Spring 2017

Efficient Implementation of Reductions on GPU Architectures

Stephen W. Timcheck
The University of Akron, swt5@zips.uakron.edu

Efficient Implementation of Reductions on GPU Architectures
Department of Computer Science Honors Project
Stephen Timcheck

Abstract

With serial, or sequential, computational operations' growth rate slowing over the past few years, parallel computing has become paramount to achieve speedup. In particular, GPUs (Graphics Processing Units) can be used to program parallel applications using a SIMD (Single Instruction Multiple Data) architecture. We studied SIMD applications constructed using the NVIDIA CUDA language and MERCATOR (Mapping EnumeRATOR for CUDA), a framework developed for streaming dataflow applications on the GPU. A type of operation commonly performed by streaming applications is reduction, a function that performs some associative operation on multiple data points such as summing a list of numbers (additive operator, +). By exploring numerous SIMD implementations, we investigated the influence of various factors on the performance of reductions and concurrent reductions performed over multiple tagged data streams. Through our testing, we determined that the type of working memory had the greatest impact on block-wide reduction performance. Using registers as much as possible provided the greatest improvement. We also found that the CUB library provided performance similar to our fastest implementation. We then explored segmented reductions and their optimizations based on our previous findings. The results were similar for segmented reductions: using registers as much as possible provided the greatest performance.

Acknowledgements

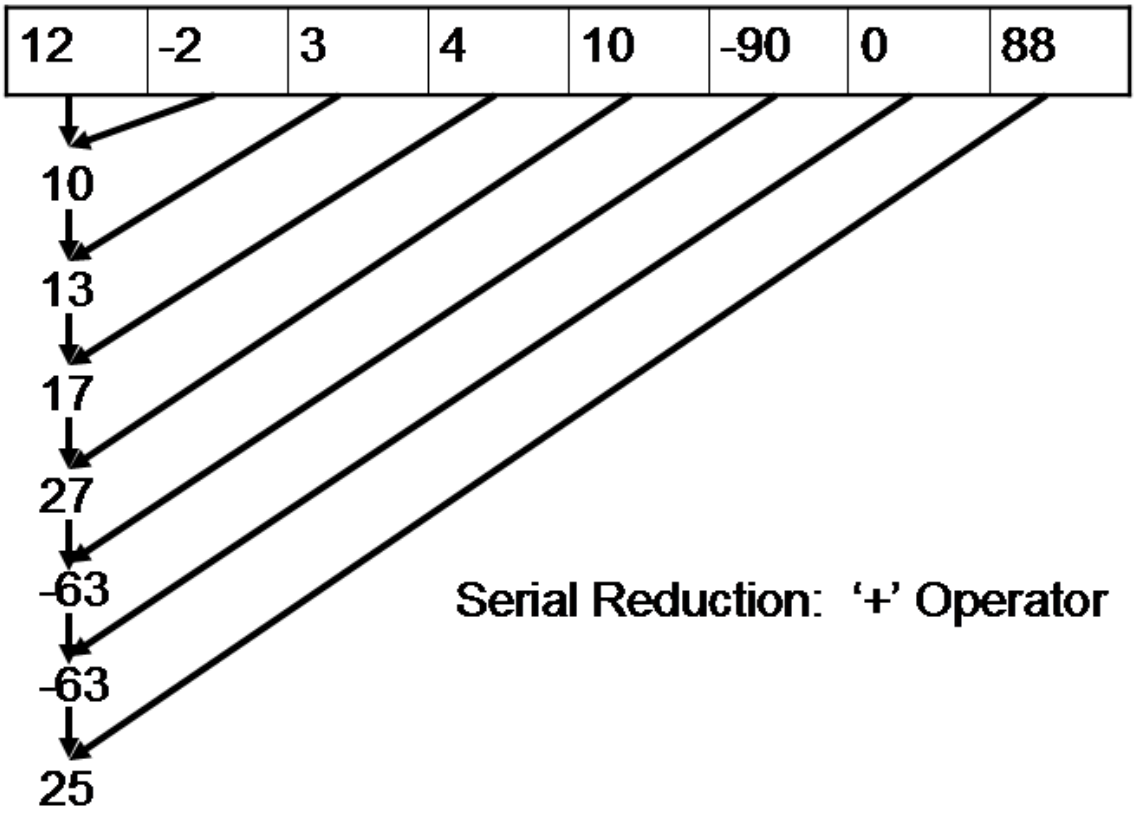
I would like to give special thanks to Dr. Jeremy Buhler of Washington University in St. Louis Computer Science and Engineering department for advising me on this research, as well as Dr. Timothy O'Neil for sponsoring the project.

Chapter 1

Introduction

A commonly performed operation on large sets of data is a reduction. A reduction is some function which performs an associative operation on a set of elements to produce an aggregate result. Reductions can be useful for finding summarizing various components of an application and its output, such as in the MapReduce algorithm [1]. In particular, we are looking at reduction operations that are associative and commutative.

In traditional sequential programming, a summation of elements in an array can easily be achieved by a simple for loop iterating over all the elements in the array. A reduction producing a summation would look as such with sequential programming:



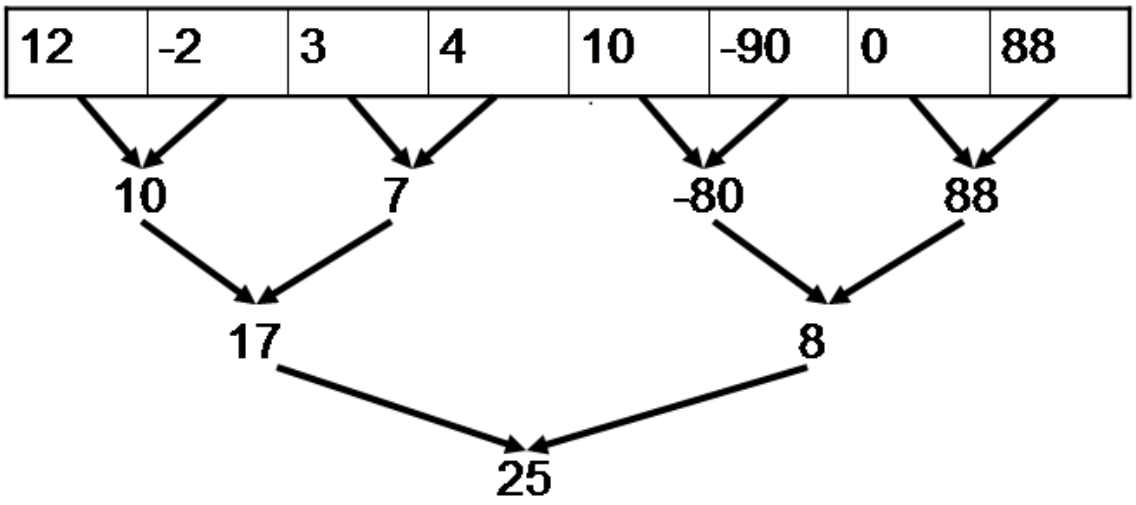
However, due to increasing size of data sets that need such aggregate results and the lack of performance gains in single-threaded applications, scalable parallel approaches are becoming more common to address this problem. In particular, the SIMD (Single Instruction Multiple Data) architectures that are utilized by GPUs can perform these operations faster. In this paper, we will explore various efficient SIMD parallel implementations of reduction operations on GPUs.

Background - CUDA

Given the nature of how reductions are performed, many-core processors such as GPUs can execute these operations. We worked with the CUDA architecture, a set of libraries designed for general purpose computing on NVIDIA GPUs. GPUs have SIMD architectures;

that is a single instruction is run on multiple points of data at once. To add two arrays of integers element-wise on a GPU, one instruction can be run to add the elements where every thread adds the two elements that it is assigned to. This provides a result not requiring a sequential "for" loop iterating over all elements like on most CPUs.

For example, an array of integers is produced from some source and the user would like to have the summation of these results, such as the problem described in the introduction. The summation of the elements would be a reduction using the '+' operator on the array. A tree based SIMD execution of the reduction would be similar to the following figure:



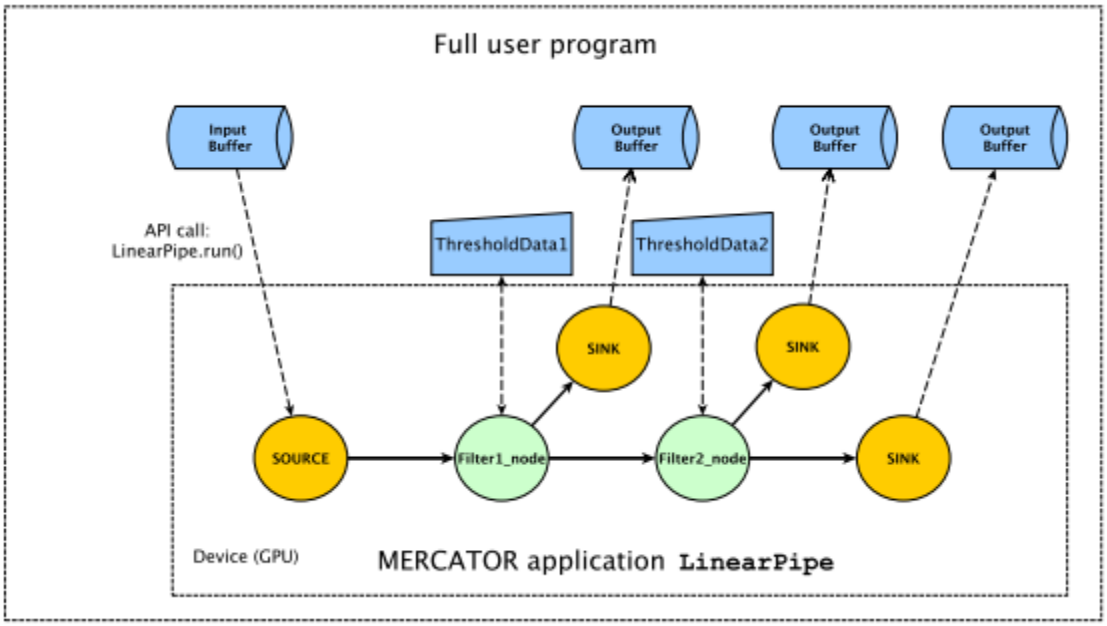
Tree-Based Reduction: '+' Operator

The GPU conceptually runs one SIMD operation per level of the tree. This reduces the number of operations to the height of the tree, O(log(n)), rather than the total number of elements, O(n). For example, the sequential method in the introduction takes 8 operations to complete whereas the tree based SIMD reduction needs only 3 operations. However, CUDA systems have many other considerations determining how well these reductions perform which we will explore in Chapter 2.

Background - MERCATOR

In addition to making reductions efficient, we also wanted our implementations to be accessible to users without extensive parallel programming backgrounds. To do this, we targeted the MERCATOR (Mapping EnumeRATOR for CUDA) system for implementation of our reductions.

MERCATOR, developed by Stephen Cole and Dr. Jeremy Buhler, is a library which aims to make data-flow applications easier to implement on NVIDIA GPUs [4]. MERCATOR supports a modular approach for implementing CUDA kernels that allows for dynamically varying numbers of inputs and outputs per module. With MERCATOR one can easily construct various applications through the use of its modular design. MERCATOR handles all of the memory management and module scheduling for every module in the pipeline of the application. The following figure shows a schematic of a simple MERCATOR application pipeline [4]:



An input buffer is received by the source module and then is passed to the rest of the modules of the application. When certain output is obtained from a module it is then passed to a sink module, which returns the output back to the user. In the case of reductions, we would like to support a *reduction sink* module that takes input and returns only the aggregate to the user.

However, the modularity of MERCATOR applications provide design constraints. First, most reductions are designed and tested around device-wide operations, that is a reduction performed across the entire GPU such as those tested by NVIDIA [5]. Our design must provide the most performance for block-wide reductions instead as this will be the most commonly applicable case in the MERCATOR library. That is, our tests will be testing the performance of reductions within each block individually, not across all blocks.

The GPU achieves its results through running some number of threads, up to 1024, inside of a block. These threads run each instruction concurrently, but work with different data points, hence being a SIMD architecture. Many of these blocks can be run at once on a GPU as well providing expansion to over 1024 threads. MERCATOR runs the application independently in each block as opposed to across the entire device, so the processors or blocks do not have to communicate between each other during execution.

Contributions

In this work we **(1) tested various reduction implementations**, and **(2) tested segmented reductions** both for block-wide reductions. In the following chapters, we will discuss the motivation for different design decisions, the impact certain design decisions have on performance, and an introduction to segmented reductions as well as their performance. Chapter 2 contains information about reductions and optimizations. Chapter 3 explains segmented reductions and the pertinence of the optimizations from Chapter 2. Chapter 4 reviews concluding remarks and future works.

Chapter 2

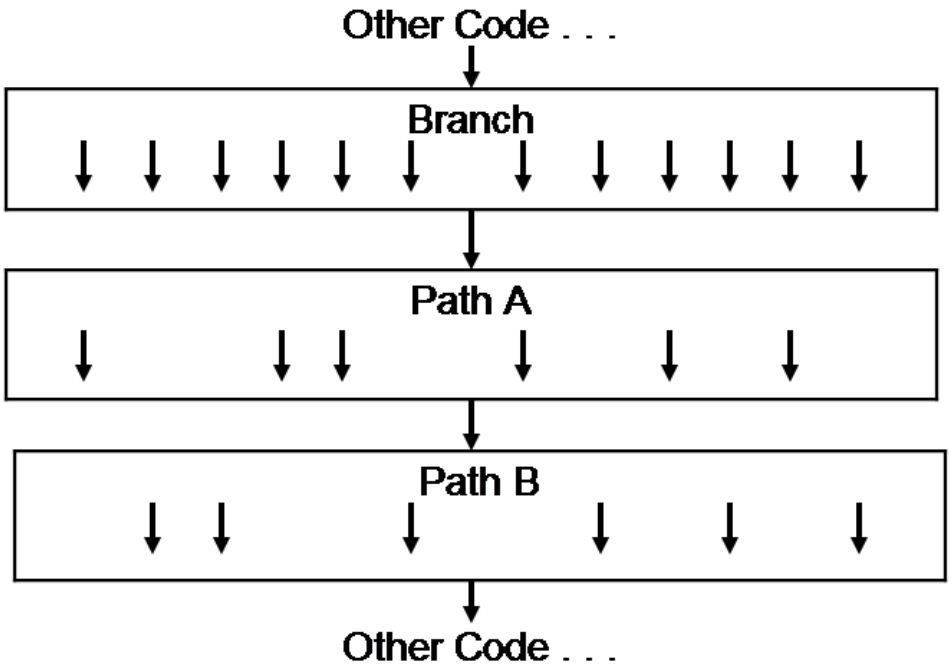
Motivation

In this chapter, we will discuss first the candidate ideas for performance improvement of SIMD reductions within a single GPU block, or block-wide reduction, as well as reasons why they were chosen. Next, we will talk about the various implementations and what improvements they made. Finally, we will review the performance of all the implementations to determine which implementation should be used for block-wide reductions. We seek to determine **(1) Factors that influence efficiency of reductions**, and **(2) The most efficient way to perform a reduction** on the GPU.

Factors Impacting Execution Efficiency on GPUs

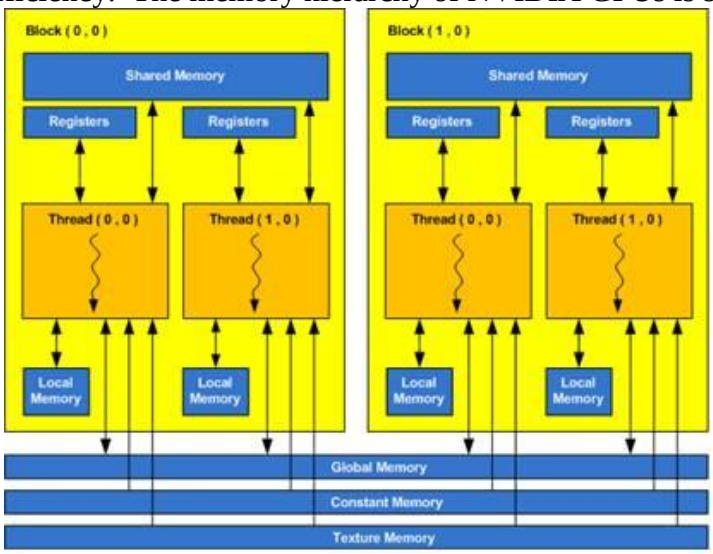
There are many factors that can hinder the speed of a CUDA application. These factors include divergent branching, operational memory storage type, and coalesced memory access.

Divergent branching occurs with control logic in a CUDA application. CUDA allows for up to 32 threads run simultaneously, otherwise known as a warp [3]. Should more threads need to be executed, NVIDIA GPUs will utilize a traditional preemptive scheduling method to time-share the processor among the warps. If there is a conditional statement within the current warp that leads to two or more separate execution paths, then branch divergence has occurred. This means that the divergence serializes some of the threads in the warp, reducing the impact of parallelization. In these cases, the execution of each path is split into separate warps, with each thread not in the executing branch remaining idle as shown below:



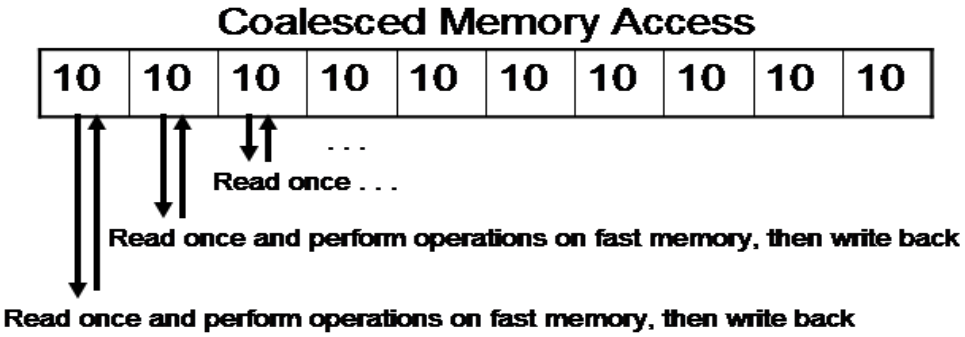
Divergence can be problematic for program efficiency as only a portion of the threads in a warp will be executing useful code. To determine if branch divergence was important to various reductions' efficiency, we tested algorithms that branched as well as those that did not.

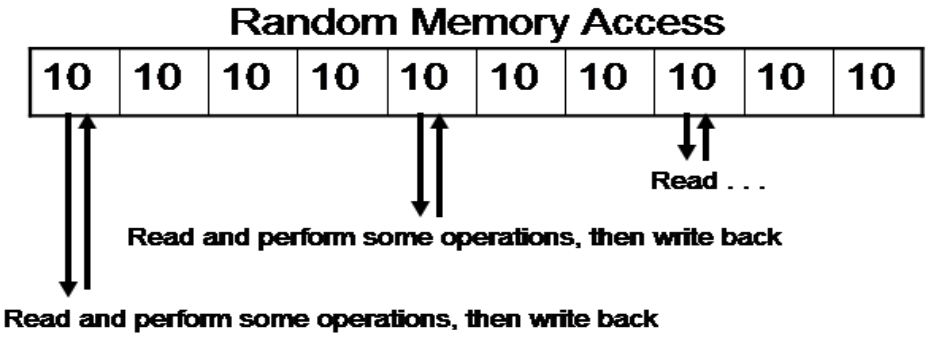
Different types of memory storage provide various benefits either to the complexity of algorithms or their efficiency. The memory hierarchy of NVIDIA GPUs is shown below [6]:



There are 3 places to store memory: global, shared, and register. Global memory is the slowest to access, but provides the easiest storage method because of its large size, usually more than 4GB on modern GPUs. Shared memory is a small pool of faster memory, around 48KB, associated with a single multiprocessor on the GPU. Registers are the fastest memory type and are available to each thread exclusively, but are even smaller than shared memory, in the number of small hundreds. Depending on which memory type one uses, there are repercussions to both performance and algorithmic complexity.

Coalesced memory access reduces the number of global or shared memory accesses by the GPU to perform operations on data. For example, given an array of 100 integers a processor can cache 10 integers into significantly faster memory at a time as shown below:





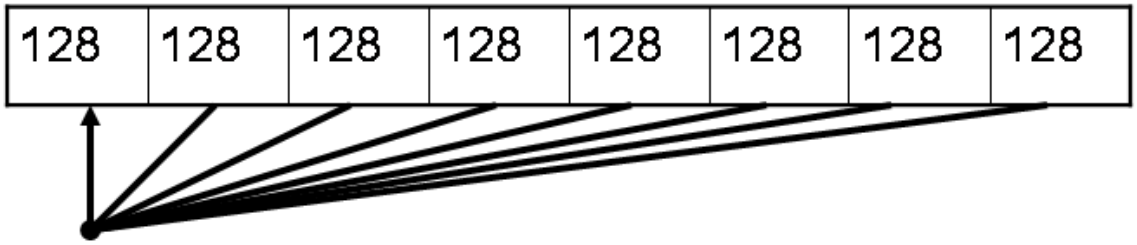
The idea of coalescing memory accesses is to design the algorithm performing a specific operation on a set of data to work on elements that are near each other in memory, thus reducing the number of global memory accesses. By processing 10 adjacent integers in adjacent threads, the processor can access all 10 integers with only one global memory read operation. This same access could have required many of these operations under random access. We wanted to test the impact of coalesced access for reductions on NVIDIA GPUs.

The general purpose of testing the reductions in a non-irregular environment is to determine the broad performance considerations in regards to reduction on GPUs as well as the ease of coding each implementation. As stated previously, there are three key testing points we hypothesized would provide the greatest performance discrepancies: **(1) divergent branching**, **(2) coalesced memory access**, and **(3) working memory type**. In addition, we had to take into account that we wanted the best performance on block-wide reductions, not device-wide reductions, thus making certain performance optimizations of other implementations potentially inert.

Our hypothesis is that all three of these factors will have significant performance impacts on reductions using CUDA, yet we do not know which will be the most important.

Methods

To investigate the impact of different architectural factors on performance, we constructed and benchmarked versions of reduction that address different performance concerns in CUDA. Each test was performed on a GTX 980Ti using CUDA 8.0 and a Unix operating system. To keep our tests between implementations consistent, we decided to maintain the total number of elements that would be reduced to approximately 4 million integers. A reduction was always performed on a set of 1024 elements regardless of the number of threads that were used in the current block. The reductions tested were block-wide reductions of 1024 elements each, not device-wide reductions constructing an aggregate value across the entire array of values. Thus, if the thread count was 128, sets of 128 elements would be reduced, and then an atomic add operation would be performed on those reductions to determine the result:



Atomic Operations Reducing all 128 element sub-results to position 0.

The test bed for each kernel was similar to the following:

```
__global__ void kernel_red1(int *d_in,
                            double *d_time) {

     //Begin trials (Number of passes needed to cover 4M ints)
     for(unsigned int j = 0; j < N_TRIALS; ++j) {
          //Pointer to initial element in block's data set
```

```
            int *d_org = d_base + j * ELT_PER_TRIAL;

            //Number of loops required for current trial given
            //thread count
            for(unsigned int h = 0;
                h < ELT_PER_TRIAL / THREADS_PER_BLOCK;
                ++h) {

                 ////////////////////////////////
                 //Perform reduction here . . .
                 ////////////////////////////////

                 //If # elements to reduce > # threads,
                 //reduce sub-reduction to index 0.
                 if(threadIdx.x == 0 && h > 0) {
                      atomicAdd(d_org, d_result[0]);
                 }
            }
      }
}
```

All the reduction kernels tested utilized an input array of elements and an array to store the timing data, integer array d_in and double array d_time respectively. Each kernel first initializes the base address for its set of reductions to d_base. Next, the kernel performs a user-defined number of trials on a user-defined number of elements per trial and threads per block of reductions. After the reductions are performed, should the number of threads in a block be less than the number of elements per trial, atomic adds are performed to add together the sub-aggregates computed for that block. It should be noted that the d_in array is used for input as well as output, which changes some reduction algorithms slightly and will be noted should it cause changes in the code.

The thread counts used in testing were powers of 2 ranging from 32 to 1024. We ran the entire benchmark concurrently on every multiprocessor by using 22 blocks per trial. We tested our implementations with 50 trials, each reducing a non-overlapping set of 1024 elements in each block, which ensured no reuse of data. Performance was measured in average number of GPU cycles to complete the task.

Many of the performance optimization ideas were gathered from [5] as well as newer memory documentation from NVIDIA [2].

Testing

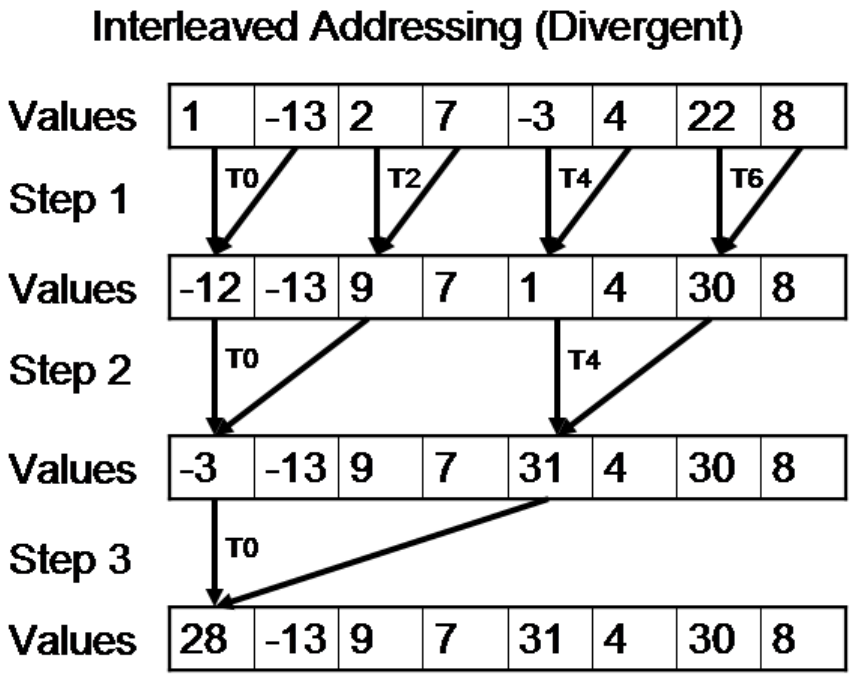
Naive Approach

The first attempt at parallel reductions was a naive approach:

```
1     for(unsigned int h = 0;
2         h < ELT_PER_TRIAL / THREADS_PER_BLOCK;
3         ++h) {
4
```

```
5          //Base for current trial
6          int *d = d_base + j * ELT_PER_TRIAL +
7                    h * THREADS_PER_BLOCK;
8
9          for(unsigned int i = 1; i < THREADS_PER_BLOCK;
10              i <<= 1) {
11              unsigned int idx = threadIdx.x;
12
13              if ((idx & ((i << 1) - 1)) == 0)
14                   d[idx] += d[idx + i];
15
16         __syncthreads();
17         }
18
19         //Finalize result if # elements > # threads
20         if(threadIdx.x == 0 && h > 0) {
21              atomicAdd(d_org, d[0]);
22         }
```

This approach utilizes interleaved addressing as well as a divergent branch and global memory to perform all of the reductions. Here, d is the input array of integers. Interleaved addressing, refers to addressing that causes active threads to access memory that grows further and further apart. This function provides a basic implementation of a tree-like reduction. The process is illustrated below for a reduction over eight elements using eight threads where Values are the current values in d and T0-8 are the threads:



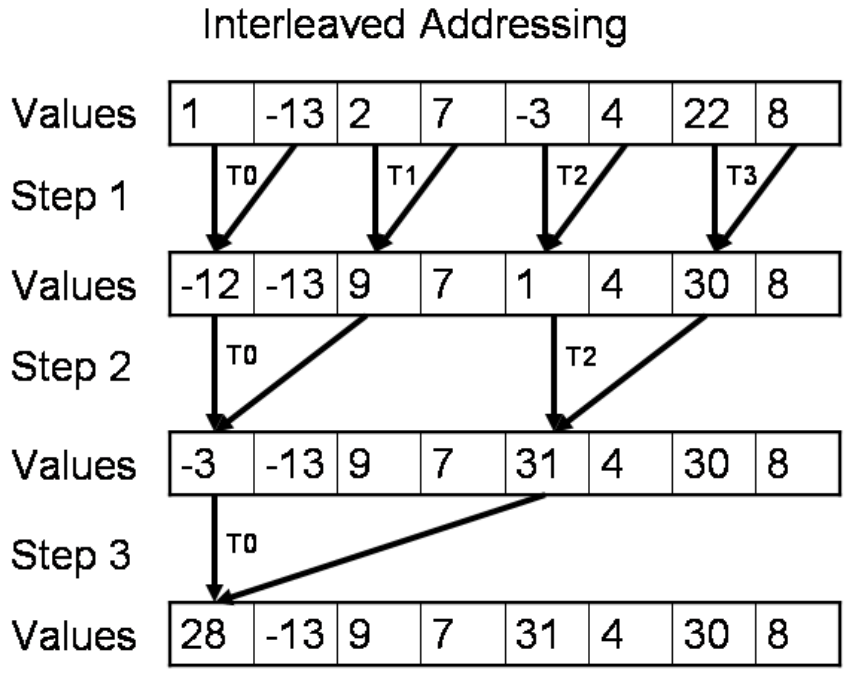
This implementation uses global memory for all operations as everything is directly modified in d. The conditional if statement testing the thread index to determine whether to reduce to that element at line 13 causes divergent branching within warps because only certain threads need to add elements to themselves at each iteration. This implementation is the basis for expansion into improvements and optimizations.

Optimization 1 - Non-Divergent Branching

The first optimization we tested was removal of divergent branching. To maintain the functionality of the code, lines 11 through 19 were replaced with the following code:

```
1     for(unsigned int i = 1; i < THREADS_PER_BLOCK; i <<= 1) {
2          unsigned int idx = 2 * i * threadIdx.x;
3
4          if (idx < THREADS_PER_BLOCK)
5               d[idx] += d[idx + i];
6
7          __syncthreads();
8     }
```

The replacement of the conditional statement in theory should prevent the current warp from branching. This is accomplished through assigning a contiguous subset of threads by modifying idx instead of threads at certain intervals from operating. In relation to the previous implementation, the reduction would now operate as such:



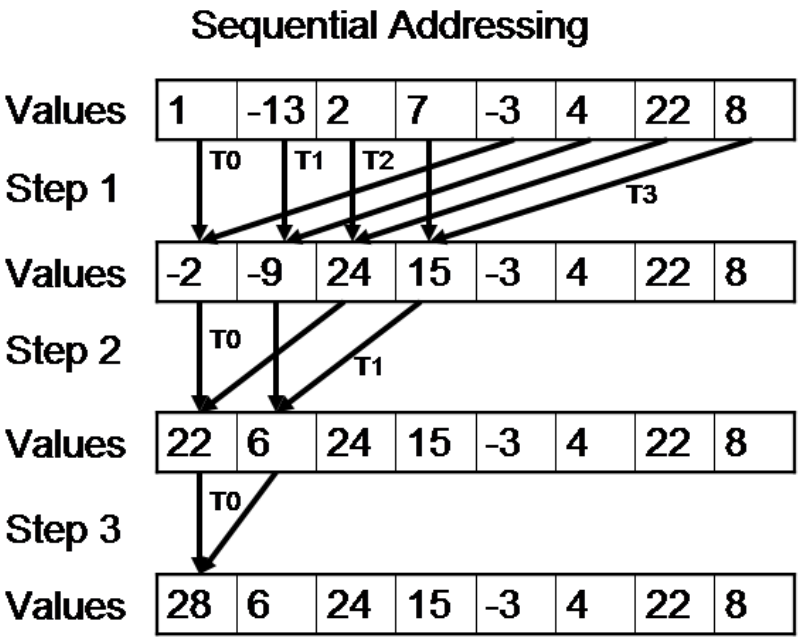
In practice however, this change actually led to comparable if not a decrease in performance from the previous implementation. Because of this, we ruled out divergent branching as a major factor in reduction performance.

Optimization 2 - Sequential Addressing

Given that locality of data points in memory should improve performance in the presence of caching and coalesced memory access restrictions, we investigated how a more cache-friendly implementation would fare against the previous two. We implemented a reduction that utilizes sequential memory accesses instead of interleaved accesses by replacing the previous loop with the following code:

```
1     for(unsigned int i = THREADS_PER_BLOCK / 2; i > 0;
2           i >>= 1) {
3           unsigned int idx = threadIdx.x;
4
5           if (idx < i)
6                 d[idx] += d[idx + i];
7
8           __syncthreads();
9     }
```

The above loop reduces the entire upper portion of the current working array to the entire lower portion of the working array. For example, with 1024 integers the upper 512 integers are reduced into the lower 512 integers. Next, the upper 256 integers of the lower 512 integers from before are reduced with the lower 256 integers. This cycle continues until there remains only the reduced element of the entire array in the first element of the array and can be illustrated as such:



This is in contrast to the previous two methods that used interleaved addressing, which reduced elements that were further and further apart in the input array. It should also be noted that this optimization will work only if the reduction operator, in this case addition, is also commutative.

Optimization 3 - Sequential Addressing with Shared Memory

The next optimization tested was the use of shared memory. All of the previous implementations used global memory, which is known to be the slowest, or highest-latency, memory on a GPU. By moving the data on the GPU to shared memory however, we suspected that the lower latency of shared memory would provide improvements over previous implementations. We added onto the previous sequential addressing example by changing the inner loop to utilize shared memory:

```
1    //New shared memory array per block, stores the first
2    //reduction immediately from global memory
3    sd[threadIdx.x] = d[threadIdx.x] +
4         d[threadIdx.x + THREADS_PER_BLOCK / 2];
5
6    //Ensures that all data points have been set in sd
7    __syncthreads();
8
9    //Since the initial reduction is set upon initialization of
10   //sd, we need one less iteration of the reduction
11   for(unsigned int i = THREADS_PER_BLOCK / 4; i > 0;
12        i >>= 1) {
13        unsigned int idx = threadIdx.x;
14
15        if (idx < i)
16             sd[idx] += sd[idx + i];
17        __syncthreads();
18   }
19
20   //Set the final result of the reduction into global memory
21   if(threadIdx.x == 0) {
22        d[0] = sd[0];
23   }
```

The improvements of this code is the move from working in global memory exclusively to working mostly in shared memory, reducing the number of required threads in half. Reducing the number of threads is achieved by line 3 by performing the first reduction when storing the input into shared memory.

Optimization 4 - Register Memory, Shuffle Down Instruction

The fastest layer of memory available on GPUs is the register file, which is a small amount of memory given to each thread for storing initial operands and results before writing back to either shared memory or global memory. A fairly recent addition to the CUDA

architecture are shuffle instructions, which allow communication of a thread's registers between threads within a warp.

With regards to reductions, the shfl_down instruction is useful. With this instruction, a programmer can tell threads to point to another thread's set of registers to perform an operation. The shfl_down instruction works within a single warp, allowing the user to move register values of higher thread indices to lower thread indices by a given offset. This allows for threads to share data without writing to memory. This implementation replaces the main loop with the following code:

```
1    //Temporary value storing final results at threadIdx.x % 32
2    int tmp = d[threadIdx.x] +
3         d[threadIdx.x + THREADS_PER_BLOCK / 2];
4
5    //Perform the shuffle down operation
6    //Works on sets of 32 elements
7    for(int offset = 16; offset > 0; offset >>= 1) {
8         tmp += __shfl_down(tmp, offset);
9    }
10
11   //All main threads add their values to the first
12   //position in the array that will be returned
13   if(threadIdx.x % 32 == 0) {
14        atomicAdd(d_org, tmp);
15   }
16
17   //Syncthreads here to prevent data race on next loop.
18   __syncthreads();
```

Once again, because the shuffle instruction uses sequential addressing and we store the first reduction into register memory, we only require half the number of threads as the naive implementation. Because shuffle instructions can transfer values only within a warp of 32 consecutive threads, we can only reduce up to 32 elements in each reduction loop. To reduce more elements one could copy these results to another array and perform the same operation with only the first 32 or 16 threads. We opted to instead use the atomicAdd function in CUDA to gather the sub-results into one aggregate on lines 13 through 15.

```
19   //Subtract the initial value of the first position in the
20   //returning array
21   if(threadIdx.x == 0) {
22        atomicAdd(d_org, -org);  //d_org is a pointer to d[0]
23   }
```

In addition, the above lines after the end of the previous code snippet are necessary to receive correct results. Since we are reducing each sub-reduction to position 0 in the array, and the original value of position 0 was never cleared, we need to subtract that original value from the final answer. The reason this problem exists in our tests is because we store the results of the

reductions back into the input array of values. Had we decided to use a separate array for output storage, this would not be a problem.

Optimization 5 - CUB Library Default

We compared our reduction code to an implementation from the CUDA UnBound (CUB) Library of GPU primitive operations [8]. The reduction loop for each trial was replaced with the following CUB calls:

```
1          typedef cub::BlockReduce<int, THREADS_PER_BLOCK>
2               BlockReduceT;
3
4          __shared__ typename BlockReduceT::TempStorage
5               tempStorage;
6
7          int result; //Temporary variable to store final result
8
9          //Perform reduction
10         if(threadIdx.x < THREADS_PER_BLOCK) {
11              result = BlockReduceT(tempStorage).Sum(
12                   d[threadIdx.x]);
13         }
14
15         //Add the final result to the first position in the
16         //array
17         if(threadIdx.x == 0) {
18              atomicAdd(d_org, result);
19         }
20         __syncthreads();
21    }
```

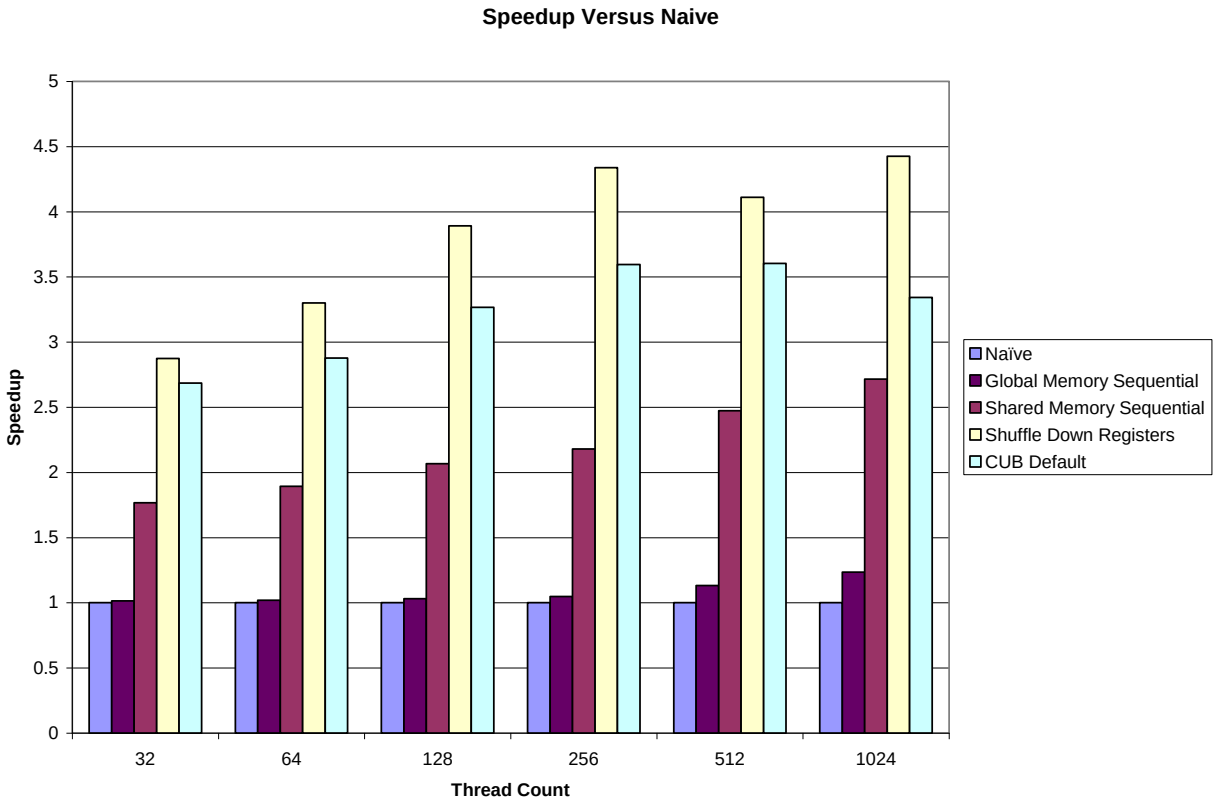
The code above first allocates temporary storage on line 4 to gather the sub-reduction results from each warp. Next, the CUB Block Reduce method is called on line 11 by each thread to reduce the elements. Finally, through lines 17 to 19, the final result of the reduction is reduced to the first element in the return array.

```
22    //Subtract the initial value of the first position in the
23    //returning array
24    if(threadIdx.x == 0) {
25         atomicAdd(d_org, -org);
26    }
```

As with the Register Memory implementation, the above lines of code are used to subtract the initial value of the input array to provide the correct result and would be removed if an output array were used instead of the input array d_in for output.

Results - Block-Wide Reduction

Below is a summary of the speedup in number of average cycles when compared to the initial naive approach:

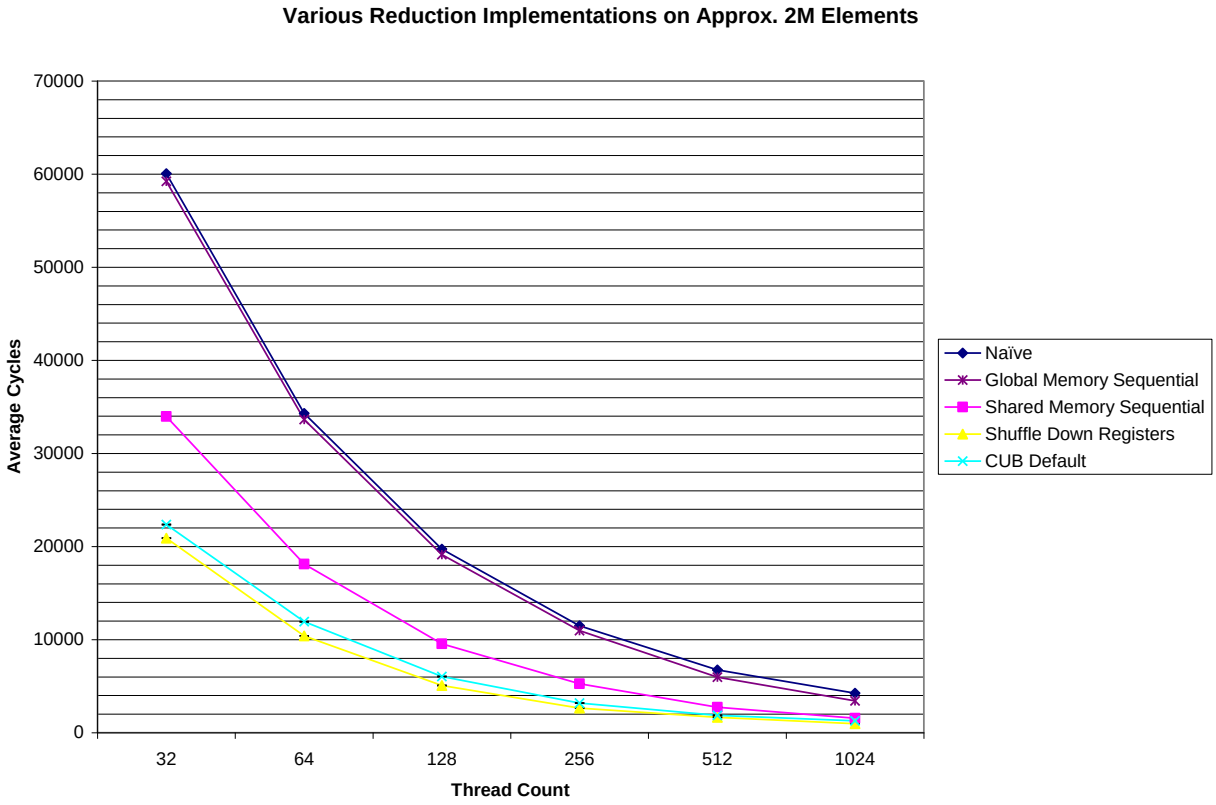


The switch from interleaved addressing to sequential addressing showed modest improvement in performance. This improvement ranged from 1.4% with the lowest thread count in our tests at 32 threads per block to 19% with the maximum 1024 threads per block. This shows that given larger sets of working data, the ability to utilize coalesced reads and writes provides greater benefits to reduction.

The increase in performance was much greater between the shared memory and naive approaches, ranging from 43% to 63% using 32 to 1024 threads per block. This speedup clearly demonstrated the impact working memory type has on performance of reductions.

Compared to the initial naive approach, the register version performed even better running around 65% to 77% using 32 to 1024 threads. Comparatively, the speedup between shared memory and registers was 20%.

Finally, the CUB implementation of reduction was 63% to 72% faster than naive with 32 to 512 threads, dropping to 70% with 1024 threads. Compared to the register version of the code, the CUB library ran 3-7% slower. The average cycles per implementation per thread count are drawn in the graph below:



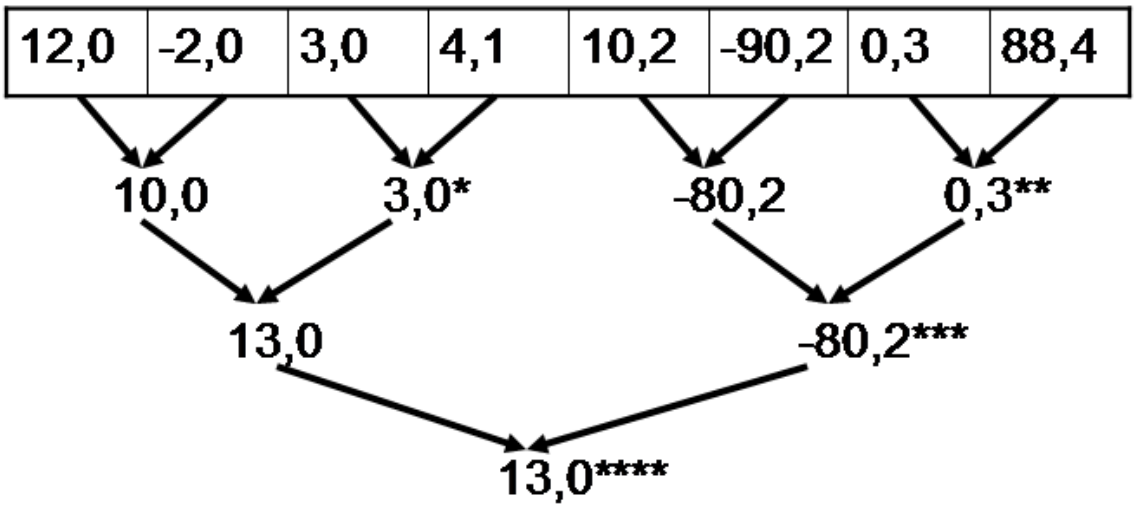
In conclusion, our tests showed that the shared memory with shuffle_down instructions would most likely be the best option for performing reductions on large sets of data. Although branch divergence and cache coherence were problems for reductions, memory types seemed to play the most significant role in the speedup of reduction operations. The most important factor for determining the speed of the reduction operation is the location of the data on the GPU, not necessarily the method in which it is reduced. Finally, as expected, the naive approach was by far the slowest implementation.

Chapter 3

Segmented Reduction

Because our investigation revealed the importance of the location of working memory, we sought to implement these new found ideas into a reduction for MERCATOR. However, reductions done in MERCATOR must reduce different sets of elements within the same output buffer to different aggregates. That is, different data streams would be combined into a single output buffer that would need to be reduced to the correct aggregate.

To simultaneously reduce several mixed data streams, one can utilize *segmented reduction* as outlined by [7]. A segmented reduction reduces elements of a similar key from the input array. An array of keys and an array of data elements are presented to the segmented reduction algorithm. All elements with the same key are reduced to a single aggregate, and the segmented reduction returns the aggregates for each key as the image below illustrates:



Tree-Based Segmented Reduction: '+' Operator

* Key 1 Reduction Finished (= 4)

** Key 4 Reduction Finished (= 88)

*** Key 3 Reduction Finished (= 0)

**** Key 2 Reduction Finished (= -80)

Key 0 Reduction Finalized at 1 Element (= 13)

As stated before, there are two arrays, one with values and the other with keys. The illustration shows the value of the index on the left and the associated key to the right. When reducing, the algorithm first checks whether the elements have matching keys. If so, the elements are reduced as normal and the common key is carried down. When the keys do not match, the right operand is reduced to an aggregate array as this is the finalized value for the given key. The left operand is then carried down for the next reduction as is. The stages with asterisks represent the mismatch of keys and the finalization of the right operand.

Our tests for segmented reduction were the same as block-wide reduction in Chapter 2 with minor changes. Instead of ranging our tests from 32 to 1024 threads, our range was from 128 to 1024 threads. Unlike block-wide reduction, we also needed to specify how many keys, or queues, we would use. For our tests, the key count remained constant at 16 keys, with all values having a randomly assigned key.

Naïve Approach

A simple implementation of the segmented reduction outlined in [7] is based on principles similar to the initial naïve implementation of Chapter 2. This approach uses interleaved addressing to reduce elements with multiple keys. Outlined below is pseudo-code that performs this operation:

```
1     CUB BlockRadixSort(keys, values);
2
```

```
3     __shared__ unsigned int  skeys[THREADS_PER_BLOCK];
4     __shared__ ValT          svals[THREADS_PER_BLOCK];
5
6     unsigned int lo, ro;
7
8     tid = thread_index;
9
10    //Perform interleaved reduction
11    for(d = 1; d < THREADS_PER_BLOCK; d <<= 1) {
12         if((threadidx & ((d << 1) - 1)) == 0) {
13              //Get left key
14              lo = skeys[tid];
15              //Get right key
16              ro = skeys[tid + d];
17
18              //Keys not the same
19              if(lo != ro) {
20                   aggregate[ro] =
21                   operation(aggregate[ro], svals[tid + d]);
22              }
23
24              //Keys are the same
25              else {
26                   svals[tid] =
27                        operation(svals[tid], svals[tid + d]);
28              }
29         }
30    }
31
32    //Finish reduction for first key
33    if(tid == 0) {
34         aggregate[0] = operation(aggregate[0], svals[tid]);
35    }
36
37    //Move results back to global memory
38    if(tid < NQUEUES) {
39         vals[tid] = aggregate[tid];
40    }
```

This code first utilizes the CUB library to perform a radix sort on the input arrays of keys and values, with values of key 0 being the initial elements. Sorting ensures that the values with the same key are adjacent in the array to be reduced. Next, the elements are reduced in a tree structure like the diagram of the Naïve implementation. However, lines 19 through 28 replace the actual reduction to accommodate the use of keys. When a reduction operation is called, the current thread first checks the keys of the values. When the keys are the same, they are reduced as normally as in line 26. If the keys are not the same, the right operand is the final aggregate for

the current key and is reduced to the aggregate array. After the reductions are performed, depending on the number of keys provided to the system, the aggregates are saved for use later.

It is non-trivial to write a segmented reduction that utilizes sequential addressing and shuffle instructions. When reducing, using a sequential approach is difficult because the current upper half of the working array of values may not have the same keys as the lower half, thus requiring atomic functions to produce aggregates. In addition, shuffle instructions would also require major re-workings of the segmented algorithm. The main issue associated with shuffle instructions is that they require the operation to be in a single warp. When operations are across warp boundaries, the user must perform an intermediary step moving results between warps. The warp boundaries also cause issues with keys, as one can only produce an aggregate for each key of each warp. These warps would then need to communicate to produce an aggregate for the entire block.

Optimized Approach

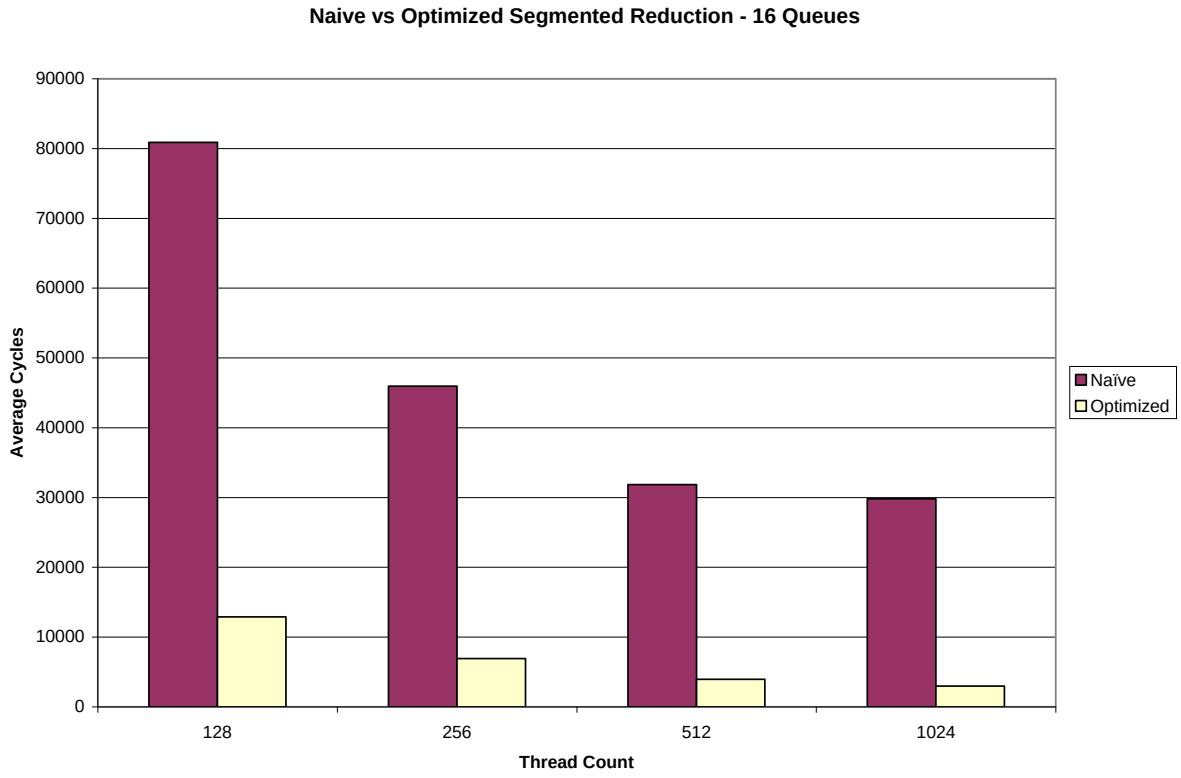
An optimized version of the segmented reduction performs the segmented reduction algorithm as outlined in the following pseudo-code:

```
1     for each warp:
2          Call WarpSort(keys, vals);
4          ReduceByKey(shuffle_instructions);
5
6     Copy WarpReduce Results to shared memory
7
8     for each key:
9          CUB BlockReduce(skeys, svals);
10
11    Copy BlockReduce Results to global memory
```

Some notable differences between the naïve and optimized implementations are in memory usage. Although the reduction is performed in a similar manner, the optimized version utilizes the faster shuffle instructions to reduce every warp before calling the CUB library's BlockReduce function using shared memory. As stated previously, by using shuffle instructions, we needed to incorporate a way for each warp to communicate their results to each other, which is where CUB's BlockReduce function is used.

Results - Segmented Reduction

A summary of the average cycles both implementations took is shown below:



The optimized approach shows a great improvement over the naive. By switching much of the computation to registers, we observed between an 6 to 10 times increase in performance. In conclusion, optimizing segmented reduction based on our results from Chapter 2 also yield performance improvements.

Chapter 4

Conclusion and Future Work

Having worked with multiple algorithms and implementations, we now have an understanding of future directions of our work. To begin, our findings showed that the type of memory utilized heavily influences the performance of block-wide reductions. In contrast, branch divergence and memory locality had little to no impact to run times. From these tests we have multiple reduction implementations. Our initial, naive approach which used interleaved addressing and global memory with divergent branching within warps. The next implementation utilized the same approach, but without divergent branching. This proved to provide no performance benefit on current hardware. Next, we developed a sequential addressing implementation which saw modest performance increases. Then, to test the memory constraints on performance, we converted the previous implementations to use shared memory instead of global, providing great performance improvements. Further improving upon the shared memory version, we implemented a sequential version which utilized register memory that also provided great performance gains. Finally, we implemented the reduction function of a well known library, CUB, to gain knowledge about the current state of general purpose reduction implementations. CUB was close to the performance of the register implementation.

By applying these ideas to segmented reductions, we found that these ideas still held true. The naive implementation for segmented reduction, much like for normal reductions, was 6 to 10 times slower than the optimized version.

By adhering to the principles learned through normal block-wide reductions, we were able to observe the same performance benefits with segmented reductions. Now, with a working version of an optimized segmented reduction, future work will include implementing the optimized segmented reduction into the MERCATOR system.

[1] MapReduce: Simplified Data Processing on Large Clusters, J. Dean, S. Ghemawat, OSDI'04: Sixth Symposium on Operating System Design and Implementation, December 2004
[2] Cuda Toolkit Documentation, NVIDIA Corporation, 12 January 2017, http://docs.NVIDIA.com/cuda/cuda-c-programming-guide/#axzz4cZZhMRHt
[3] CUDA Warps and Occupancy, GPU Computing Webinar, Drs. J. Luitjens, S. Rennich, 12 September 2011, http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf
[4] Cole, Stephen V. and Buhler, Jeremy, "MERCATOR (Mapping EnumeRATOR for CUDA) User's Manual" Report Number: WUCSE-2016-003 (2016). All Computer Science and Engineering Research. http://openscholarship.wustl.edu/cse_research/980
[5] Optimizing Parallel Reduction in CUDA, Mark Harris, NVIDIA Corporation, http://developer.download.NVIDIA.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf
[6] CUDA Programming, M. Romero, R. Urra, Computer Engineering RIT, http://cuda.ce.rit.edu/cuda_overview/cuda_overview.htm
[7] Algorithmic Strategies for Optimizing the Parallel Reduction Primitive in CUDA, Drs. P. Martín, L. Ayuso, R. Torres, A. Gavilanes, Universidad Complutense de Madrid, 2012
[8] CUDA UnBound (CUB) Library In https://nvlabs.github.io/cub/ by Duane Merrill, NVIDIA-Labs