PA01: Comparing Vector Addition Performance on the CPU and the GPU

Terence Henriod
Dr. Fred Harris
Dr. Lee Barford

CS791v: Parallel Computing
Thursday January 29, 2015

Introduction

Throughput is of particular significance in scientific computing. The GPU architecture was designed originally to produce large amounts of throughput for graphical applications, but has since been adapted for general purpose high performance computing. In this assignment, we explore the performance comparison between a GPU using many different thread configurations and a single threaded CPU. The kernel used is a simple, classic SIMD one: vector addition.

Theory

The CPU architecture features high performance processing cores capable of very high performance and a wide array of operation. This can be good for performing low-latency work, but the complexity of the cores limits the number of cores that can fit on a die/chip.

The GPU architecture features many, simpler cores that do not perform as highly as a CPU core on their own, but their relative lack of complexity allows many more cores to be fit on a die/chip. Further, GPUs tend to feature enhanced memory bandwidth in order to supply memory to the compute operations even faster.

It is well established that GPUs can produce speedup for many kernels, especially those that are SIMD in nature. We are now interested in *how much* speedup can be gained. That question may not be answered here, as this was an exploratory exercise to familiarize ourselves with the basic capabilities of the GPU.

Results

The following are a collection of results that were discovered in performing various vector addition trials. It should be noted that while many trials were tried, it would be staggering to try all parameter combinations (there are 65535 * 1024 = 67107840 possible combinations of blocks and threads for any vector size, and many vector sizes could be used). For this reason, among others described in the issues section, there are gaps in the combinations used, but general trends should be sufficiently observable.

GPU Information	
Device Name	NVS 5400M
Cuda Version	2.1
Multiprocessors	2
CUDA Cores	96
Clock Rate	950 mHz
Total Global Memory	1073 MB
Warp Size	32
Max Threads/Block	1024
Max Threads-Dim	1024 x 1024 x 64
Max Grid Size	65535 x 65535 x 65535

Size per element	4 bytes

Table 1: GPU Specifications for this exercise.

CPU Information	
Device	i5-3360M
Cores	2 (4 with Hyper-threading)
Clock Rate	2.80 GHz
System RAM	15.7 GB

Table 2: CPU Specifications for this exercise.

Allocation Failures			
Device	**Vector Size (number of int values)**	**Memory per Vector**	**Reason for Failure**
GPU	10,000,000	40 MB	OS Defined Timeout
	85,000,000	340 MB	Memory Allocation Failure
CPU	95,000,000	380 MB	Memory Allocation Failure

Table 3: Vector sizes that resulted in kernel failure.

Maximum Speedups Observed				
	Non-Striding Algorithm		**Striding Algorithm**	
	Speedup	**Vector Size; Blocks, Threads**	**Speedup**	**Vector Size; Blocks, Threads**
Compute Time Only	31.3	10,000,000; 512, 256	15.41	10,000,000; 4096, 64
Total Time (Compute + Transfer)	3.95	10,000,000; 4096, 256	3.59	10,000,000; 4096, 64

Table 4: Maximum speedup obtained using the GPU.

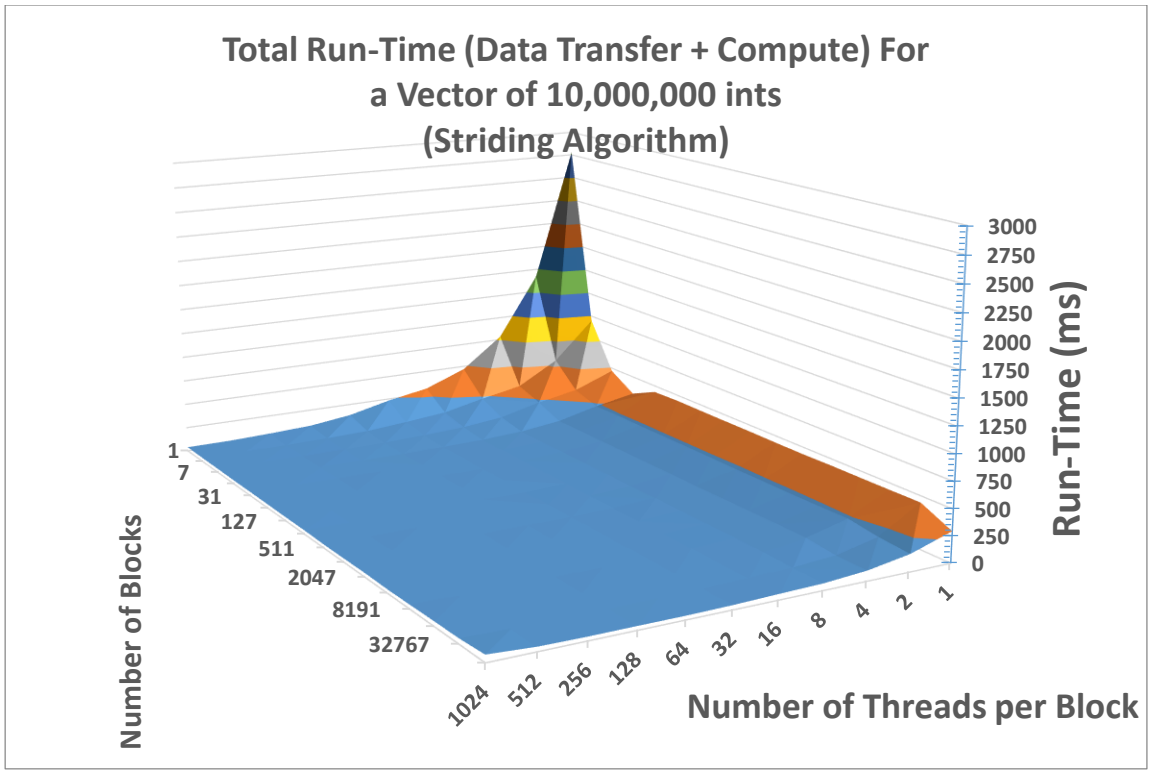

Figure 1

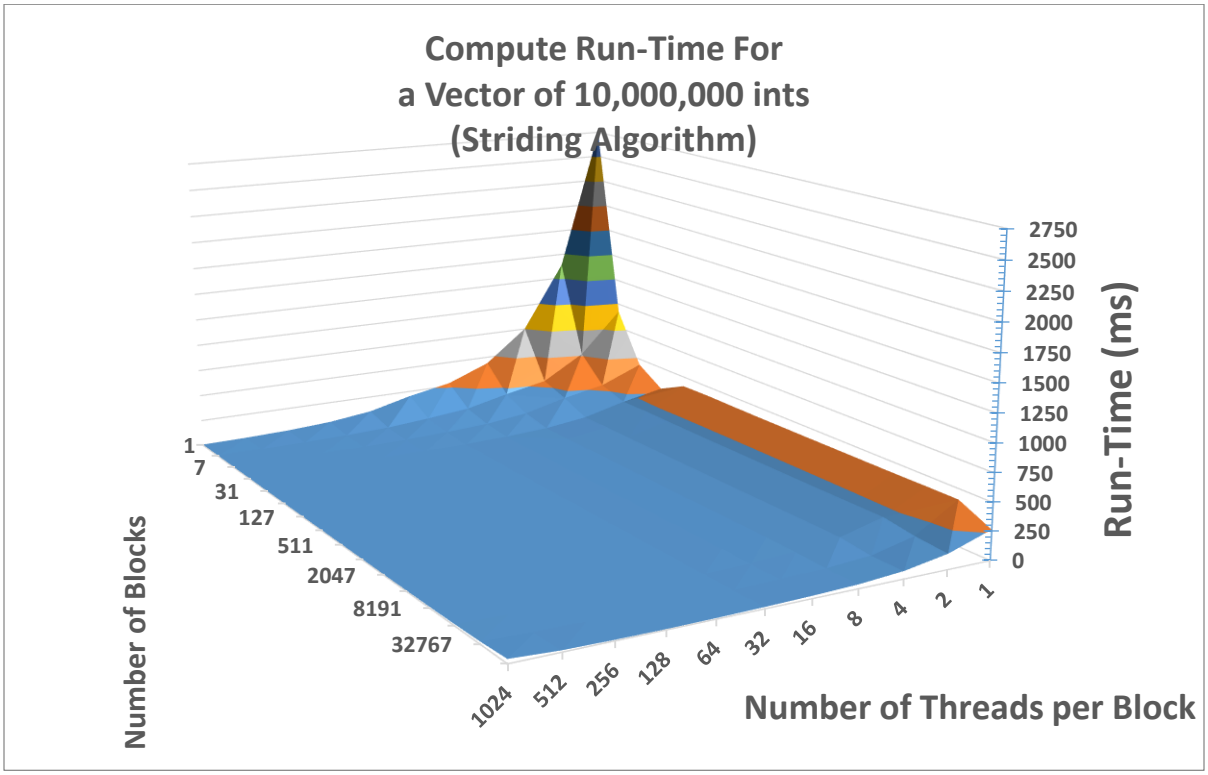

Figure 2

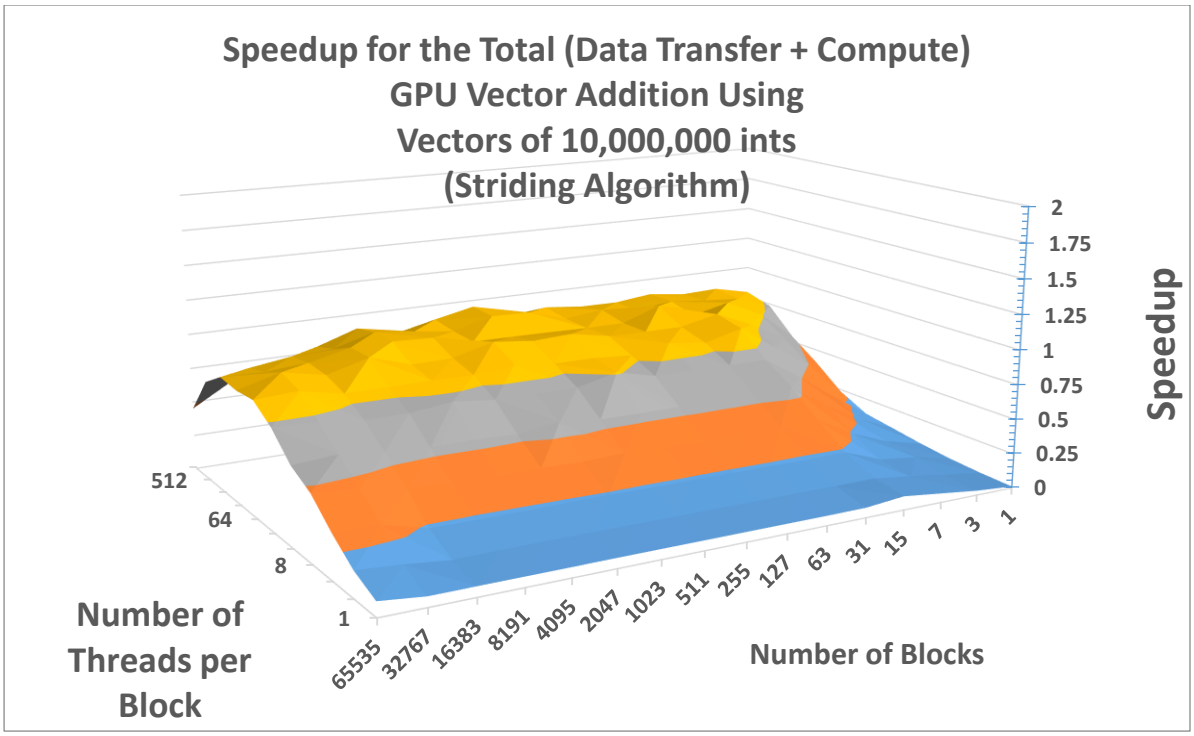

Figure 3

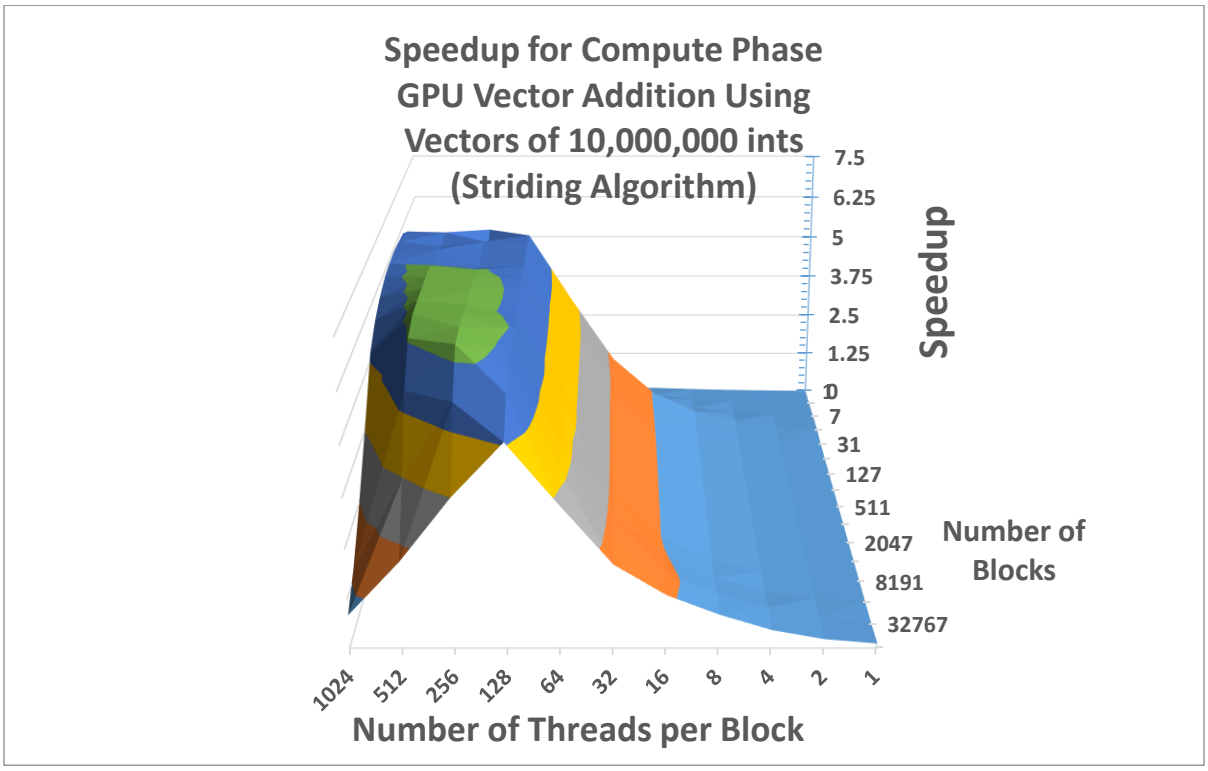

Figure 4

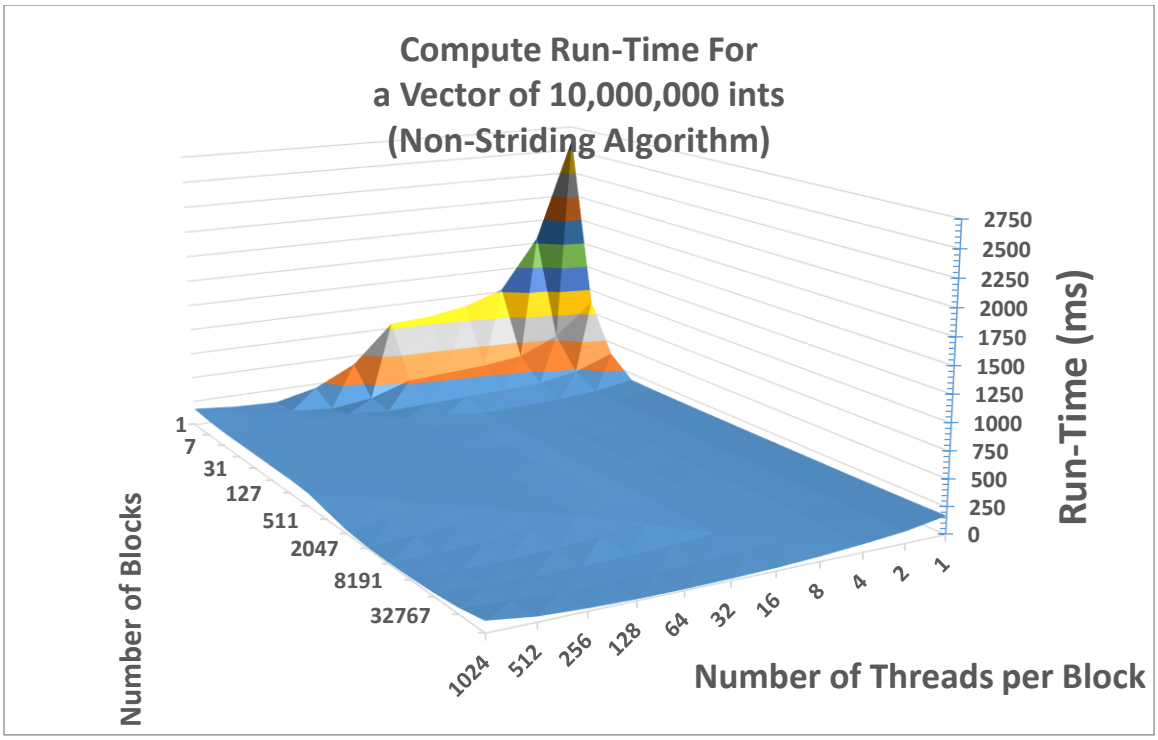

Figure 5

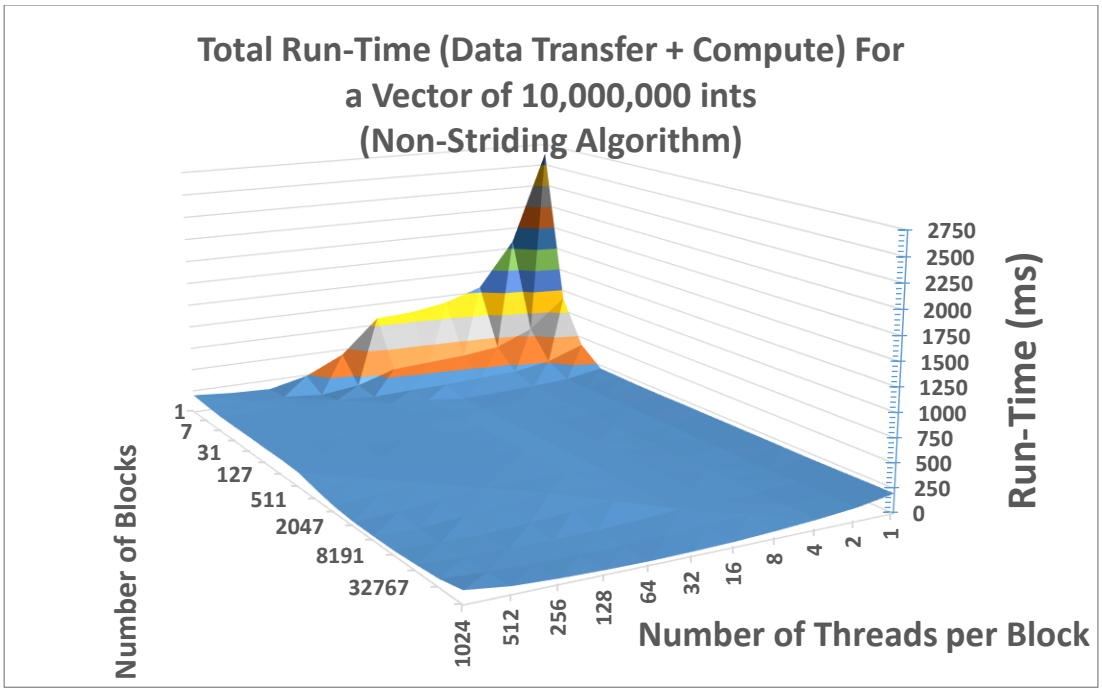

Figure 6

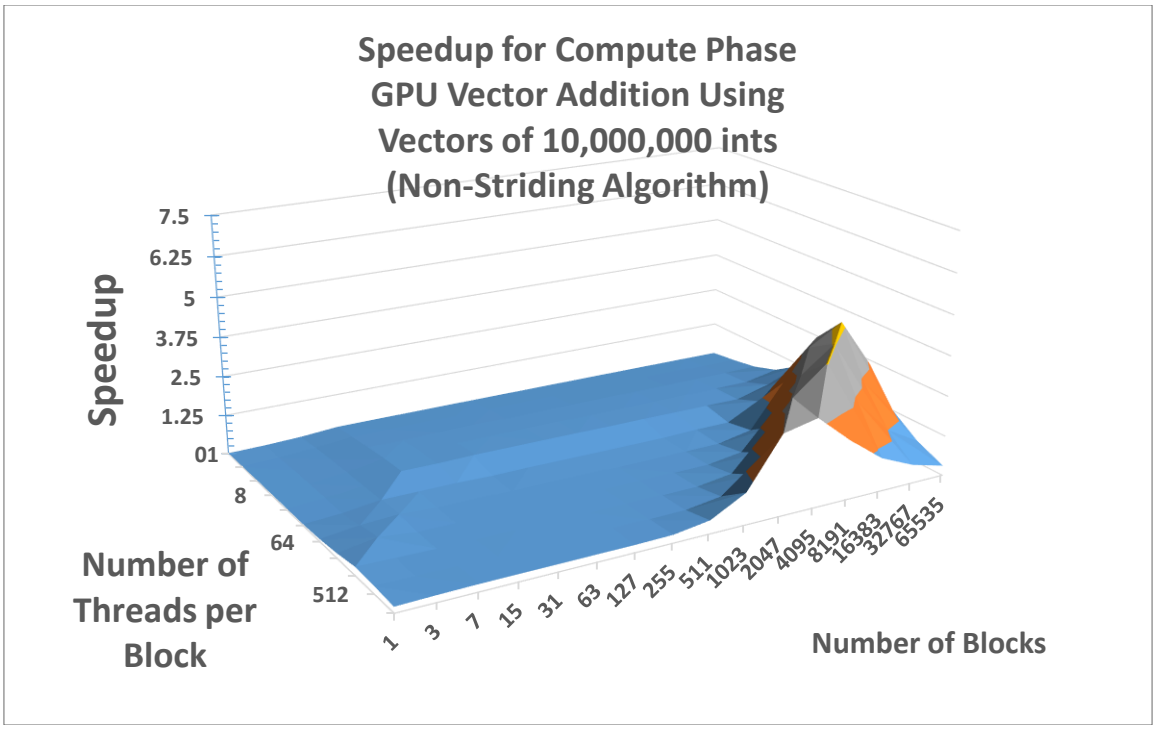

Figure 7

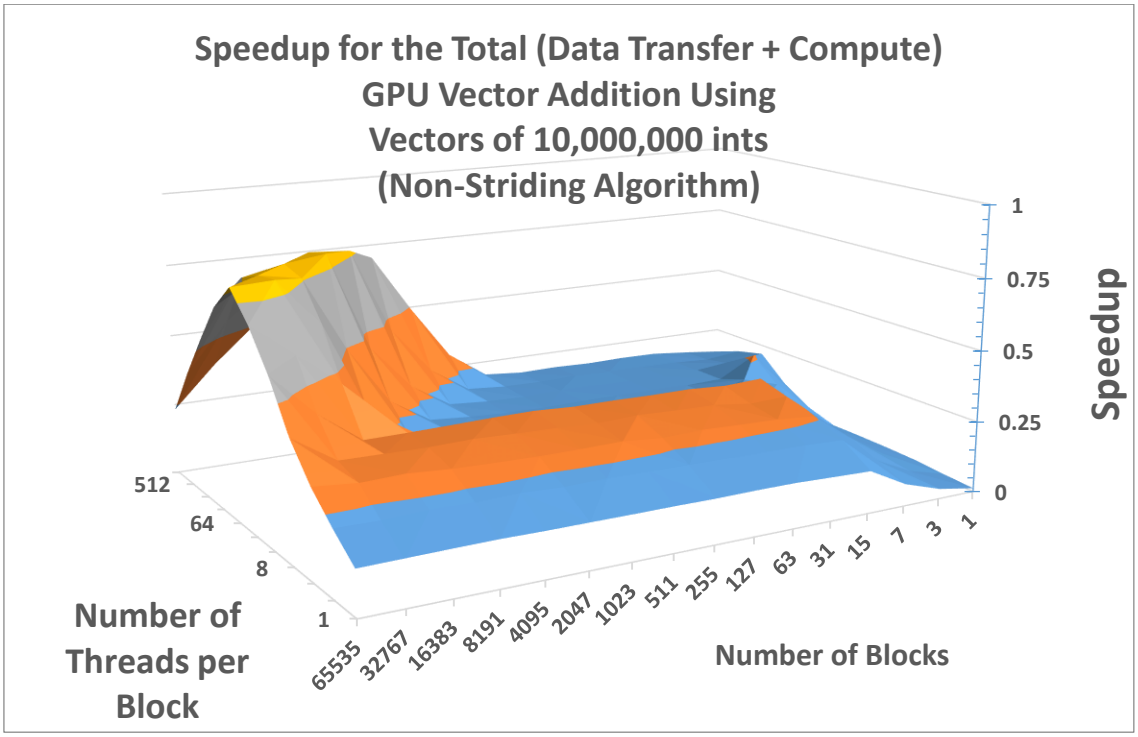

Figure 8

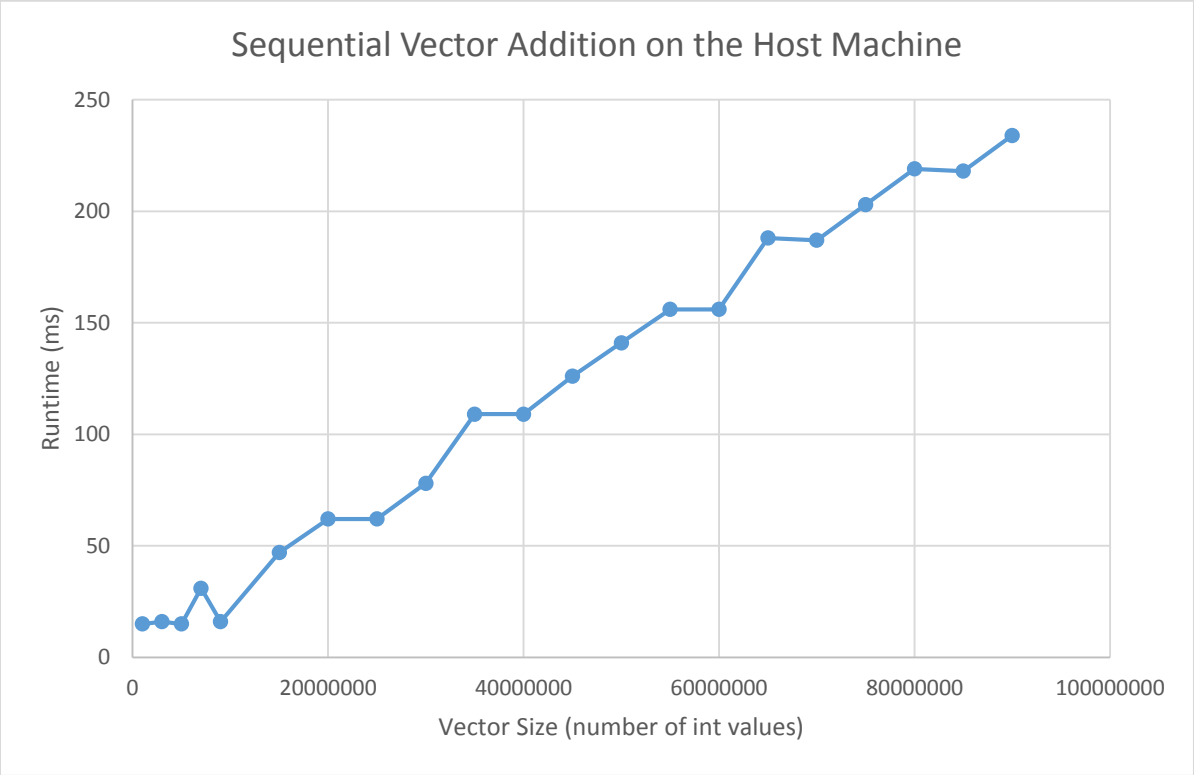

Figure 9

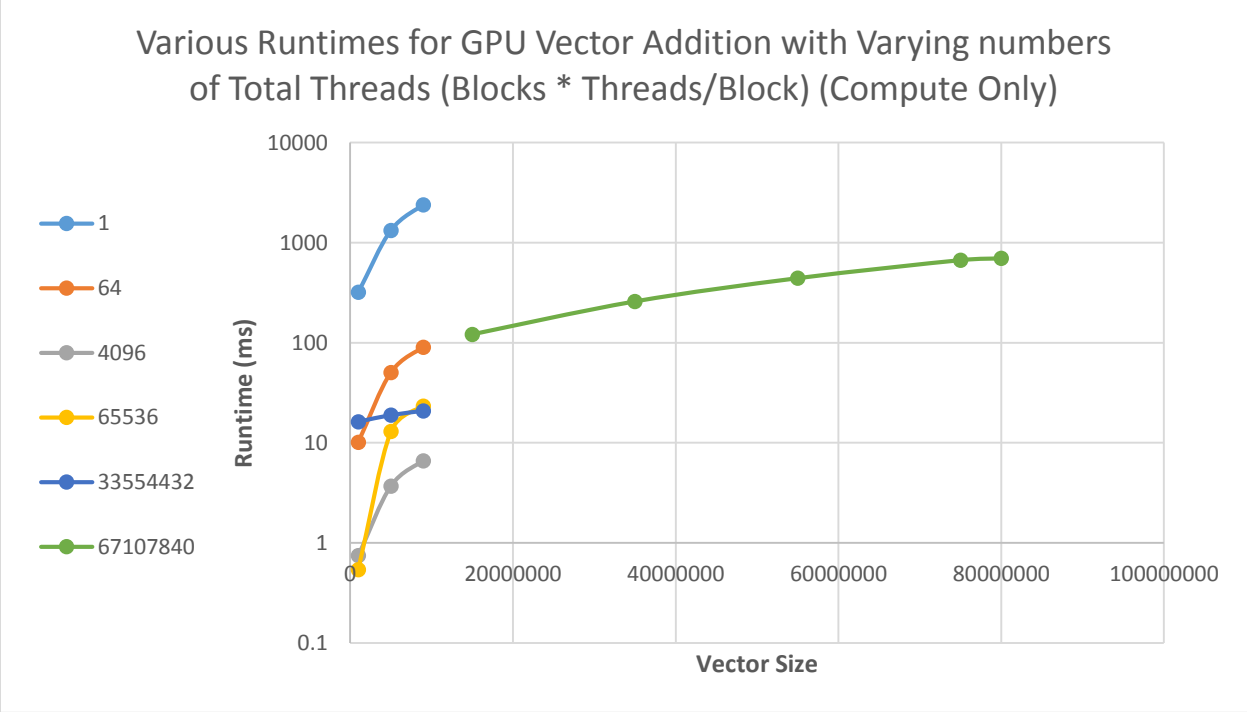

Figure 10

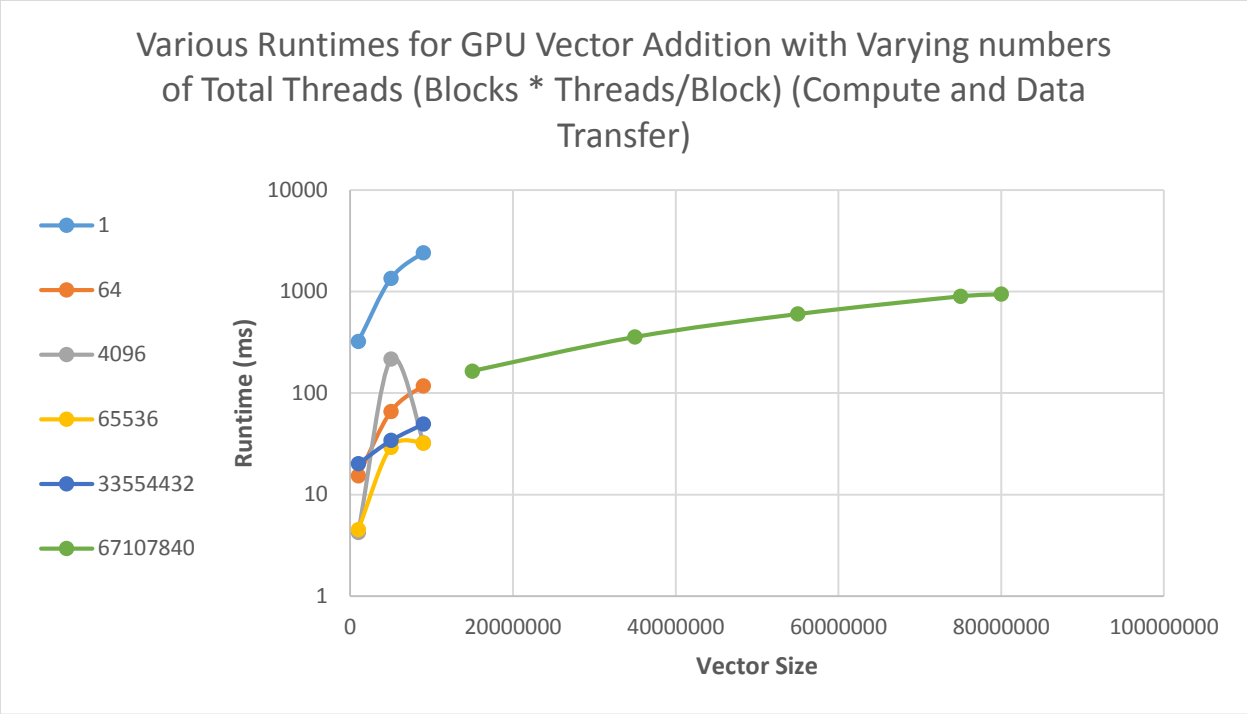

Figure 11

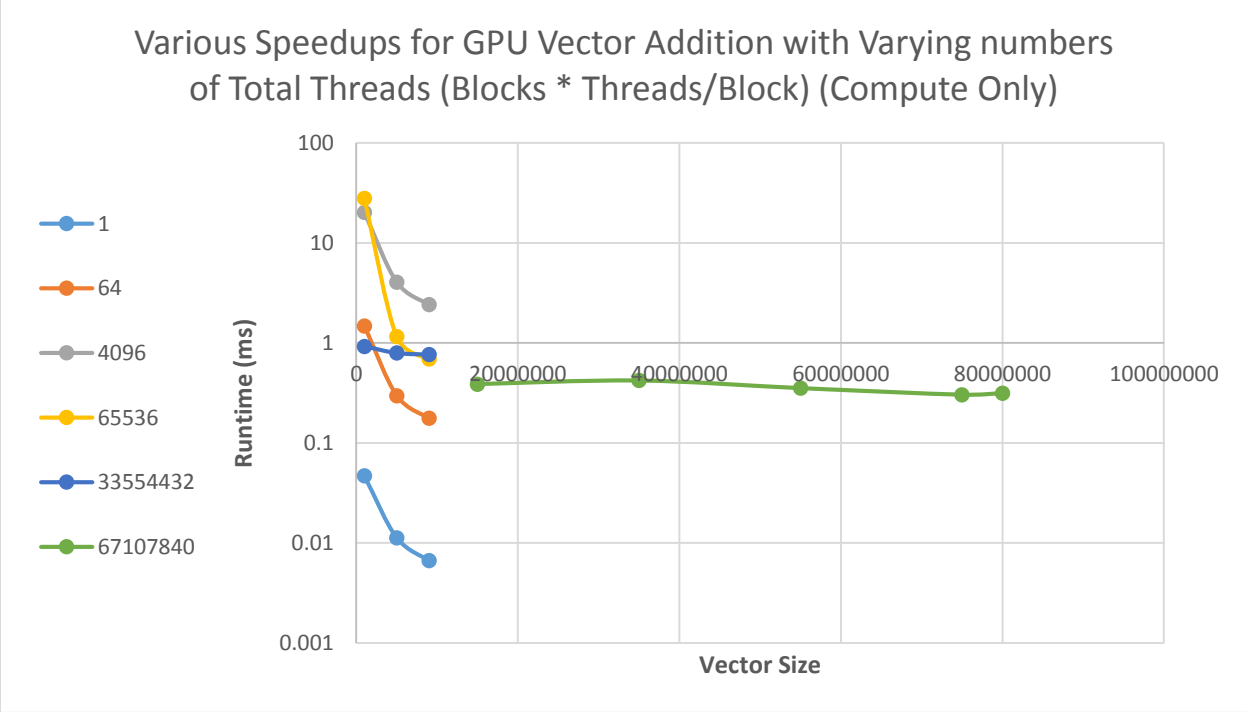

Figure 12

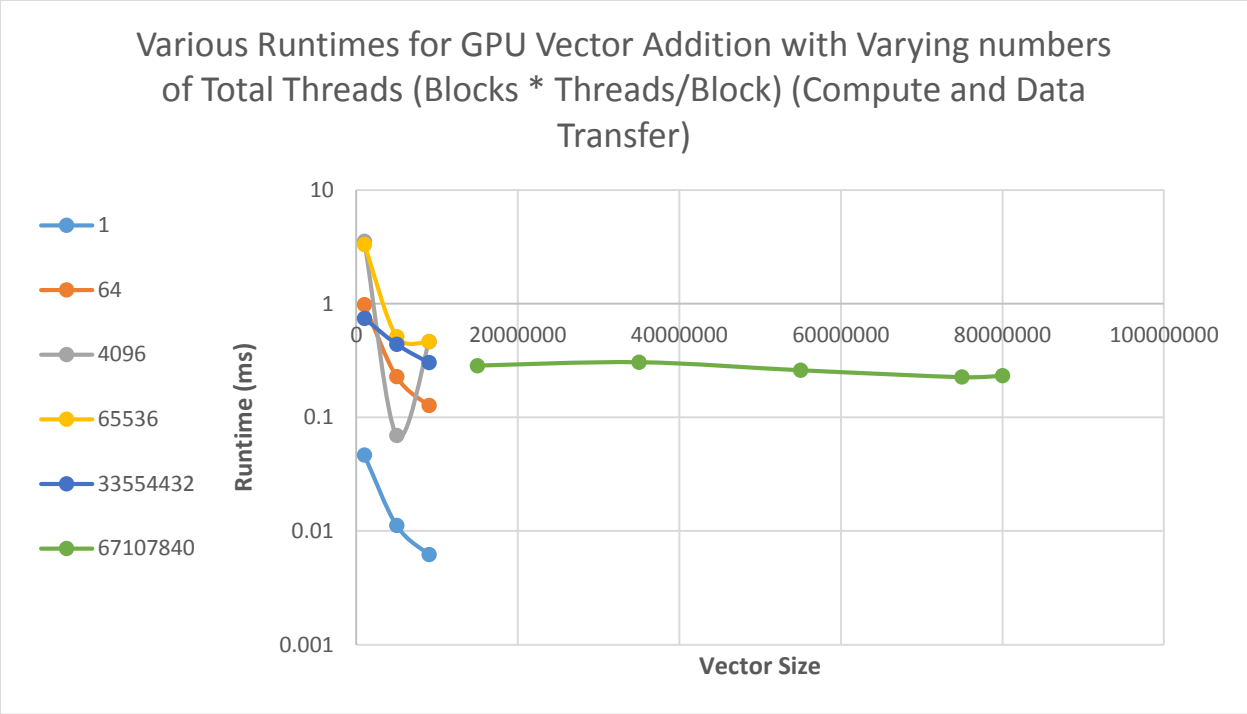

Figure 13

Discussion

Choice of Graphs
The results of test runs with a vector size of 10,000,000 were focused on because a) these were the test set that included the best speedups, and b) it was also the largest vector size that could be run reliably on my machine. There are, of course, other graphs briefly summarizing computation performance on other vector sizes, but more detailed examination was directed to the 10,000,000 size runs.

Striding vs. Non-striding
As expected, the striding vector addition algorithm was faster than the non-striding one in all cases. This largely has to do with the both the efficiency of the memory accessing of the striding algorithm (stretch of memory accessed per iteration can supply most/all threads, rather than making many memory accesses for each iteration of the algorithm) as well as the reduced complexity of the striding version of the algorithm itself.

Low Speedups
Compared to the speedups that others observed, but that is due to the lack of high performance hardware used in my experiments. Significant speedup was still observed with higher numbers of blocks/threads, still demonstrating the superiority of the GPU for throughput applications.

Further, due to having to launch multiple kernels to compute a single vector addition, all speedup is lost after a vector size of 10,000,000.

Optimal Hardware Parameters
Without any formal analysis, intuitively by looking at the speedup graphs, it appears there is a linear trend that balances number of threads per block with number of blocks. Any configuration not on this line, even ones with a near maximal number of threads and blocks results in a severely decreased level of speedup, sometimes even slowdown.

Issues

Windows TDR
A very interesting issue arose that really limited my ability to fully test the capabilities of my laptop's GPU. As it turns out, the Windows operating system has a "feature" known as **T**imeout **D**etection and **R**ecovery. This feature means that if an operation on the GPU does not finish in 2 seconds (the default value, it could be changed by making changes in the registry), Windows considers the GPU to be in an error state and restarts the GPU in order to share the GPU resources with any processes that need it to prevent starvation. Unfortunately, this results in a failure in kernel execution, and any CUDA operations that followed in my program would also fail (even when going down to smaller vectors – ALL GPU usage was taken from my program). Perhaps there is some sort of CUDA command that could "reset" the GPU and allow for error recovery, but at this time it is unknown to me.

In order to work around this TDR issue, I had to add modified kernels to my code that would process the vectors in segments that were small enough that the kernels could run in under 2 seconds. Obviously, this resulted in a hit in performance, but I was able to use this strategy to test the GPU with larger vectors in order to see it fail to allocate the memory. This work-around was not 100% effective in preventing the crash, however; it eliminated about 95% of such crashes though.

Weak GPU, Not So Weak CPU
I was able to see speedup in this exercise using my laptop's GPU, but it was weak enough that I don't believe that the GPU architecture got a "fair shot" at demonstrating its effectiveness. It probably didn't help that the CPU I used is probably comparatively a better piece of hardware (a Core i5 with decent specs for a laptop compared to a "mobile gpu"). In the future, I will be looking to gain access to a higher performing GPU so that I can see the real effects of GPU computing.