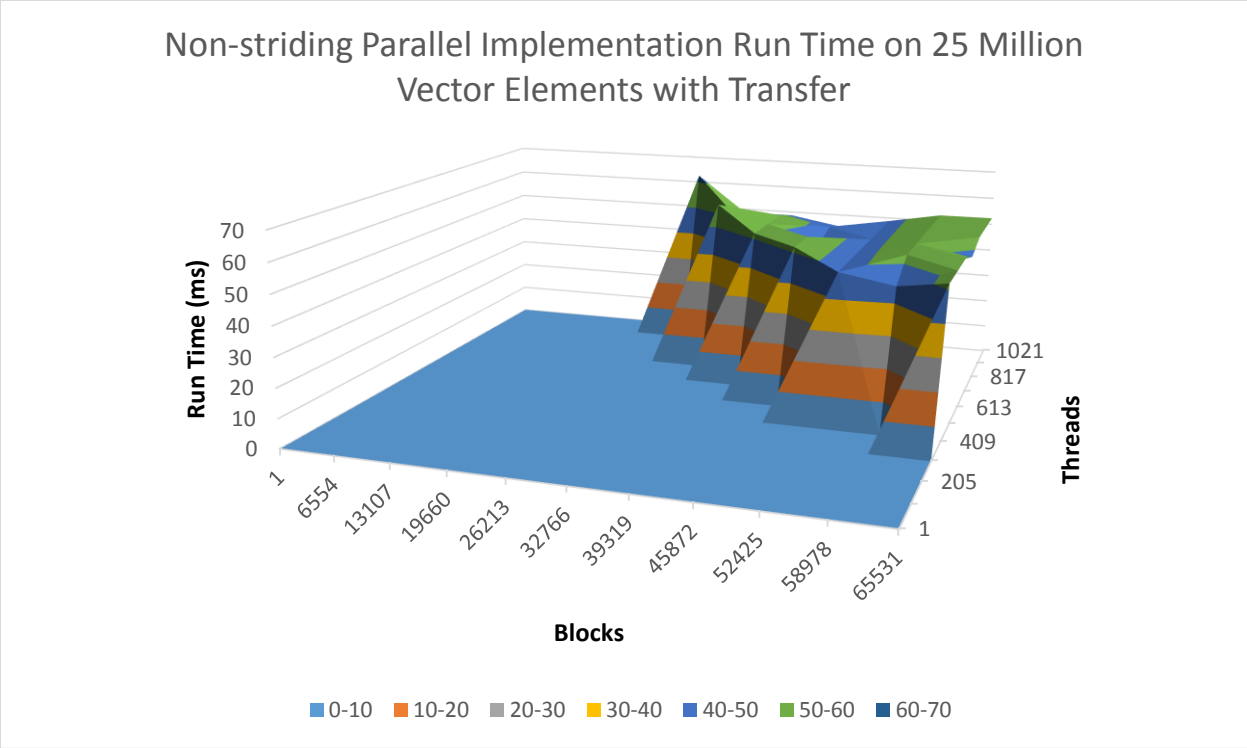

Figure 6: This graph shows the various run times (including I/O transfer time) on 25 million elements the non-striding parallel implementation took using variety of number of threads and blocks.

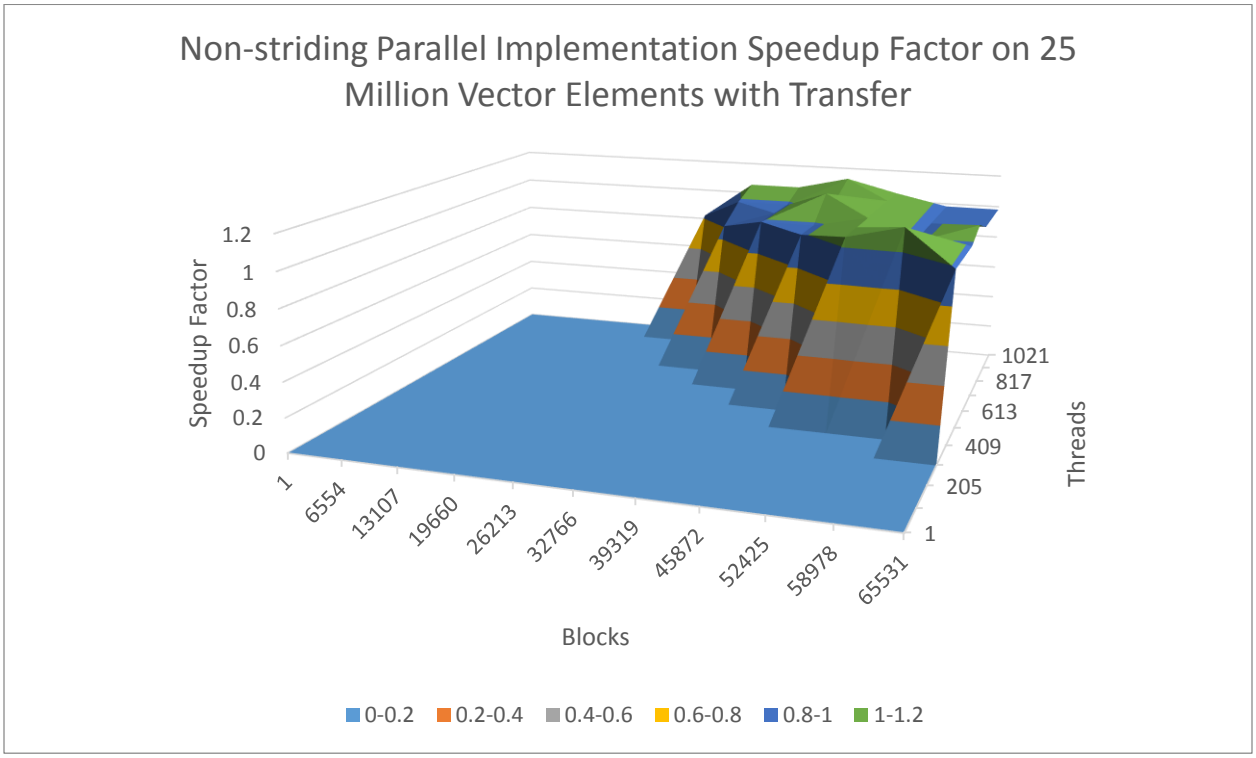

Figure 7: This graph shows the GPU speedup factor of the non-striding parallel implementation using a variety of number of threads and blocks on 25 million elements.

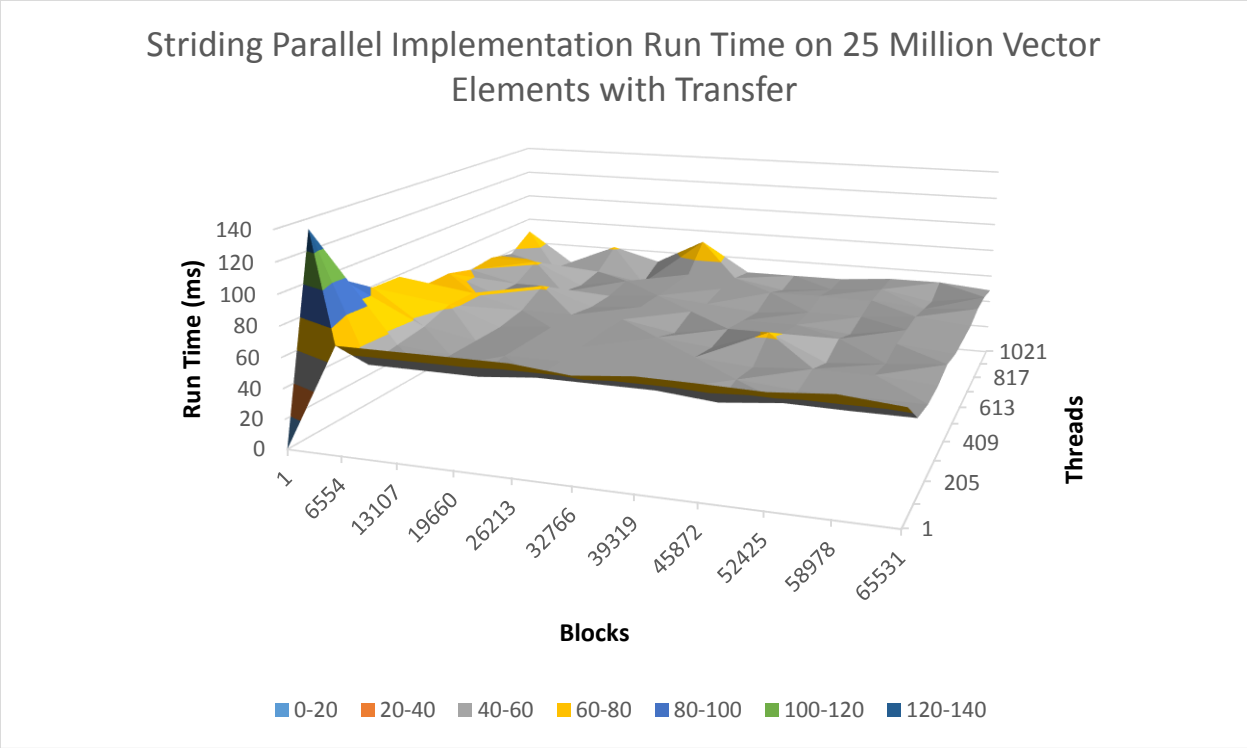

Figure 8: This graph shows the various run times (including I/O transfer time) on 25 million elements the striding parallel implementation took using variety of number of threads and blocks.

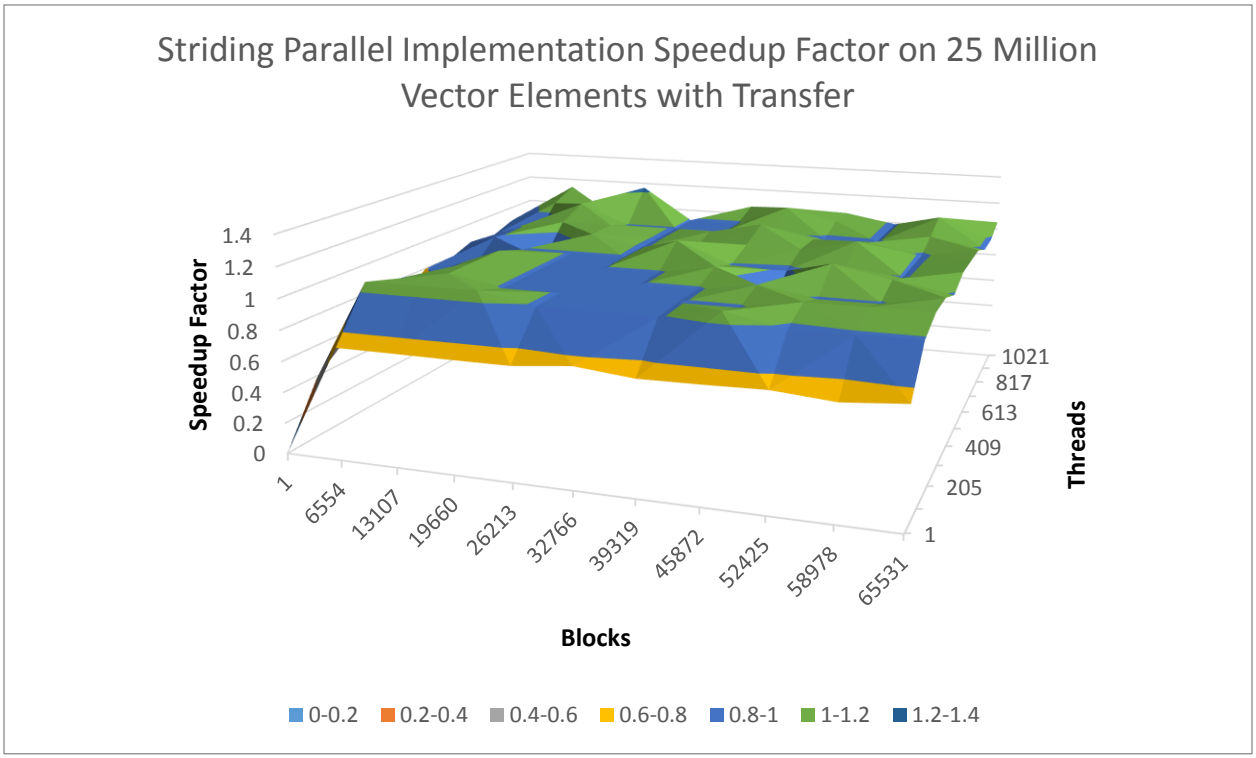

Figure 9: This graph shows the GPU speedup factor of the non-striding parallel implementation using a variety of number of threads and blocks on 1 million elements.

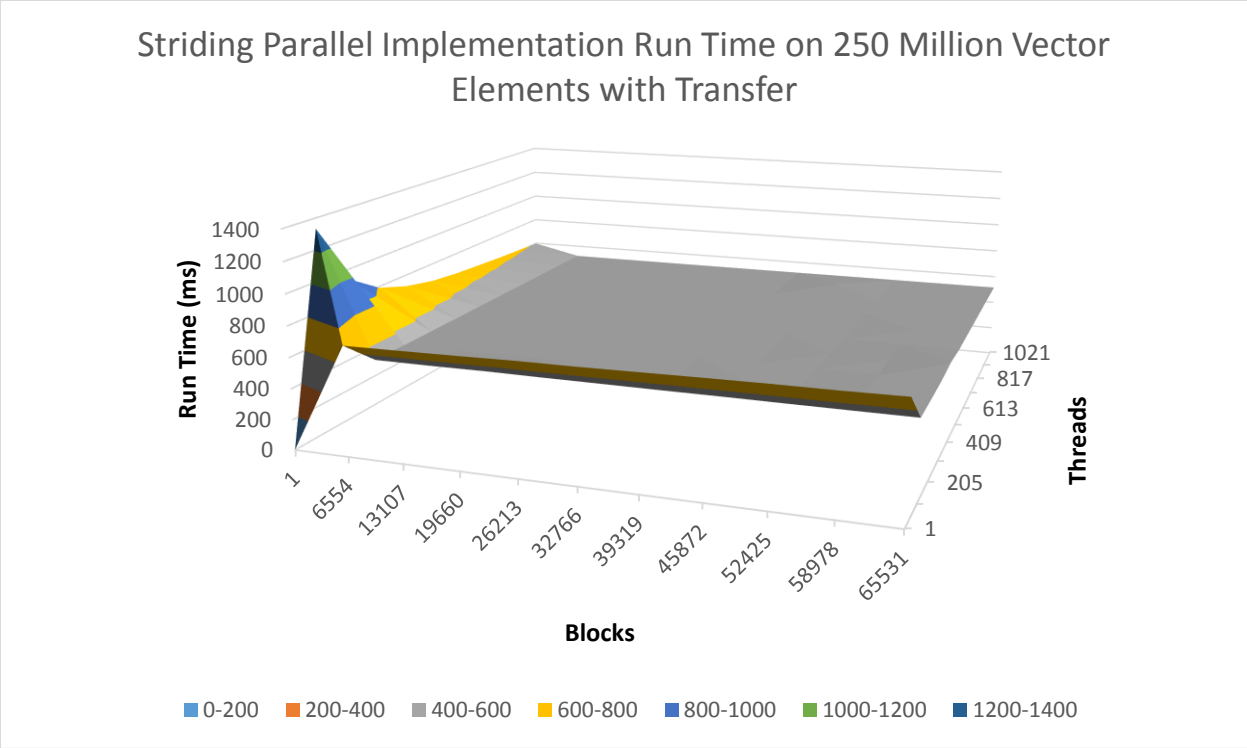

Figure 10: This graph shows the various run times (including I/O transfer time) on 250 million elements the striding parallel implementation took using variety of number of threads and blocks.

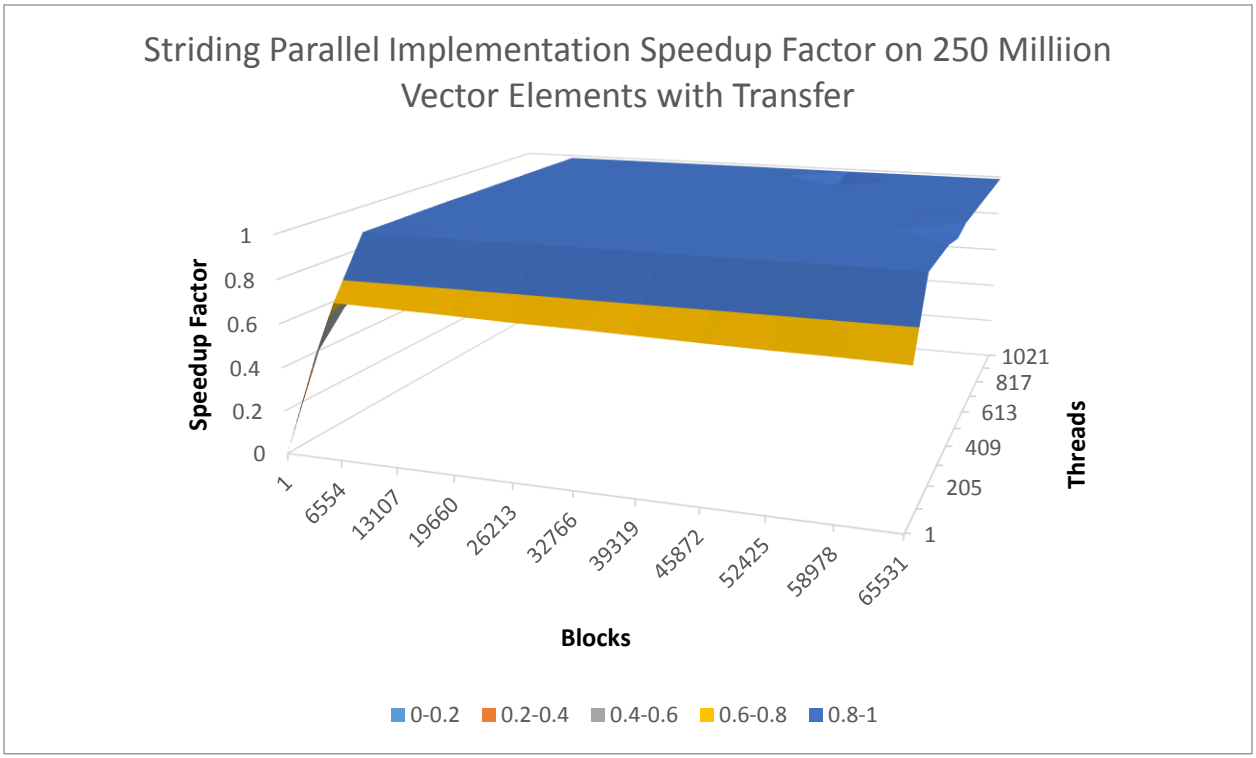

Figure 11: This graph shows the GPU speedup factor of the striding parallel implementation using a variety of number of threads and blocks on 250 million elements.

Runtime without transfer time

Although it is not very practical to consider the use of a GPU for parallel use and ignore the amount of time it takes to transfer data to and from it, it is useful to determine the costs of using the GPU as an I/O device. Figures 12, 13, and 14 show the run time the striding implementation without transfer time taken into account for both 1 million, 25 million, and 250 million sized vectors. When compared to Figures 4, 8, and 10 that do take into account the transfer time, it is evident that using the GPU comes at a very large cost.

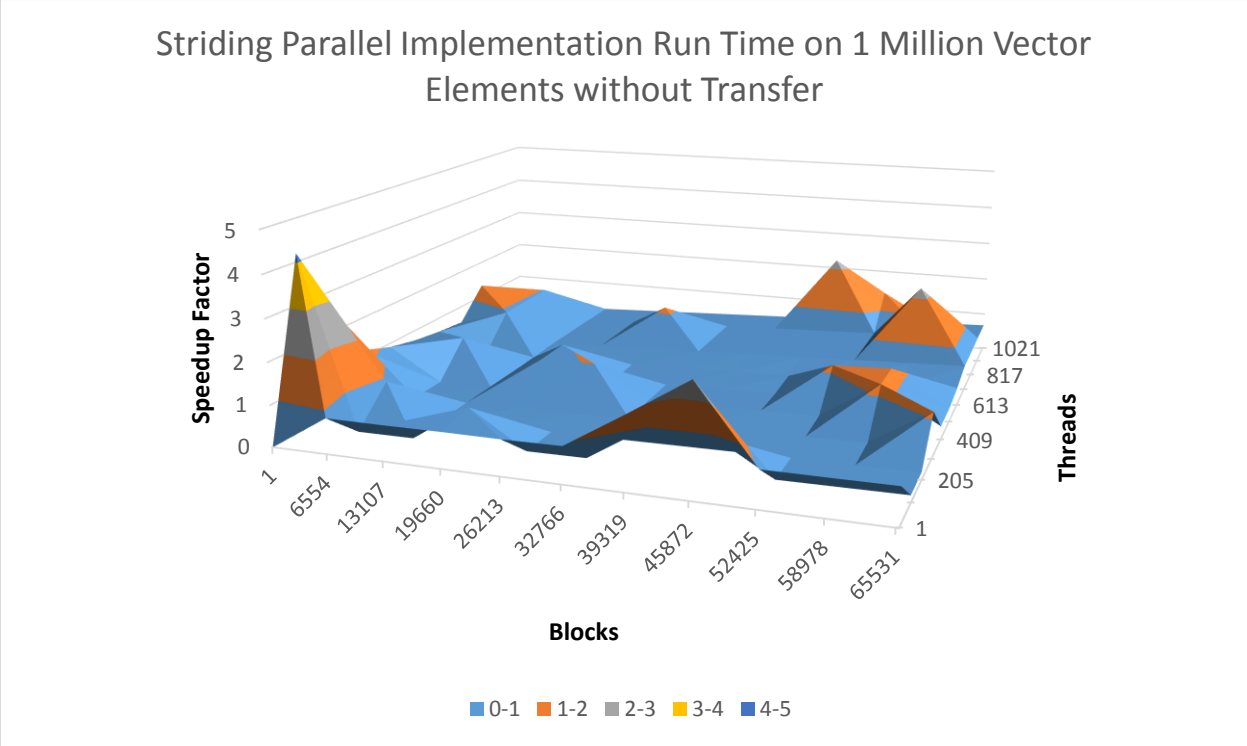

Figure 12: This graph shows the various run times (excluding I/O transfer time) on 1 million elements the striding parallel implementation took using variety of number of threads and blocks.

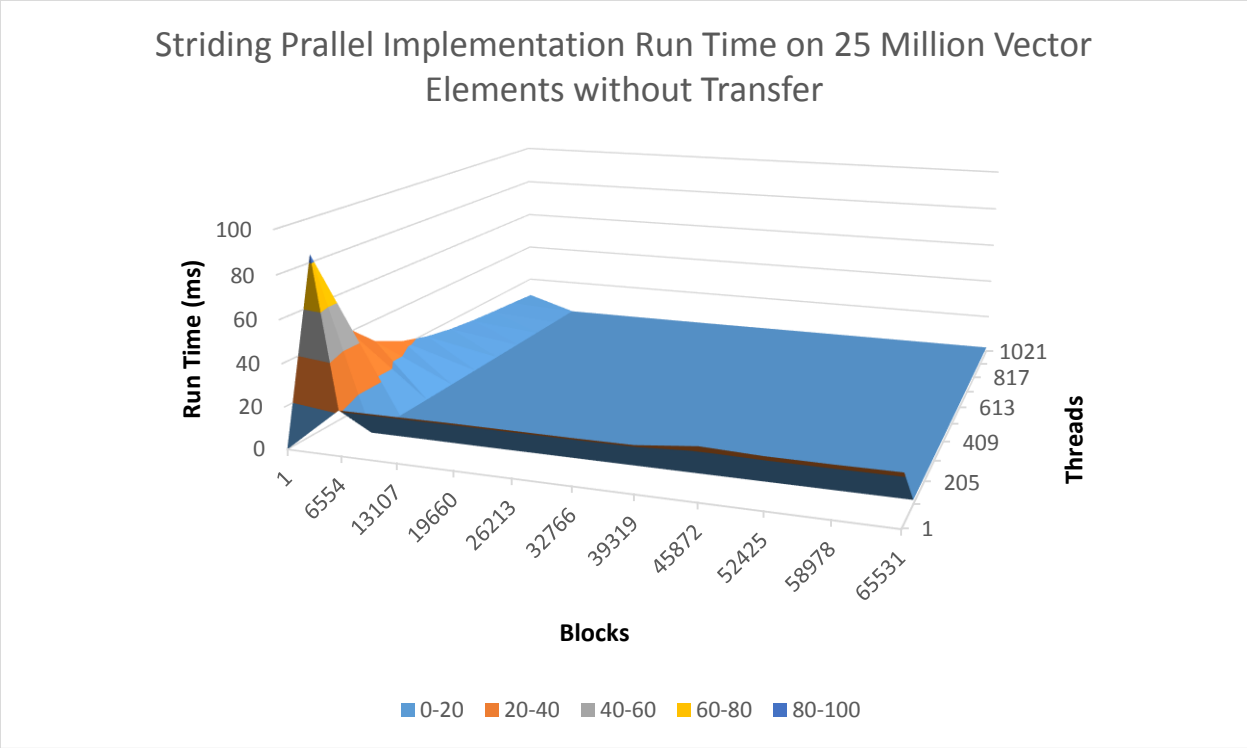

Figure 13: This graph shows the various run times (excluding I/O transfer time) on 25 million elements the striding parallel implementation took using variety of number of threads and blocks.

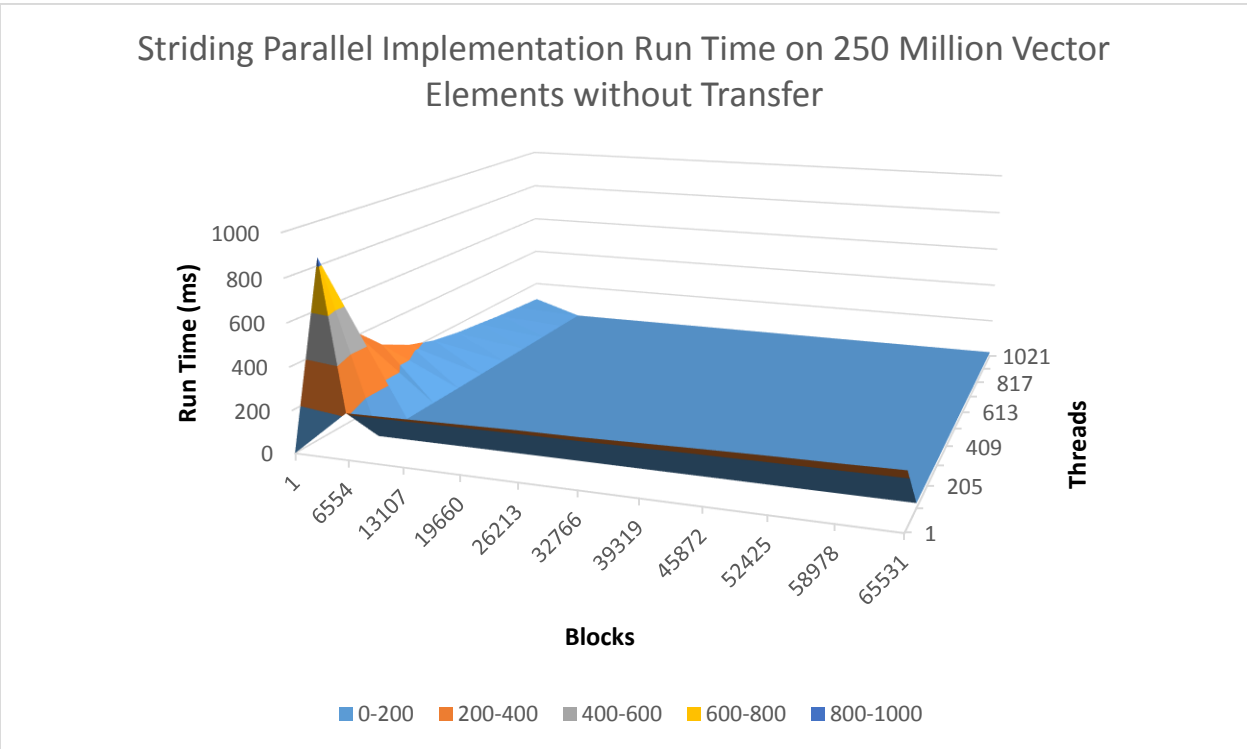

Figure 14: This graph shows the various run times (excluding I/O transfer time) on 250 million elements the striding parallel implementation took using variety of number of threads and blocks.

Analysis

Block and Thread Sizes

Even with the abundance of data gathered it is not exactly clear what the ideal choice for the number of blocks and threads is. One conclusion that can be made is that parallel performance is often worse when dealing with a very small number of threads or a very small number of blocks. This is probably explained by the fact that having few blocks or threads limits how well the GPU can parallelize the computations. Other than these two cases, results appear too random to easily recognize a pattern. It would probably take a good amount of knowledge of both CUDA and the device (GPU) the program is being ran on to pick the best combination possible of blocks and threads. Even then, the differences in performance between most block and thread number variations is not extreme.

Data Transfer Cost

The cost of transferring data to and from the GPU is very expensive, and in many cases would make it not worth implementing a program on the GPU at all. Vector addition is one computation that is not worth the cost, as shown by the results. Figures 3, 5, 7, 9, and 11 all show that the GPU rarely experienced any speedup, and in most cases was slower, than the sequential implementation when data transfer time was taken into account. The fact that a highly parallelized implementation was not better than a sequential one when dealing with an embarrassingly parallel problem says a lot about how extreme transfer time actually is. When comparing the same computation with and without time transfer, the difference is huge. For example, when looking at Figure 8 and Figure 13, you can see that the same computation took almost 25 times less time in most cases when excluding transfer timings. If data did not have to transfer to the GPU, the parallel implementation of vector addition would have around a speedup factor of around 20 over the best sequential implementation, as supported by the data.

Striding vs. Non-striding

The striding parallel implementation has the benefit of being able to compute vectors of sizes larger than the maximum number of blocks times the maximum number of threads supported by the GPU because of the jumps it makes to un-computed portions of data. The non-striding implementation has the limitation of only being able to make computations of a fixed size or smaller, even if the GPU can store much more in device memory. This alone makes striding the superior of the two methods used. In terms of performance, both the striding and non-striding methods produced very similar results, at least in the various scenarios of this project.

Issues

I did not experience very many issues in implementing the project. Most of the code was provided, and any additional code that needed to be added was fairly straight forward. I do however still have a hard time grasping how the number of blocks and the number of the threads you provide to a kernel is used on the GPU's hardware. One minor issue that I had with the project is the vagueness of the report guidelines. In the future I hope that reports will be given more structure to follow.