PA02: Vector Reduction
CS791v: Parallel Computing

Terence Henriod

February 13, 2015

Abstract

Vector reduction is a readily parallelizable task. The fact that it is a common task provides the motivation to carry out the parallelization. This report discusses the implementation and results of the task.

1 Introduction

Vector Reduction is a commonly performed task, and is simple enough that it begs parallelization. While the best way to do this may not be readily apparent, it essentially boils down to giving threads a share of the vector to reduce, then passing the result to a logarithmically decreasing number of threads to finish the task. In fact, it is not the actual reduction algorithm that requires the most thought and care, but rather the tailoring of the algorithm to the GPU architecture.

2 Theory

2.1 Sequential Algorithm

Vector reduction in the sequential form equates to the use of a common `for (int i = 0; i < size; i++)` style loop. I think little more discussion is required here.

2.2 Parallelization on the GPU

2.2.1 Parallel Algorithm

In the simplest case, vector reduction in parallel means that as many threads as there are elements of a vector are used to reduce 2 of the elements of the vector, and then half as many threads can carry on the process recursively, and half of those threads can recursively reduce, and so on until one thread reduces the last 2 resulting elements. In the case that there are fewer threads than elements, one might use a striding strategy to stride threads across the vector in order to reduce the vector to a size that can be handled by the previously described algorithm (I have no proof that this is either performance optimal or sub-optimal). If there are more threads than elements, then one just needs to carry out the original algorithm carefully so as to use a number of threads that is a power of two so as not to miss any vector elements in the reduction.

2.2.2 The GPU

The algorithm is relatively straightforward. The complexity comes when it is tailored to use on a GPU.

First, the GPU uses blocks of threads, so the vector must be subsetted appropriately so that each block gets a sub-vector, then the algorithm can be applied at the thread level. Second, the GPU architecture is such that not all threads can have the rapid memory access required for fast reduction (this is similar to the CPU case if the vector would not fit in cache memory, or similar). Because of this, vector segments must be duplicated in shared memory which allows faster memory access to threads at the block level (as would caching for CPU threads). Finally, as with most parallel tasks, synchronization is an unfortunate necessity. While synchronization is relatively straightforward for threads within a block, the GPU offers no method of synchronization between blocks. This problem is easily remedied however, but splitting the reduction into two separate kernel launches: one to reduce the vector into a small enough intermediate vector that can be handled by a single block, and the second one to perform the final reduction.

3 Results

The performance results of the vector reduction implementation are listed here. To provide representative data of performance of the GPU over the CPU, samples representing Vector Reduce at size 14000000 and using 256 threads per block, 63 blocks were chosen.

3.1 Information on the GPU device used

Attribute	Value
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
SharedMem/Block	49 KB

3.2 Performance Graphs

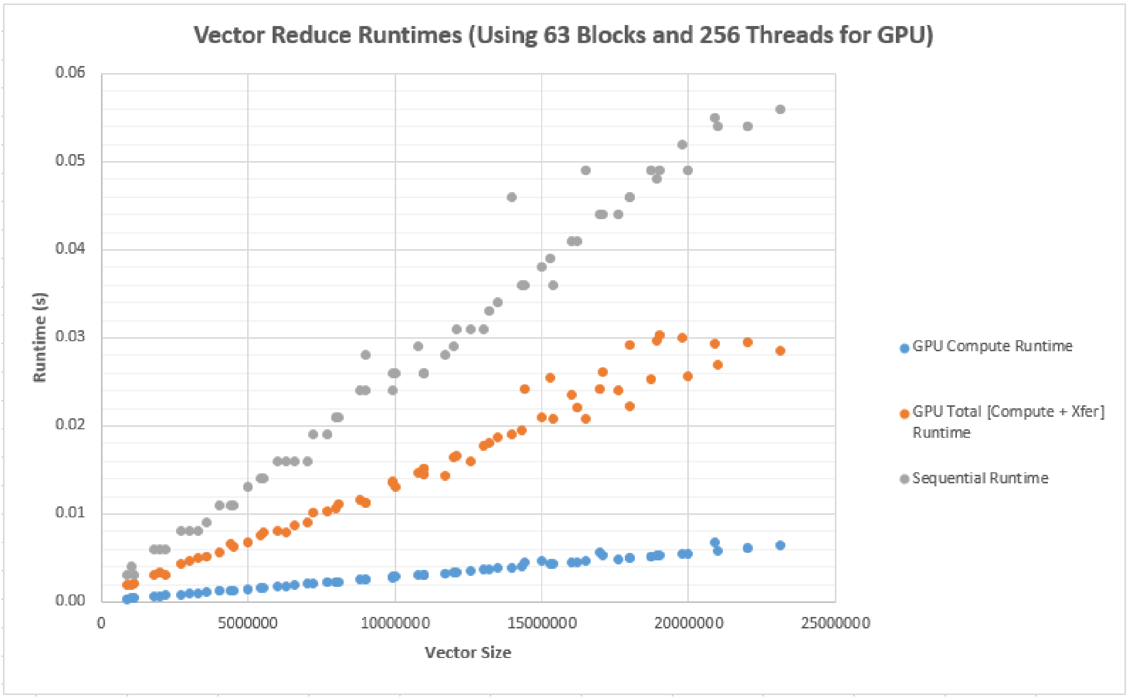

Figure 1: Runtimes for vector reduce (using a good number of GPU blocks/threads).

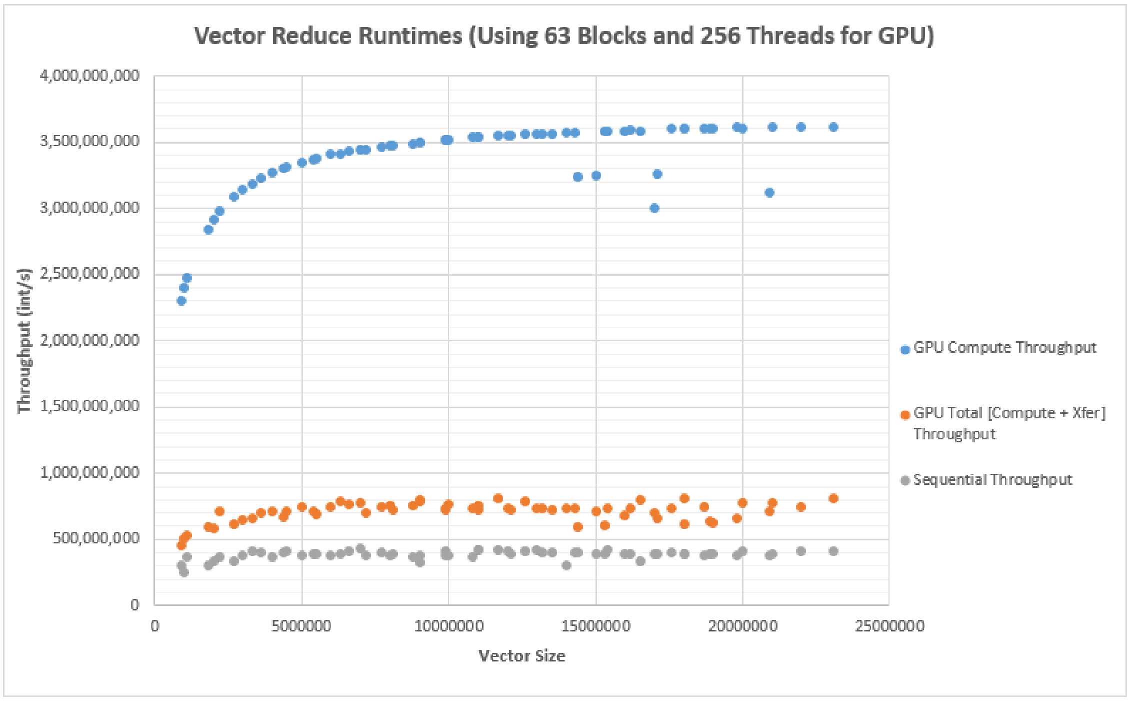

Figure 2: Runtimes for vector reduce (using a good number of GPU blocks/threads).

4 Discussion

4.1 Speedups

The speedups were lower than I would have expected, with a maximum speedup of around 11. As usual, this was likely due to the weak GPU. However, we can still see where the best speedups occurred. It appears that the best speedup occurrs using around 128 - 256 threads with around 63 - 255 blocks. Note that there is a sharp dropoff in speed once more than 1024 blocks are used due to the fact that this is when the reduction algorithm makes a significant change: intermediate kernel launches start to be used to reduce the vector enough (so that it has fewer elements than the maximum of threads per block) for the final kernel call to complete the reduction. Over using threads does seem to cause some reduction in performance, but not as significantly as overuse of blocks does.

4.2 Data Transfer

Data transfer was a small, but relatively uniform hindrance to performance. Interestingly enough, however, data transfer apears to play a smaller role in the runtime of the GPU reduction at larger vector sizes. It would be interesting to explore this further in an environment where GPU operations weren't limited to 2 seconds to see if the performance hit truly does level off.

5 Issues

5.1 Windows Takes Away the GPU

As I have stated before, Windows takes control of the GPU away from the program should a GPU operation take longer than 2 seconds. While this does limit the range of vector sizes I am allowed to try, I believe I still acquire meaningful runs that demonstrate the necessary penomena.

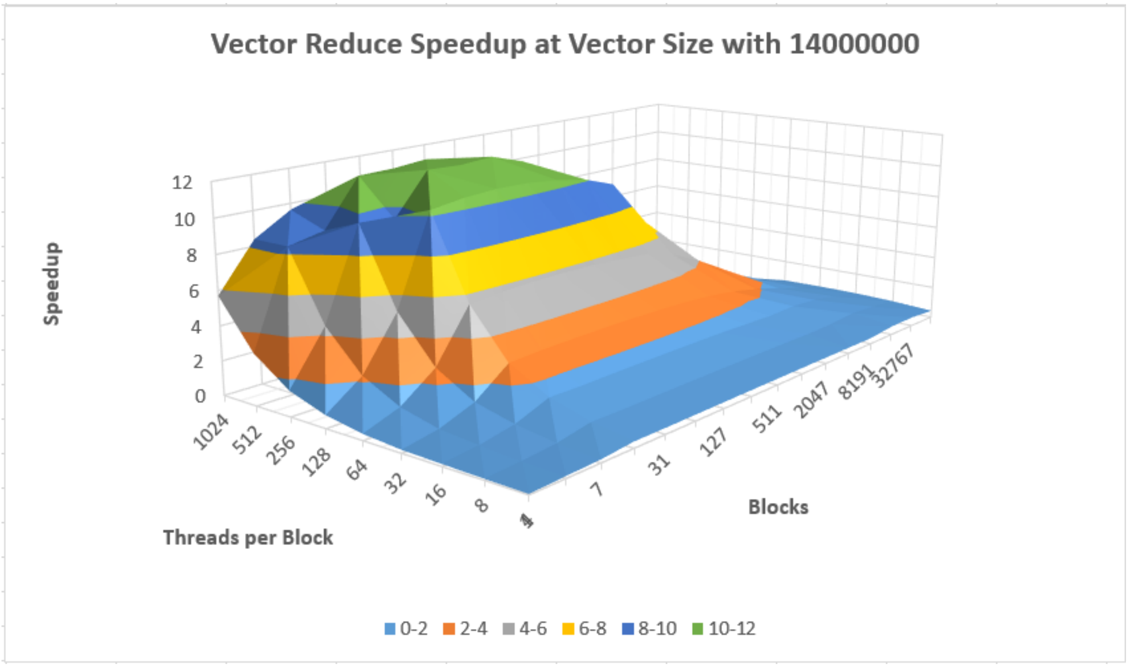

Figure 3: A comparison of speedup for the computation portion of vector reduce across differing numbers of block and threads using a good vector size.

5.2 Weak GPU

The weakness of the GPU makes it hard to demonstrate impressive performance.

5.3 Ill Defined Assignment

Not having specific guidelines made it somewhat difficult to complete the assignment on an appropriate timeline.

5.4 Personal Folly

I wanted to attempt using a personal algorithm where I divided the vector into shares for each block rather than simply striding the threads across the entire array. This became difficult to debug and likely would have reduced performance, so this strategy was abandoned (although much later than it should have been).

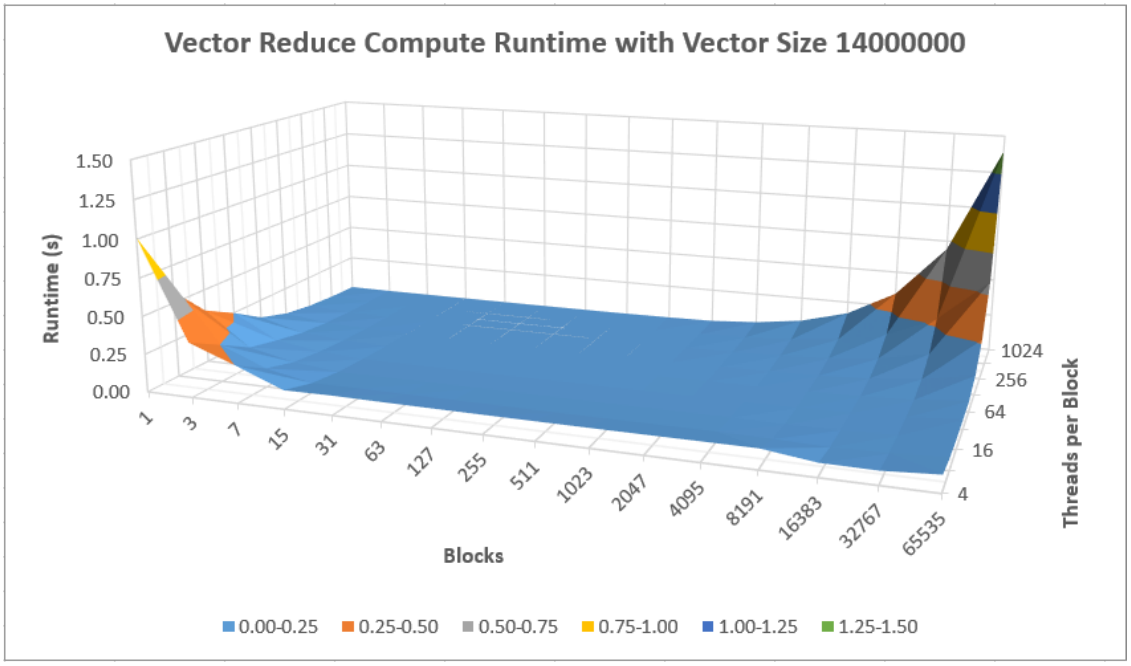

Figure 4: A comparison of runtime for the computation portion of vector reduce across differing numbers of block and threads using a good vector size.

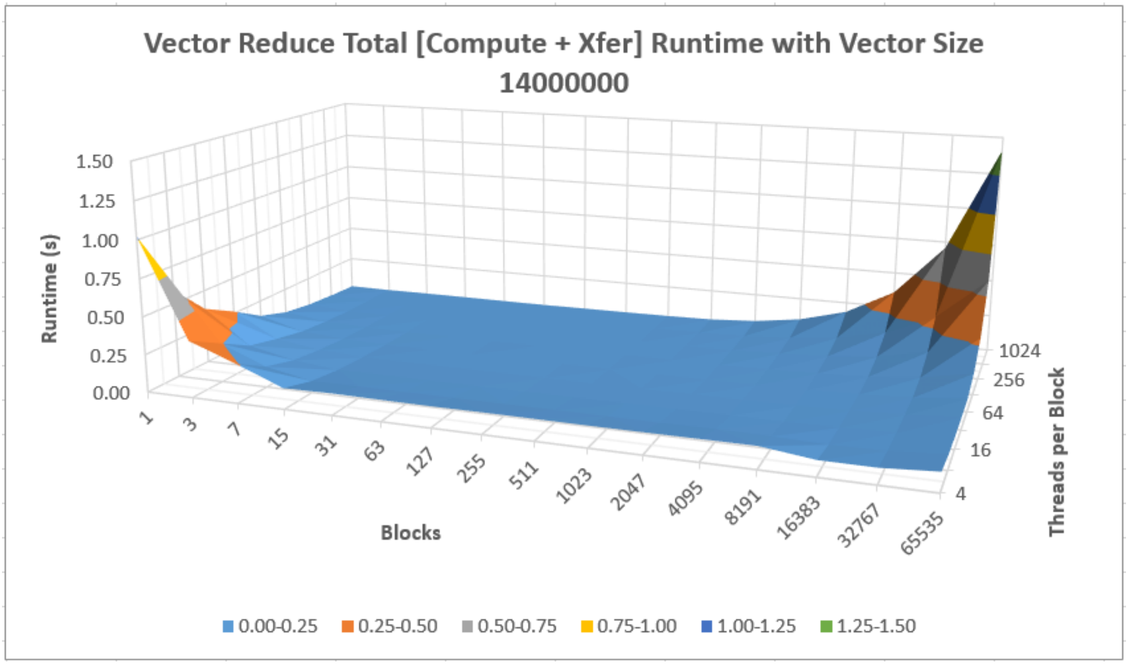

Figure 5: A comparison of runtime for the computation and data transfer of vector reduce across differing numbers of block and threads using a good vector size.

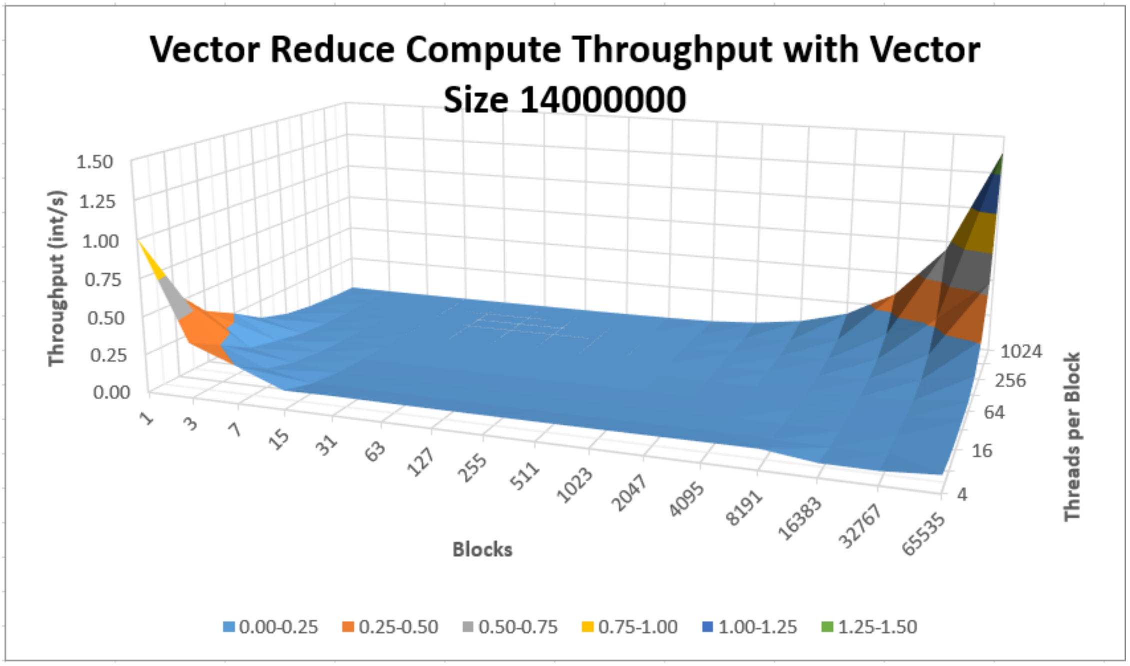

Figure 6: A comparison of throughput for the computation portion of vector reduce across differing numbers of block and threads using a good vector size.

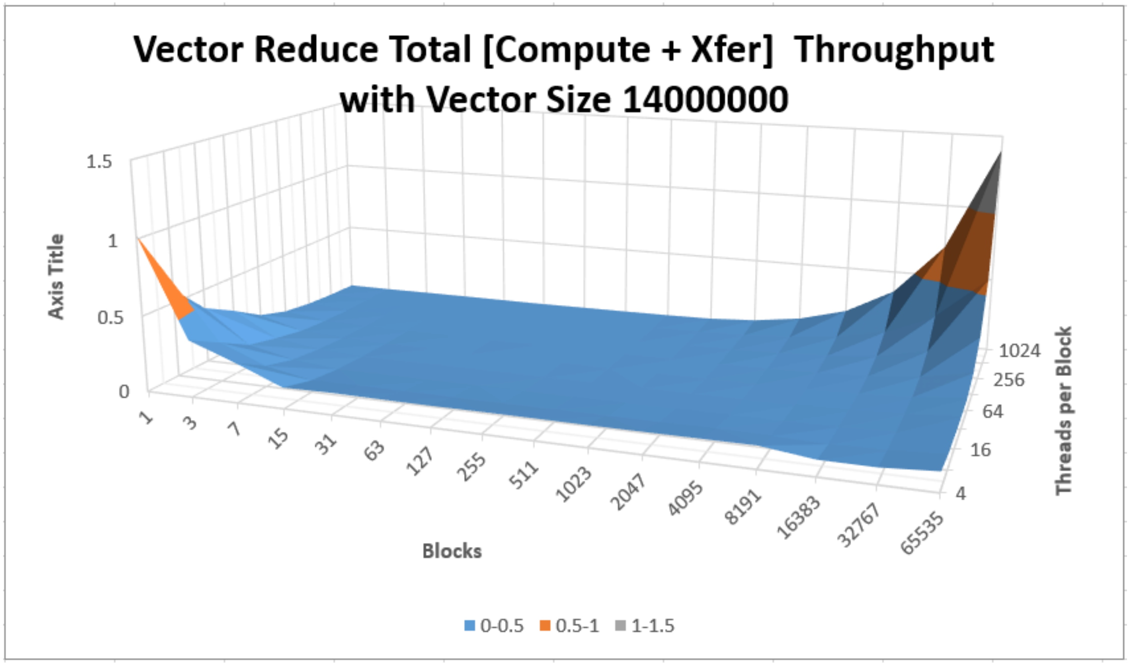

Figure 7: A comparison of throughput for the computation and data transfer of vector reduce across differing numbers of block and threads using a good vector size.