Matthew VanCompernolle
CS791v 2/4/15

PA2 Report – Vector Reduction

Introduction

The purpose of the project is to implement a sequential and a parallel version of vector reduction using the addition operation and compare the two in terms of performance. The sequential implementation is designed to run on a single core of a CPU, and the parallel version is designed to run on modern multicore Nvidia GPUs. The program implements vector reduction on the GPU using CUDA, which is Nvidia's parallel computing platform. The parallel implementation reduces the vector by breaking the vector down into smaller chunks on blocks that perform additions simultaneously using multiple threads in each block until each block only has a single partial sum. The partial sums of each block are then summed together to produce the total sum of every element in the vector. The goal of the parallel implementation is to yield the highest performance possible on the GPU by successfully created balanced distribution of workload across threads and blocks, utilizing shared memory on each block, and synchronizing information properly.

Theory

Vector Reduction

Vector reduction is the process of applying an operation to every element in a vector to reduce it to a single value. The operation used in the project is addition. The result of applying vector addition on a vector is that it is reduced from its original size to a single element, which is the sum of each of the elements.

Speedup Factor

For the project it is necessary to determine how much faster a parallel implementation of vector reduction 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.

Throughput

Throughput is the rate in which components can be processed. In computer science applications, throughput is often measured in Gigabits/second (Gbps). It is often useful to measure and visualize both throughput and run time, as throughput can reveal important trends in the data that may have gone undetected when only analyzing run time. Throughput can be defined as:

$$Throughput = \frac{amount\ of\ data}{execution\ time}$$

Implementation

Sequential Implementation

The sequential implementation of vector reduction was executed on an Intel i7 4970k running at 4.0 GHz with a maximum turbo frequency of 4.4 GHz. The implementation was executed using three 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 8,388,608 million, 67,108,864 million, and 268,435,456 million which are all powers of two. Even at these sizes computation times were extremely quick, with each of the 3 sizes taking less than half of a second. This is due to the fact that vector reduction only has a time complexity of O(n), which can be seen in Figure 1, with low computational intensity. The run times produced by the sequential implementation for the three vector sizes are shown in Figure 1.

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 vector being reduced to about three hundred million elements. The kernel for the parallel implementation first has each thread copy data from global memory to shared memory on each block. The shared memory is initialized to be the size of the number of threads in a block so that each thread gets its own index. To improve performance each thread attempt to access two elements from the vector stored in global memory at a time in an attempt to sum the two elements before placing them into shared memory. In the case the vector in global memory is too large to be copied into shared memory across all of the blocks, each thread will stride to further memory locations in global memory to sum into their shared memory location, effectively reducing the vector to a size that will fit no matter what. After the entire vector is copied into shared memory (after possibly being partially reduced already) the threads in a block synchronize to ensure that all of the data is initialized for the next step. The threads then sum vector elements on the left side of shared memory with the vector elements on the right side of shared memory, effectively reducing the vector to half the size. This is repeated with synchronization in-between until the number of partial sums of the vector is small enough that they can be reduced by a warp. A thread warp consisting of 32 threads then reduces a chunk of 64 partial sums repeatedly. Threads within a warp repeat the reduction process without synchronizing because threads in a warp execute at the same time, computing more efficiently. The overall result is that each block is left with a single partial sum in the first index of shared memory. The partial sums in each block are then placed in a results array and sent back to the CPU, where the CPU then performs a small final reduction finally yielding the overall sum. The run time for the GPU portion of the reduction and the CPU portion of the reduction were compared on many different runs with different numbers of threads, blocks, and vector sizes. It was determine that the final CPU reduction was negligible time wise in all cases, which is why it was used for the final reduction instead of passing the partial reduction across blocks back to the GPU for further reduction.

The parallel implementation was executed using the same 3 vector sizes as the sequential version (8,388,608 million, 67,108,864 million, and 268,435,456 million). The program that implements the parallel version was designed to be able to run using many different variations for the number of blocks and the number of threads ran on the GPU. Figure 2 shows the run times of parallel implementation of vector reduction on a vector size of 67,108,864 elements using a variety of different combinations for the number of blocks and the number of threads. The number of blocks begin at 8 and doubles at each step until the maximum of 65535 blocks is reached, and the number of threads begins at 8 and doubles at each step until the maximum of 1024 is reached. Figure 1 shows the best run time produced from various block and thread combinations for the parallel version on all three vector sizes.

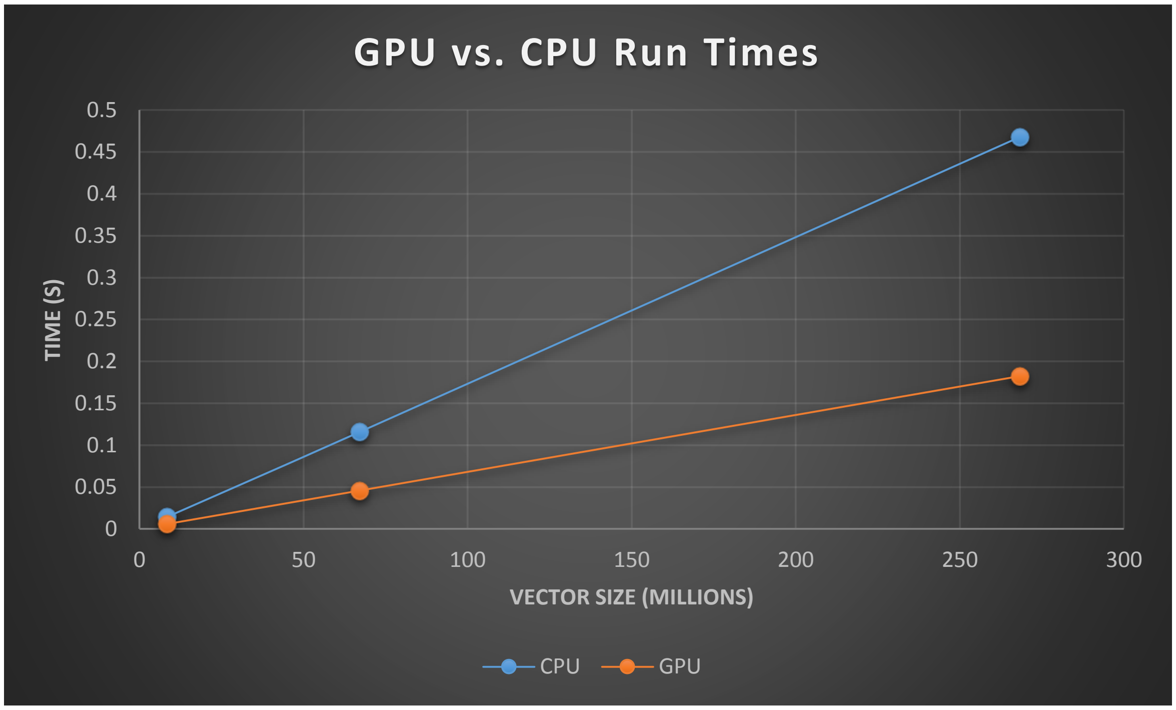

Figure 1: This graph shows the run times for the sequential implementation on CPU and the parallel implementation on the GPU for vector sizes of 8,388,608 million, 67,108,864 million, and 268,435,456 million.

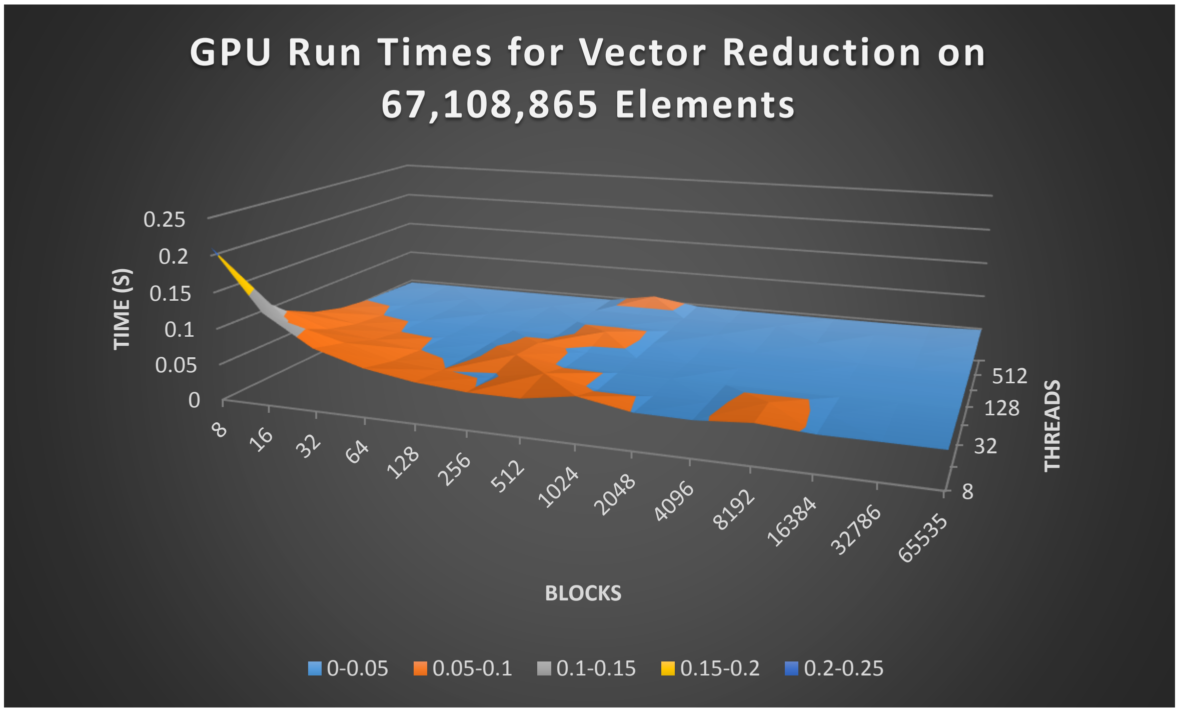

Figure 2: This graph shows the how the run times for the parallel version changes based on the number of blocks and threads used. Run time is consistent in most variations except for the runs that tended to use a very low number of threads, blocks, or both.

Analysis

The execution times for the parallel version were noticeably faster than those of the sequential version, as can be seen in Figure 1 and Figure 3. Overall, the parallel implementation on the GPU achieved a speedup factor of about two and half when compared against the sequential implementation, as seen in Figure 4. The GPU was able to perform faster due to its ability to perform many partial sums simultaneously across the many blocks and threads within those blocks that it executes on. It is important to note that all run time measurements took into account the transfer time of any memory across the PCI-E bus to and from the GPU. The transfer time causes significant overhead. If run time were graphed without considering transfer time, the speedup factor would be much higher, but would not realistically depict the cost of using a GPU for parallel computation for real world applications.

Another important conclusion that can be drawn from the data is that number of blocks and the number of threads did not have a large effect on parallel execution times. Figure 2 shows many different tests ran on with different numbers of block and threads, and the majority of test produced very similar results. Only in the cases where the number of threads, the number of blocks, or the number of both were low was there a noticeable negative change on performance. Fewer threads or blocks means less parallelism, which is expected to cause slow down. Figures 3, 4, and 5 are all functions of the number of threads used in the parallel version, and each of the three graphs are flat with respect to the number of threads except for at very low numbers indicating that they played very little role in the results. Based on the mostly flat nature of Figure 2, it can also be assumed that if the same data was a function of the number of blocks in the algorithm that the results would change relatively little as the number of blocks change. Figure 5 also shows that throughput of the algorithm is approximately 45 Gbps regardless of the number of threads, vector size when size is relatively large, and the number blocks because the number of blocks affects the algorithm in a very similar way that the number of threads does.

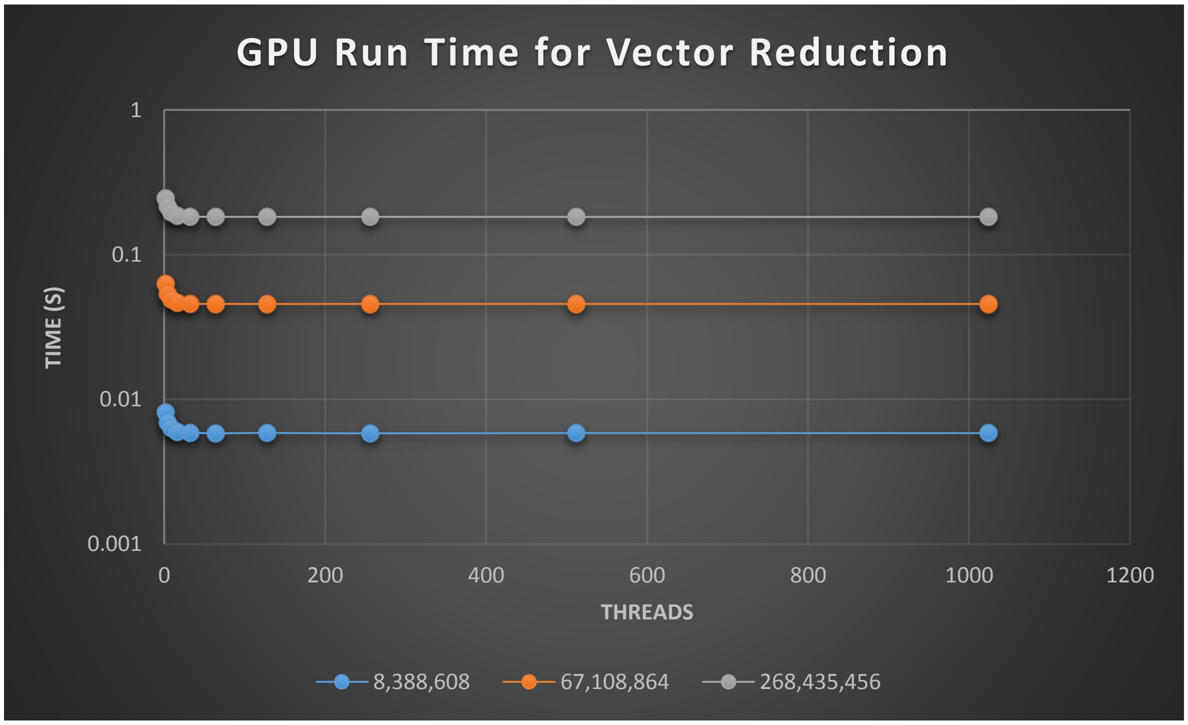

Figure 3: This graph shows the run times for the parallel implementation on the GPU for the three different vector sizes and how they scale with the number of threads used per block. The data depicts that after a certain minimum threshold the number of threads has little effect on the performance.

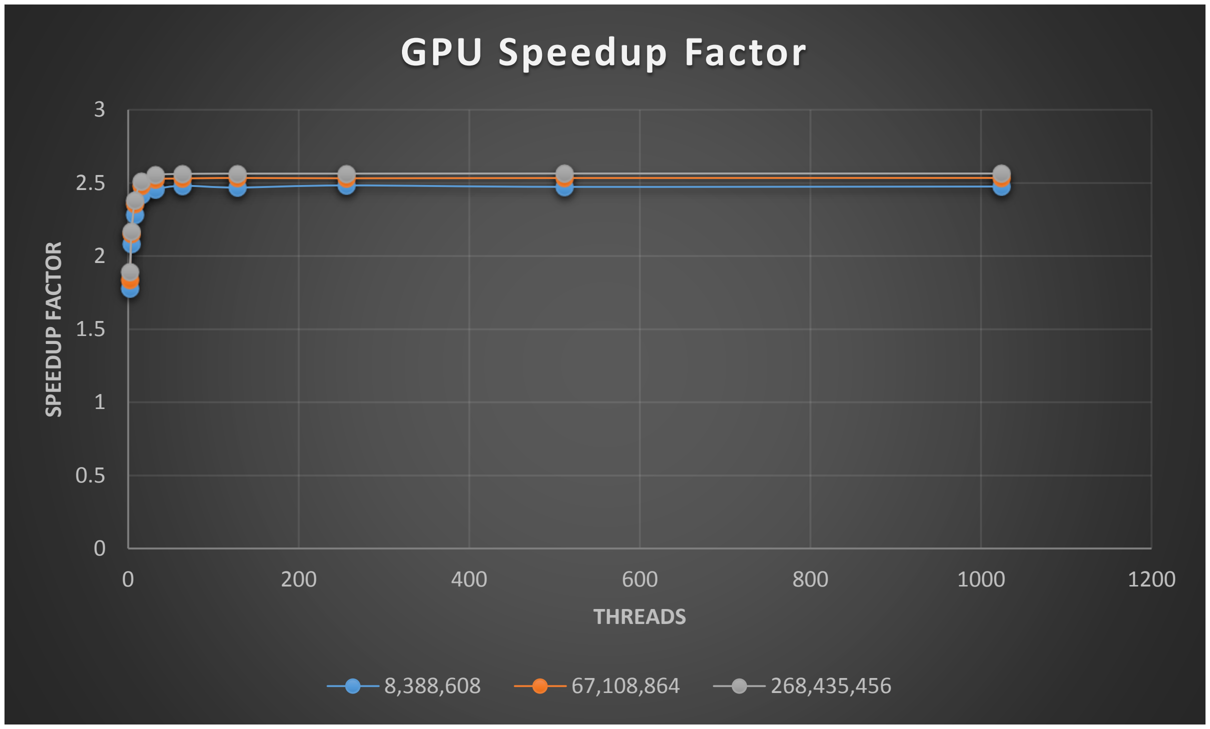

Figure 4: This graph shows the speedup factor for the parallel implementation on the GPU for the three different vector sizes and how they scale with the number of threads used per block. The data depicts that the parallel version tends to achieve 2.5 times speedup over the sequential version.

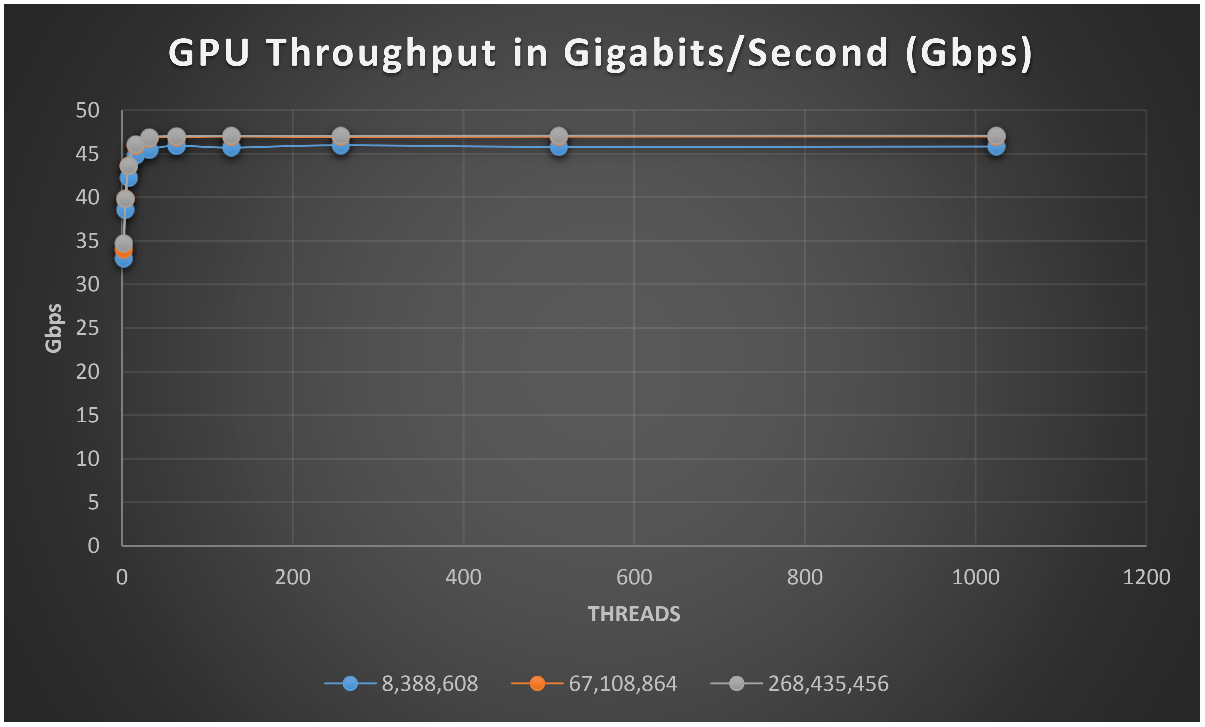

Figure 5: This graph shows that the parallel implementation achieves a throughput of about 45 Gbps in almost every thread configuration, except for those with very few threads per block.

Block and Thread Sizes

Issues

The biggest issue I had was trying to figure out how to effectively utilize warps, blocks, and threads to efficiently perform a correct version of vector reduction. It was difficult to wrap my head around what index a thread should be adding with and where it should be storing the summation in shared memory. I also was and still am a little confused at what the difference between a warp wise reduction and block wise reduction is. I understand that a warp wise reduction utilizes up to 32 grouped threads to perform a partial reduction, but in doing so is it not also technically partially reducing the block it is running in at the same time? The two seem to have significant overlap in my mind which makes it difficult for me to understand how to code the two as completely distinct steps.