PARALLEL MERGE FOR MANY-CORE ARCHITECTURES

BY

JIE LV

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Professor Wen-Mei W. Hwu

Abstract

This thesis proposes a novel GPU implementation for merging two sorted arrays.

We consider the problem of merging two arrays A and B into a single array C. Each element in the arrays has a key. An ordering relation denoted by $\leq$ is defined on the keys. Array A and array B have m and n elements, respectively, where m and n do not have to be equal. Both array A and array B are sorted based on the ordering relation. The task is to produce the output array C of size $m + n$. Array C consists of all the input elements from array A and array B, and is sorted by the ordering relation.

We applied several GPU-specific optimizations to a parallel merge algorithm. The optimizations include coordinating the memory access pattern, making full use of the shared memory and reducing the thread divergence. Our implementation achieves up to 10x and 40x speedup on Titan-Z and GTX 980 GPU respectively compared to thrust merge implementation.

I would like to dedicate this thesis to my parents,
for their endless love and unconditional support.

Acknowledgments

I would first like to thank my thesis adviser, Professor Wen-Mei W. Hwu, for his tremendous mentorship and support. He has always motivated my work and been patient with me. His wisdom will keep inspiring me in my professional career and personal life.

I would also like to thank all members of the IMPACT research group, past and present, for their help and camaraderie. They are, in no particular order, Sao-Jie Chen, Juan Gómez-Luna, Nicolás Guil Mata, Nacho Navarro, Tom Jablin, John Larson, Xuhao Chen, Chris Rodrigues, Hee-Seok Kim, Li-Wen Chang, Izzat El Hajj, Abdul Dakkak, Simon Garcia de Gonzalo, Carl Pearson, Sitao Huang, Cheng Li, and Mert Hidayetoglu.

I want to thank Marie-Pierre Lassiva-Moulin and Andrew Schuh, for their help, and convey my gratitude to Jamie Hutchinson for his help on editing this thesis.

Finally, I would like to thank my parents for their endless love and unconditional support.

Table of Contents

List of Figures

Chapter 1

Introduction

Merge is an important operation in contemporary computing systems. It is used as a subroutine by many popular algorithms and applications such as merge sort and database operations. As a frequently used subroutine, the performance of merge is critical.

As the need for high performance computing grows, single-chip multiprocessors (CMPs) become more and more popular. However, most existing sequential algorithms cannot fully utilize the computing resource on CMPs. To exploit the performance of CMPs, parallel algorithms are developed. In [1], Siebert et al. proposed a parallel merge algorithm. With p processing elements, the time complexity of merge could be reduced from $O(m+n)$ to $O(\frac{m+n}{p} + \log min(m,n))$. This parallel merge algorithm can be implemented on CMPs using openMP with minimum effort, and achieve considerable speedup compared to the sequential merge. This algorithm can also run on graphics processing units (GPUs). We implemented it on GPUs (we call it Naive Parallel Merge) without any GPU-specific optimizations. To the best of our knowledge, thrust library has the fastest GPU merge implementation.

Figure 1.1 shows the memory throughput for different input sizes of naive parallel merge and thrust merge. We observe that for some input sizes (from 1K to 1M), the performance of naive parallel merge is better than that of thrust library.

Compared to CMPs, GPUs have more cores and larger memory bandwidth. A direct implementation of this parallel merge algorithm on GPUs (naive parallel merge) will result in suboptimal performance due to the underlying architecture differences between GPUs and CMPs. To exploit the massive parallelism on GPUs, we need to coordinate the memory access pattern, make full use of the shared memory and reduce the thread divergence. This motivates us to do further optimizations on naive parallel merge, and find a better GPU merge implementation that outperforms thrust library.

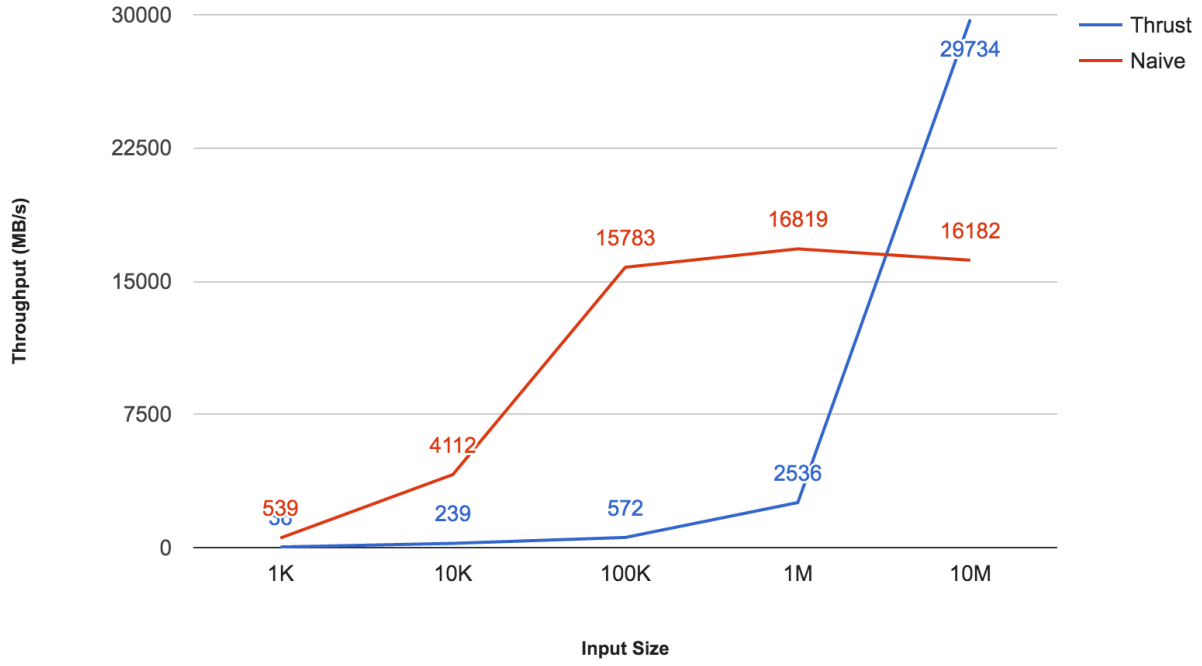

Figure 1.1: Performance of Thrust Merge and Naive Parallel Merge on GTX 980

This thesis proposes a novel GPU implementation for merging two sorted arrays. Our implementation achieves up to 40x speedup compared to the thrust library for certain input sizes.

The rest of the thesis is organized as follows: Chapter 2 describes algorithm and architecture background. The parallel merge algorithm [1] is described in Chapter 3. Chapter 4 describes the implementations of the parallel merge algorithm with GPU-specific optimizations. Chapter 5 describes further optimizations. Evaluation is presented in Chapter 6. Finally, Chapter 7 concludes the thesis.

Chapter 2

Background

2.1 Algorithm Background

2.1.1 Problem Definition

In this thesis, we consider the problem of merging two arrays A and B into a single array C. Each element in the array has a key. An ordering relation denoted by $\leq$ is defined on the keys. Array A and array B have m and n elements, respectively, where m and n do not have to be equal. Both array A and array B are sorted based on the ordering relation. The task is to produce the output array C of size $m + n$. Array C consists of all the input elements from array A and array B, and is sorted by the ordering relation.

Figure 2.1 gives an example of merging two arrays of integers. We will use this example throughout this thesis to demonstrate the parallel merge algorithm.

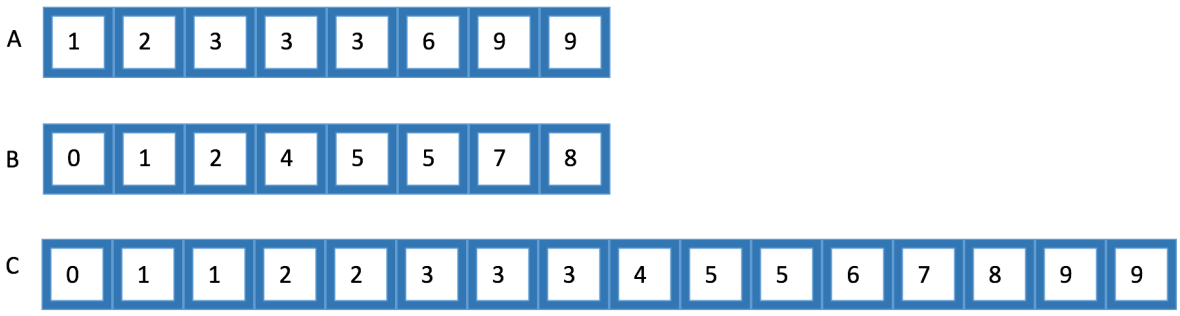

Figure 2.1: Merge Example

2.1.2 Sequential Merge

Sequentially merging two sorted arrays has been solved for a long time. Listing 2.1 shows the sequential code implemented in C++.

```
void merge(int *A, int m, int *B, int n, int *C)
{
    int i, j, k, l;
    i = 0;                  //index A
    j = 0;                  //index B
    k = 0;                  //index C

    /* handle the start of A[] and B[] */
    while ((i < m) && (j < n))
    {
        if (A[i] <= B[j]) {
            C[k] = A[i];
            i++;
        } else {
            C[k] = B[j];
            j++;
        }
        k++;
    }
    if (i == m) {           //handle remaining b[]
        for (l = j; l < n; l++)
        {
            C[k] = B[l];
            k++;
        }
    } else {                //handle remaining a[]
        for (l = i; l <m; l++)
        {
            C[k] = A[l];
            k++;
        }
    }
}
```

Listing 2.1: Sequential Merge Implementation in C++

This implementation uses a while loop to do the merge. It first merges A and B to C when there are both remaining A and remaining B. When it reaches the end of A or B, it copies the remaining A or B to C. The time complexity of this implementation is $O(m+n)$.

2.2 Architecture Background

2.2.1 GPU Architecture

- **Computation:** GPUs are designed for high computation throughput instead of low latency. GPUs typically contain hundreds of cores. Programmers are allowed to set the number of blocks and the number of threads for each block to create massive parallelism that utilizes the huge number of cores on a GPU.

- **Memory hierarchy:** The GPU memory hierarchy consists of global memory, shared memory and registers. Global memory is shared across thread blocks. It is the largest in terms of size. However, it is the slowest. Shared memory is shared only among the threads in a single block. It is faster than global memory but slower than registers. The size of shared memory is limited. In all the GPUs we use, the shared memory size per block is 48 KB. Registers are the fastest but have the smallest size.

2.2.2 Global Memory Coalescing

When we launch a kernel on GPU, it is executed by the parallel threads. If the kernel has a global memory reference, then each thread will also generate a global memory request, and the memory addresses for each thread will most likely be different. Listing 2.2 shows an example of global memory reference.

```
__global__ void kernel(int* a)
{
    int tid = threadIdx.x;
    a[tid] = tid;
}
```

Listing 2.2: Memory Access Pattern

These memory requests are grouped into a number of memory transactions. When consecutive threads access consecutive global memory addresses (as in Listing 2.2), we call this coalesced access. When coalesced access happens, a single transaction may be implemented [2] to maximize the bandwidth usage.

When the memory addresses accessed by threads are not consecutive (e.g., for an access $a[tid * N]$ instead of $a[tid]$ in Listing 2.2), we call this non-coalesced access. It is not possible anymore to pack the different requests from different threads into a single transaction. In the worst case, we may need to make one transaction per thread. This will result in a poor usage of memory bandwidth.

It is desirable to make all the accesses to global memory coalesced. One approach is to use shared memory as the scratch pad. This approach requires complex code restructuring and is one of the GPU-specific optimizations that we use to improve the performance of parallel merge.

Chapter 3

Parallel Merge Algorithm

In [1], Siebert et al. proposed a parallel merge algorithm, in which each processing element calculates the output range it is going to produce, and uses that output range as the input to a **co-rank function** to identify the corresponding input ranges that generate the output. Finally, each processing element calls the sequential merge function to do the merge independently in parallel.

3.1 Co-rank Function

Let A and B be two input arrays with m and n elements respectively. Both input arrays are sorted according to an ordering relation $\leq$. The index of the arrays starts from 0. The task is to merge A and B into an array C with $m+n$ elements. They use $C[m+n] = merge(A[m], B[n], \leq)$ to denote this task. In their paper, Siebert et al. pointed out two observations:

- For any i, $0 \leq i < m+n$ in C, there is either a j, $0 \leq j < m$ such that $C[i] = A[j]$ or a k, $0 \leq k < n$ such that $C[i] = B[k]$.
- For any i-element prefix $C[0, ..., i-1]$ of C, there must be indices j and k of A and B such that $C[0, ..., i-1] = merge(A[0, ..., j-1], B[0, ..., k-1], \leq)$.

Siebert et al. also proved that j and k, which define the prefixes of A and B needed to produce the prefix of C of length i, are unique. For an element $C[i]$, they call the index i its rank. And they call the unique indices j and k its co-ranks. Consequently, they use the term **co-rank** for the process of determining j and k from A, m, B, n and i. Notice that $i = j + k$ because the number of elements in the output array equals the sum of the number

of elements in the input arrays. Figure 3.1 shows an example of co-rank. In this example, $C[16] = merge(A[8], B[8], \leq)$. $C[8] = merge(A[5], B[3], \leq)$. Therefore, the co-rank of 8 is 5 and 3.

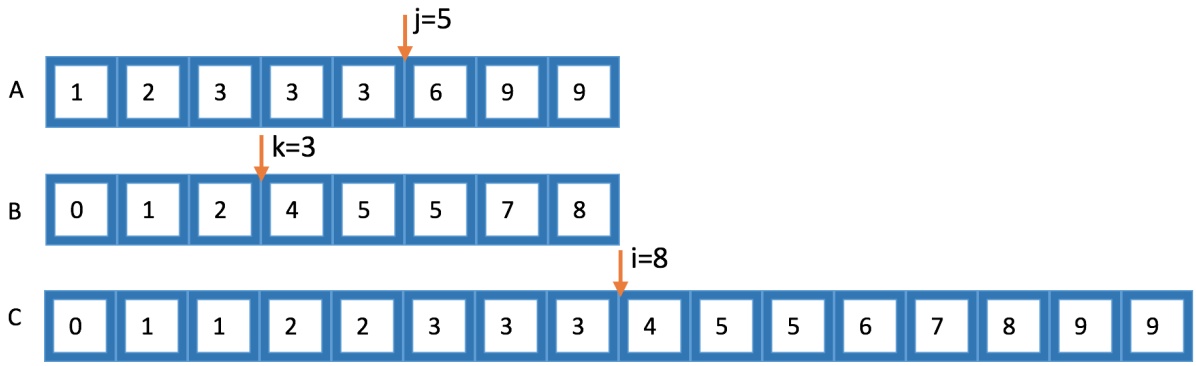

Figure 3.1: Co-rank Example

The pseudo code to find the co-rank from A, m, B, n and i is also given in their paper. In listing 3.1, we transform their pseudo code into the C++ implementation of co-rank function. We calculate j by using $j = co\_rank\_j(i, A, m, B, n)$. Then we calculate k by $k = i - j$.

```
int co_rank_j(int i, int* A, int m, int* B, int n)
{
    int j = i < m ? i : m;                  //j = min(i,m)
    int k = i - j;
    int j_low = 0 > (i-n) ? 0 : i-n;        //j_low = max(0, i-n)
    int k_low;
    int delta;
    bool active = true;

    while(active)
    {
        if (j > 0 && k < n && A[j-1] > B[k]) {
            delta = ((j - j_low - 1) >> 1) + 1;
            k_low = k;
            j = j - delta;
            k = k + delta;
         } else if (k > 0 && j < m && B[k-1] >= A[j]) {
            delta = ((k - k_low - 1) >> 1) + 1;
            j_low = j;
            j = j + delta;
            k = k - delta;
        } else {
            active = false;
        }
    }
    return j;
}
```

Listing 3.1: Original Co-rank

3.2 Overall Parallel Merge Algorithm

The co-rank function provides a simple and efficient way to perform merging in parallel. Let p processing elements be given, all of which can access input and output arrays A, B and C. Each processing element has its own id r, $0 \leq r < p$.

Each processing element calculates the output range ($C[i\_start, ...i\_end]$) it is going to produce. The output ranges can be chosen such that they cover the whole output array of size $m + n$, and the size of output each processing element producing differs by at most 1. Then, each processing element computes the corresponding co-ranks for both the start and

end index. These co-ranks determine the input ranges of the input arrays this processing element needs to merge sequentially.

Listing 3.2 shows the pseudo code for the overall parallel merge algorithm.

```
void paralle_merge(int *A, int m, int *B, int n, int*C)
{
    r       = processing_id;         // 0 <= r < p
    i_start = floor(r*(m+n)/p);      // start index of output
    i_end   = floor((r+1)*(m+n)/p); // end   index of output
    j_start = co_rank_j(i_strat, A, m, B, n);
    j_end   = co_rank_j(i_end,   A, m, B, n);
    k_start = i_start - j_start;
    k_end   = i_end   - j_end;
    merge( A[j_start,...,j_end-1], B[k_start,...,k_end-1],
           C[i_start,...,i_end-1] );
}
```

Listing 3.2: Pseudo Code for Parallel Merge Algorithm

We use figure 3.2 to show an example of the parallel merge process. In this example, there are two processing elements ($p = 2$). These two processing elements are going to perform the task $C[16] = merge(A[8], B[8], \leq)$ collaboratively in parallel.

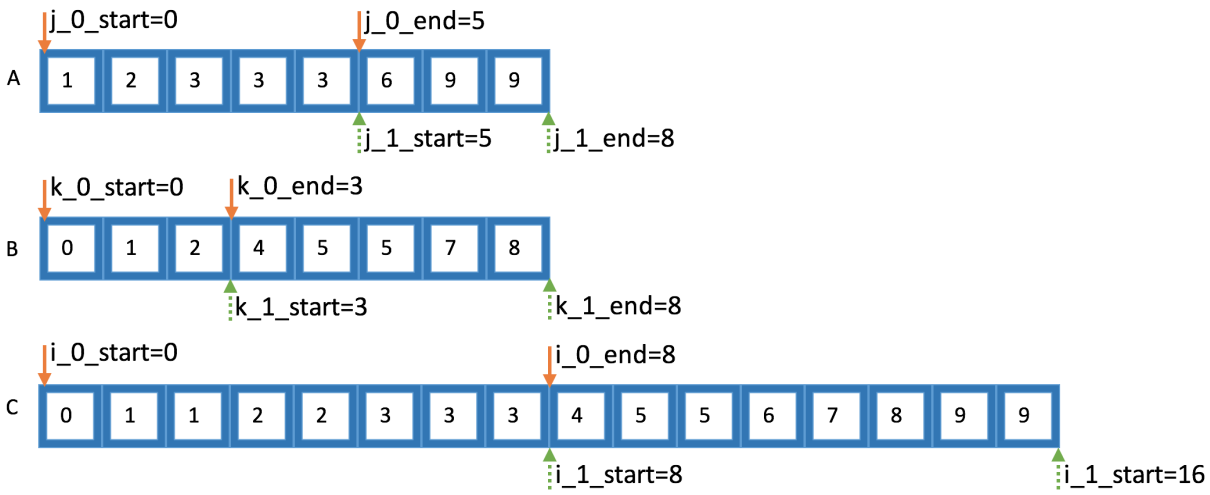

Figure 3.2: Parallel Merge Algorithm Example

For p_0 (solid red arrows), $r = 0$, $i\_start = 0$, $i\_end = 8$, p_0 is going to produce $C[0, ..., 7]$. After running the co-rank function, p_0 knows the input ranges: $j\_start = 0$, $j\_end = 5$,

$k\_start = 0$, $k\_end = 3$. Therefore, p_0 will call $C[0, ..., 7] = merge(A[0, ..., 4], B[0, ..., 2], \leq)$.

For p_1 (dashed green arrows), $r = 1$, $i\_start = 8$, $i\_end = 16$, p_1 is going to produce $C[8, ..., 15]$. After running the co-rank function, p_1 knows: $j\_start = 5$, $j\_end = 8$, $k\_start = 3$, $k\_end = 8$. So p_1 will call $C[8, ..., 15] = merge(A[5, ..., 7], B[3, ..., 7], \leq)$.

Because p_0 and p_1 are working on different parts of the input and output, they can run in parallel without interference. Also, the sizes of output they produce are the same. Therefore, load balance is guaranteed.

Chapter 4

Implementations of Parallel Merging Algorithm

We implement the parallel merge algorithm on CMPs using OpenMP before implementing the parallel merge algorithm on GPUs. The result is shown in Chapter 6.

Compared to CMPs, GPUs have more threads and larger memory bandwidth for the purpose of massive parallelism. However, a direct translation of a parallel algorithm that suits CMPs may not run efficiently on GPUs. To explore massive parallelism on GPUs, we need to coordinate the memory access pattern, make full use of the shared memory, reduce the thread divergence, improve the load balance for different processing units, and create enough parallelism.

We have three GPU implementations. We name the first implementation **naive parallel merge**. Naive parallel merge is a direct translation of the parallel merge algorithm on GPU without any GPU-specific optimization.

Since coalesced global memory is critical to improve the application performance that runs on GPU[3], we implement a second GPU version that utilizes shared memory as a scratch pad to make the accesses to global memory coalesced. We name it **single buffer parallel merge**.

In single buffer parallel merge, we only consume half of the data we load into the shared memory. The other half is wasted. To better utilize the data we load into shared memory, we implement a third version and we call it **double buffer parallel merge**.

4.1 Naive Parallel Merge

In naive parallel merge, we copy the input arrays $A[\,]$ and $B[\,]$ from host to device global memory first. Then each thread calculates the thread index (r) and total number of threads

(p) using $r = blockIdx.x * blockDim.x + threadIdx.x$ and $p = gridDim.x * blockDim.x$ respectively. Based on r and p, each thread calculates the output range it is going to produce, and uses the output range as the input to the co-rank function to identify the corresponding input ranges. After getting the input ranges, all threads start to merge independently by calling the sequential merge function in parallel and writing the result to $C[\,]$ in device global memory. Finally, we copy $C[\,]$ from device global memory back to the host.

Figure 4.1 shows an example of naive parallel merge. $A[\,]$, $B[\,]$ and $C[\,]$ are in device global memory. This example shows the work of one thread. After identifying its output range($C[\,]$) and input ranges($A[\,]$, $B[\,]$), it uses sequential merge to write the result to $C[\,]$. All the threads run in parallel and produce the result for the entire output array $C[\,]$.

Figure 4.1: Naive Parallel Merge

4.2 Single Buffer Parallel Merge

Naive parallel merge does not have a coalesced memory access pattern, so this implementation results in a poor usage of memory bandwidth [2]. To better utilize the memory bandwidth on GPU, coalesced global memory accesses are critical. For this reason, we use shared memory on GPU as a scratch pad, and make the access pattern to global memory coalesced in single buffer parallel merge. Single buffer parallel merge works as follows: Co-rank function is run in two levels, block level and thread level. At the block level, all the threads in the same block do the same searching. Each thread calculates the block index and total number of blocks using $b\_id = blockIdx.x$ and $b\_num = gridDim.x$ respectively.

Based on b_id and b_num, all threads within the same block calculate the output range that block is going to produce, and use the output range as the input to the co-rank function to identify the corresponding input ranges for that block. The co-rank function is run on global memory in the block level. After knowing the input ranges for the block, all threads in the block cooperatively load the input to the shared memory. In this way, we can guarantee that the global memory access pattern is coalesced.

However, shared memory may not be large enough to hold all the input data. So we create a loop. In each iteration, we load x elements from input array A and x elements from input array B into shared memory. This will produce x (not $2x$) elements to the output array C, because in extreme cases, all the output may come from one of the input array. We waste half of the data loaded into the shared memory.

Then, we run the co-rank function at thread level. Threads in a block will merge x elements using the data we load into shared memory. So the co-rank function is run on shared memory in the thread level. Each thread calculates the thread index and total number of threads in the block using $t\_id = threadIdx.x$ and $t\_num = blockDim.x$ respectively. Based on t_id and t_num, each thread calculates the output range that thread is going to produce, and uses the output range as the input to the co-rank function to identify the corresponding input ranges. Then each thread starts its work independently and calls the sequential merge function to perform the merge in parallel on the input we load to shared memory and write the output to device global memory. Figure 4.2 shows the first iteration of single buffer parallel merge. The solid orange box is the block level run of the co-rank function. The dashed green box is the first iteration of the thread level run of co-rank function. The data marked by the solid blue arrow are wasted.

In the next iteration, we will load the data we have not merged into shared memory, run co-rank in the thread level, and perform merge in parallel. Figure 4.3 shows the second iteration of single buffer parallel merge. The dashed red box is the second iteration of the thread level run of co-rank function.

The loop runs until we have merged all the data that block is going to produce (filling the entire solid orange box).

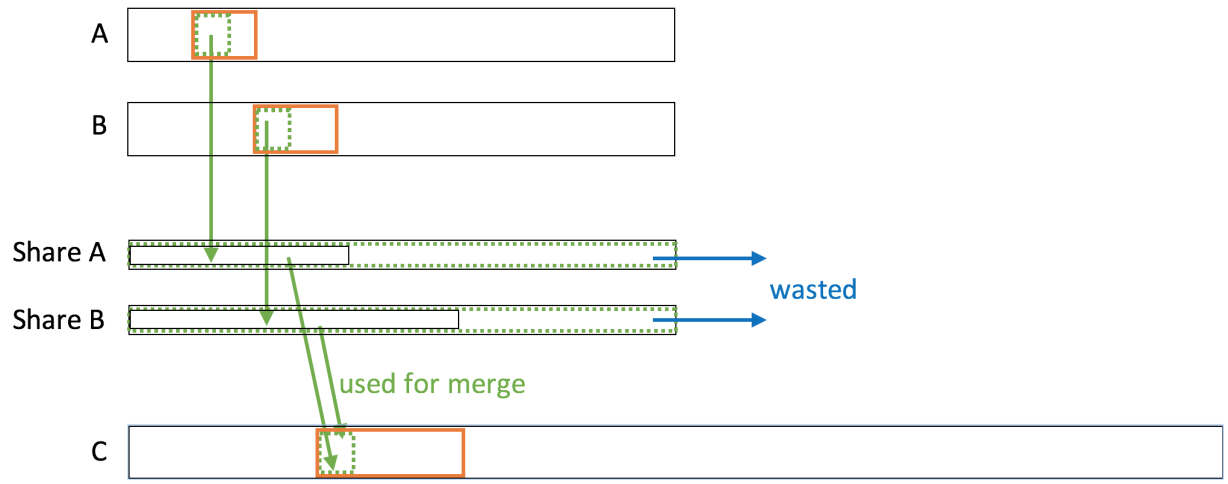

Figure 4.2: First Iteration of Single Buffer Parallel Merge

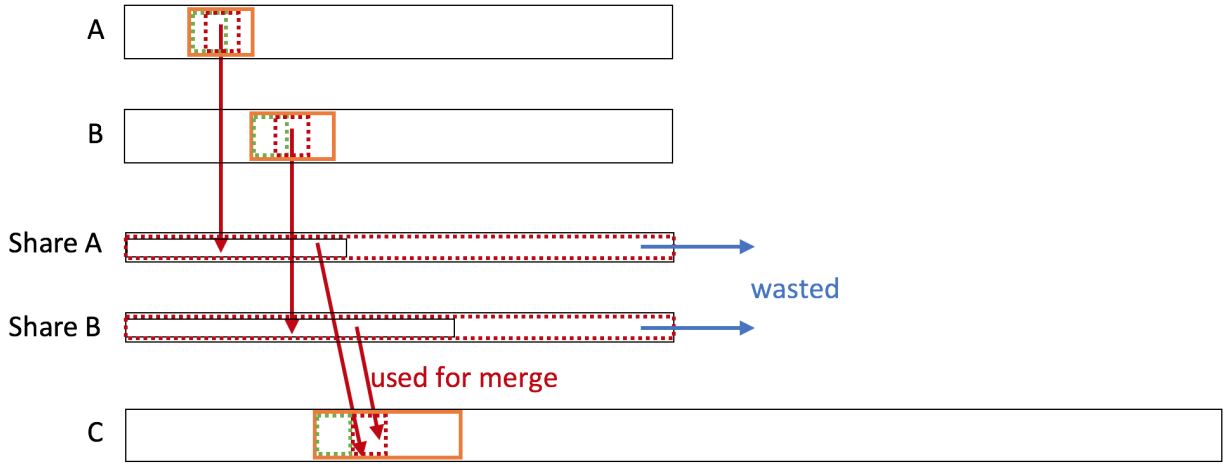

Figure 4.3: Second Iteration of Single Buffer Parallel Merge

4.3 Double Buffer Parallel Merge

In single buffer parallel merge, we only consume half of the data we load into shared memory. The other half is wasted. To further optimize the performance, we create the double buffer parallel merge, in which we utilize all the data we load into the shared memory.

The overall process is similar to single buffer parallel merge except for how we use shared memory. Co-rank function is run in two levels, block level and thread level. At the block

level, all the threads in the same block do the same searching. Each thread calculates the block index and total number of blocks using $b\_id = blockIdx.x$ and $b\_num = gridDim.x$ respectively. Based on b_id and b_num, all threads within the same block calculate the output range that block is going to produce, and use the output range as the input to the co-rank function to identify the corresponding input ranges for that block. After knowing the input ranges for the block, all threads in the block cooperatively load the input to the shared memory.

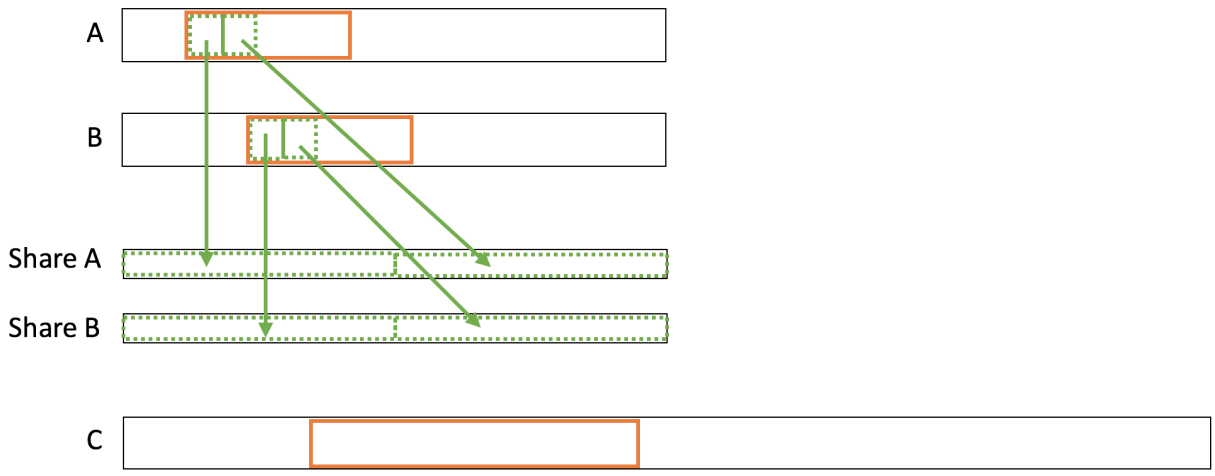

Figure 4.4: Initialization of Double Buffer Parallel Merge

We use a loop because shared memory may not be large enough to hold all the input data. When initializing, we load $2x$ elements from input array A and $2x$ elements from input array B into the shared memory. Figure 4.4 shows the initialization process.

In each later iteration, all the threads in a block will produce x elements to the output array C. If there are more than x elements in shared array A and shared array B, we do not load from global memory. Figures 4.5 and 4.6 show examples of the first iteration and second iteration of double buffer parallel merge. In these two iterations, there are enough data remaining in the shared memory, so we did not load from the global memory to shared memory.

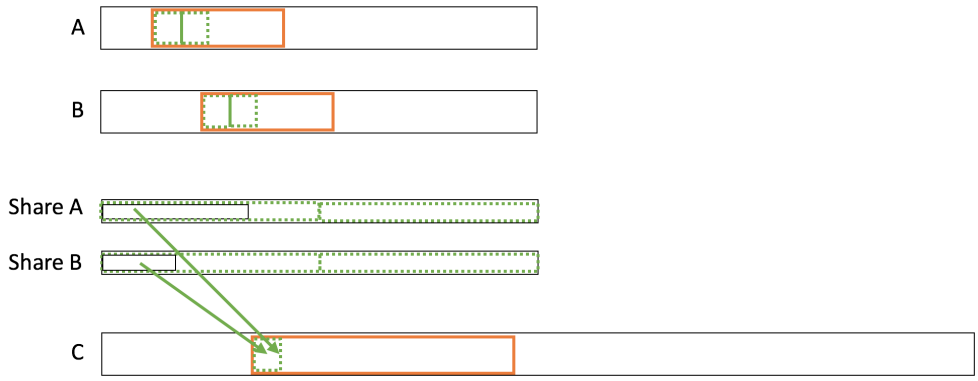

Figure 4.5: First Iteration of Double Buffer Parallel Merge

Figure 4.6: Second Iteration of Double Buffer Parallel Merge

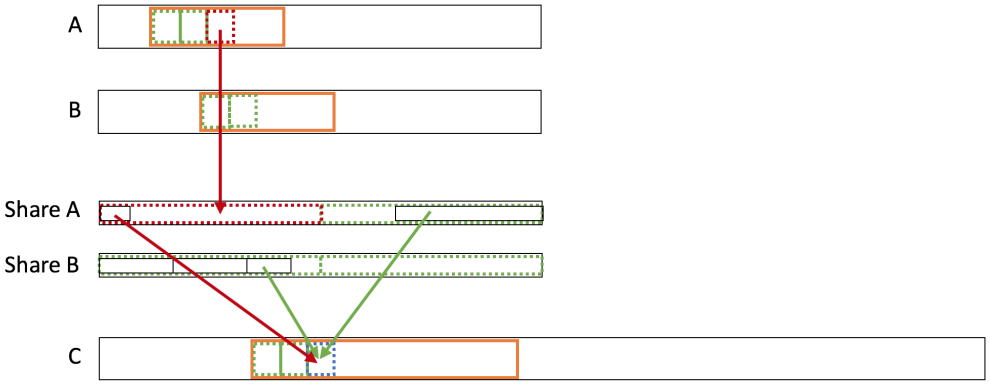

Figure 4.7: Third Iteration of Double Buffer Parallel Merge

If there are less than x elements in either shared array A or shared array B, we will load another x elements into shared memory. Figure 4.7 shows an example of the third iteration of double buffer parallel merge. In the third iteration, there are enough data in the shared array B. However, there are not enough (less than x) data in shared array A. So we load x elements from the global memory to shared memory (marked by the red box). Data in shared memory will wrap around.

Then we run the co-rank function at thread level. The co-rank function is run on shared memory. Each thread calculates the thread index and total number of threads in the block using $t\_id = threadIdx.x$ and $t\_num = blockDim.x$ respectively. Based on t_id and t_num, each thread calculates the output range that thread is going to produce, and uses the output range as the input to the co-rank function to identify the corresponding input ranges. Then each thread starts its work independently and calls the sequential merge function to do the merge in parallel on the input we load to shared memory and write the output to device global memory.

The loop runs until we have merged all the data that block is going to produce (filling the entire solid orange box).

Chapter 5

Further GPU Optimizations

5.1 Reduce Number of Calls to Co-rank Function

In all the GPU parallel merge implementations above, each thread needs to call the co-rank function twice: once for the start point of the output range, and once for the end point of the output range. Figure 5.1 gives an example. In this example, there are 8 threads. We mark the call to co-rank function using "$*$". Each thread will call co-rank twice to calculate j_start and j_end. Therefore, the total number of calls to co-rank function is 16.

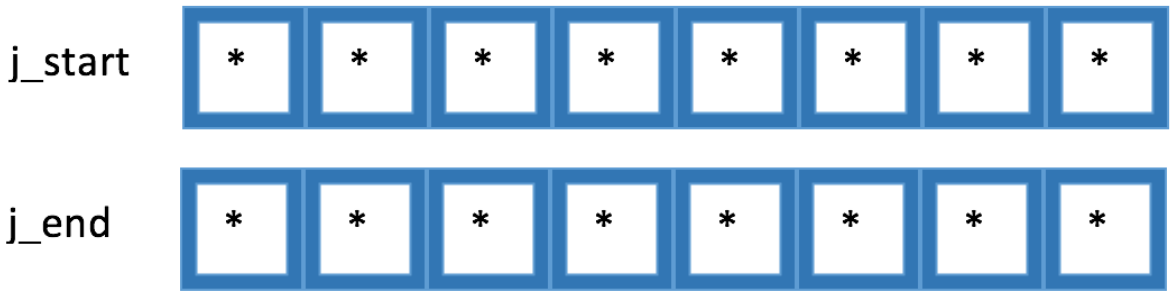

Figure 5.1: Number of Calls to Co-rank Function before Optimization

One observation we have is that the start point of thread r is the same as the end point of thread $r-1$. In the example we provide in figure 3.2, the start point of $p1$ is 8 and the end point of $p0$ is also 8. For this reason, we could reduce the number of calls to co-rank function. We first let all the threads calculate the co-rank for the end point, as shown in figure 5.2. Instead of using co-rank function again to calculate the co-rank for starting point, we store the result(j_end) in an array in shared memory. Then after a synchronization, we

read the co-rank of start point of thread r from the result of thread $r-1$. If thread index is 0, we will set the co-rank to 0 instead of reading from thread -1. Now we only need to call co-rank function for 8 times (as marked by "$*$" in figure 5.2) and reduce the number of calls by half. The overhead is that we are using more shared memory, and we need to perform a synchronization before reading the co-rank stored in shared memory.

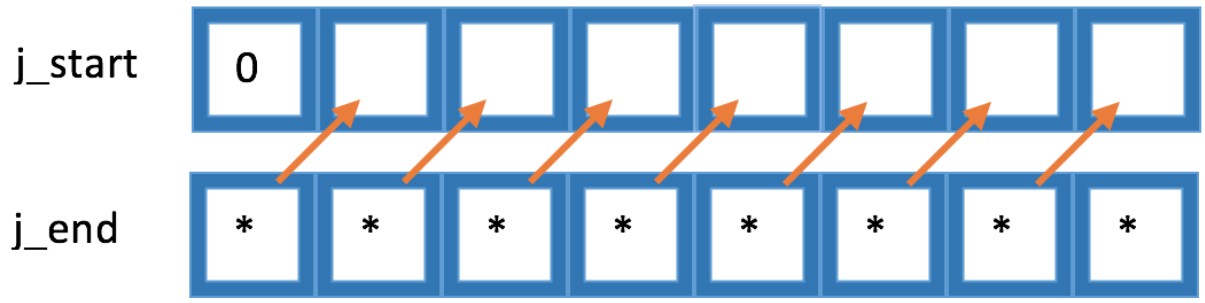

Figure 5.2: Number of Calls to Co-rank Function after Optimization

5.2 Change Control Divergence to Memory Divergence

Control divergence can degrade the performance of GPU application [4]. When the threads in the same warp take different branches for an $if-else$ statement, control divergence will occur. The hardware needs to execute the if part and $else$ part sequentially. This will hurt the overall performance. As a result, it is desirable to write code with a minimum amount of control divergence to achieve high performance. In parallel merge, we are concerned about two functions: merge and co-rank.

- Merge

  Listing 2.1 shows the original sequential merge code. It first merges A and B to C when there are both remaining A and remaining B. When it reaches the end of A or B, it copies the remaining A or B to C. The time complexity of this implementation is $O(m+n)$. We can see that the original merge has control divergence due to if and $else$ statements inside the while loop. To remove the control divergence, we use *selection*

to replace the *if* and *else* statements. Listing 5.1 shows the sequential merge code after removing control divergence. After this optimization, we effectively remove all the control divergence. However, memory divergence still exists because the threads in a warp are reading from different addresses, and these memory requests may complete at different time [5].

```
void merge(int *A, int m, int*B, int n, int*C)
{
  int count = m+n;
  int ai = 0, bi=0;
  for(int i=0; i< count; ++i)
  {
    bool p;
    bool c1 = (bi >= n);
    bool c2 = (ai >= m);
    p = c1 ? true : c2 ? false : A[ai]>B[bi] ? false : true;
    C[i] = p ? A[ai++] : B[bi++];
  }
}
```

Listing 5.1: Merge Remove Code Divergence

- Co-rank

  Listing 3.1 shows the original code for co-rank function. The code divergence also comes from the *if* and *else* statements. We replace the *if* and *else* statements by *selection*. The resulting code is shown in Listing 5.2.

```
int co_rank_j(int i, int* A, int m, int* B, int n)
{
    int j = i<m ? i : m;              //j = min(i,m)
    int k = i - j;
    int j_low = 0>(i-n) ? 0 : i-n;   //j_low = max(0, i-n)
    int k_low;
    int delta;

    while(1)
    {
        bool cond_1 = j > 0 && k <n && A[j-1] > B[k];
        bool cond_2 = k > 0 && j <m && B[k-1] >= A[j];

        delta = cond_1 ? ((j-j_low-1)>>1) + 1 :
                cond_2 ? ((k-k_low-1)>>1) + 1 : delta;
        k_low = cond_1 ? k : k_low;
        j_low = cond_2 ? j : j_low;
        j     = cond_1 ? j - delta : cond_2 ? j+delta : j;
        k     = cond_1 ? k + delta : cond_2 ? k-delta : k;
        if(!cond_1 && !cond_2)
            break;
    }
    return j;
}
```

Listing 5.2: Co-rank Remove Code Divergence

Since naive parallel merge, single buffer parallel merge and double buffer parallel merge all use the co-rank function and merge function, we apply these two optimizations to all of them. The performance improvement after the optimizations is shown in Chapter 6.

Chapter 6

Evaluation

6.1 Methodology

6.1.1 Experiment Platform

We use OpenMP to implement the CMP parallel merge on an intel CPU, and CUDA to implement the GPU parallel merge on Nvidia GPUs. The CPU and GPUs we use are listed as follows, and their specifications are presented in table 6.1.

- CPU: Intel Core i7-4960 HQ Processor
- GPU: Nvidia Titan-Z, Kepler Architecture
- GPU: Nvidia GTX 980, Maxwell Architecture

Table 6.1: Specifications of CPU and GPUs

	Intel Core i7	Titan-Z	GTX 980
Theoretical Memory Bandwidth (GB/s)	25.6	336	224
Experimental Memory Bandwidth (GB/s)	NA	240	163
Peak Floating Point Calculation Rate (GFLOPS)	89.6	8122	4616
Number of Cores	4	5760	2048

6.1.2 Tuning Parameters

When launching the kernel, we can set the number of blocks, the block dimension, and shared memory size. These parameters will affect the performance. Moreover, the best combination

of these parameters on one architecture may not achieve the best performance on a different architecture. Therefore, we set these parameters as tunable.

- **number of blocks:** The number of blocks is related to the amount of work for each block. Increasing the number of blocks will create more parallelism that could be scheduled on different streaming multiprocessors. Meanwhile, the amount of work for each block will decrease. It is desirable that we have enough number of blocks to explore the parallelism, as well as enough amount of work for each block to amortize the block launching overhead. The number of blocks is chosen from 1, 4, 16, 64, 256, and 512.

- **block dimension:** Block dimension determines the number of threads in the thread block. We choose the block dimension from 128, 256, and 512.

- **shared memory size:** Shared memory size can affect the number of blocks that can run simultaneously on the streaming multiprocessors. Increasing shared memory size will increase the amount of data we merge in each iteration, and decrease the number of blocks that can be scheduled simultaneously on the streaming multiprocessors. It is desirable to merge enough data in each iteration, while keeping enough number of blocks scheduled on the streaming multiprocessors to hide latency. We choose shared memory size from 1024B, 2048B and 2560B.

We enumerate all the combinations of tuning parameters, and launch the same kernel multiple times with different combinations of tuning parameters. We record the running time of all the combinations, and choose the one that has the best performance. The best parameters are different for different input sizes and architectures.

The input sizes we use are 1K, 10K, 100K, 1M, 10M and 100M. The input size is the number of elements (integers) in an input array A. For example, if the input size is 100M, we will have 100M integers from A, 100M integers from B, and 200M integers from C. The total memory usage will be 1.6 GB. We start from 1K because this is a relatively small number to demonstrate the advantages of sequential merge and CMP merge over GPU merge implementations. We increase the input size until 100M. There are two reasons we didn't

go beyond 100M: (1) When the input size is 100M, the total memory usage is 1.6 GB. As we go beyond this size, we will possibly run out of memory. Thrust merge library will fail for larger input size by complaining: not enough memory. (2) Merge is a basic subroutine. It is only part of an application and cannot occupy too much memory. In [6], the maximum input size used is also 100M.

6.2 Results

6.2.1 Performance on CMPs

Figure 6.1 shows the performance of CMP parallel merge over sequential merge.

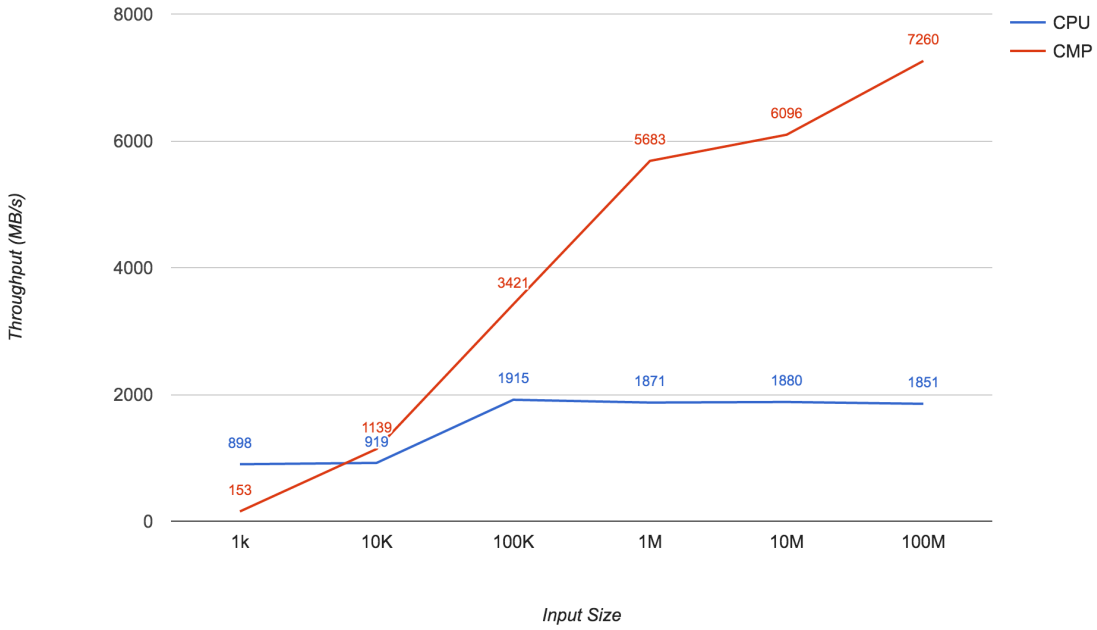

Figure 6.1: Performance of CMP Merge over Sequential Merge

The CPU we use has 8 threads. Ideally, the speedup of CMP parallel merge could achieve 8 compared to sequential merge. In reality, CMP parallel merge is slower than sequential merge when the input size is small. As the input size grows, parallel merge becomes faster, and can achieve a speedup of 5x compared to the sequential merge.

For the small input size, parallel merge is slower than sequential merge because the overhead of binary search from co-rank function dominates the actual merge. As the input size grows, the overhead of binary search from co-rank could be amortized, and parallel merge could outperform the sequential merge. However, the overhead of binary search cannot be neglected. Moreover, memory congestion may occur because the threads are issuing more memory requests when doing merge in parallel. Due to the non-negligible overhead and potential memory congestion, the actual speedup can achieve 5x instead of 8.

6.2.2 Performance on GPUs

We only count the kernel time for all the GPU implementations in our experiments. If the application is running on the device (GPU), we don't need to copy data between host and device because everything is on the device. If the application is running on the host side, we can use CUDA stream to transfer data asynchronously and hide the overhead of data transfers[3]. On the host side, we can run the co-rank function and partition the input data. Data transfers, including host to device transfer and device to host transfer, could be hide by the computation.

Figures 6.2 and 6.3 show the memory throughput of all implementations of parallel merge on Titan-Z and GTX 980 respectively.

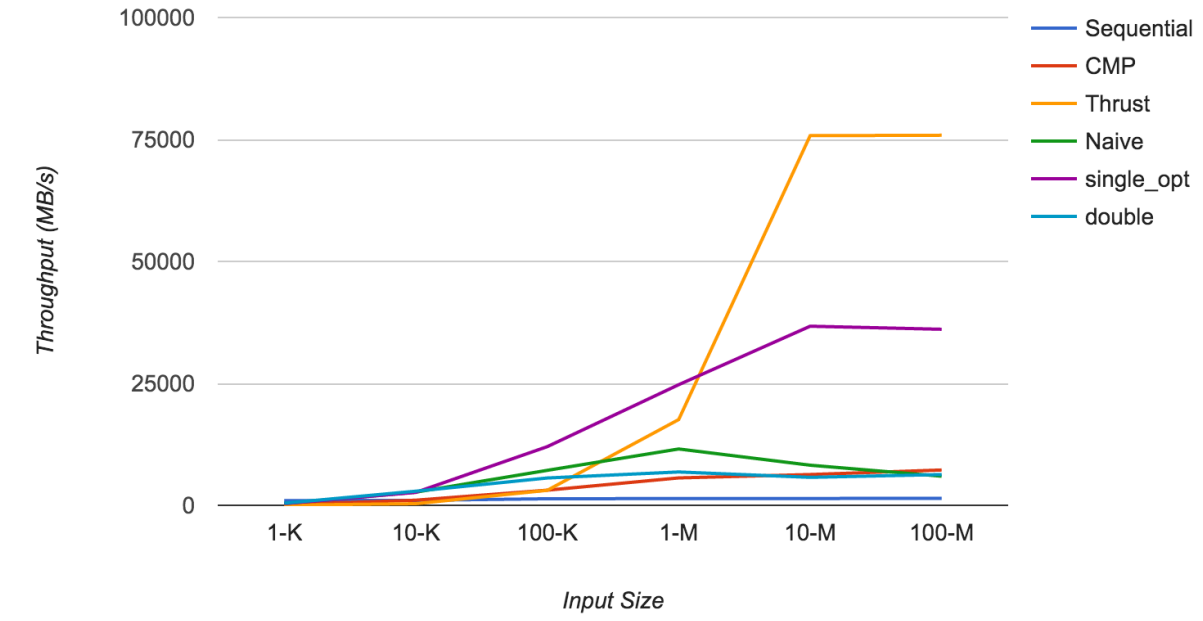

Figure 6.2: Throughput on Titan-Z

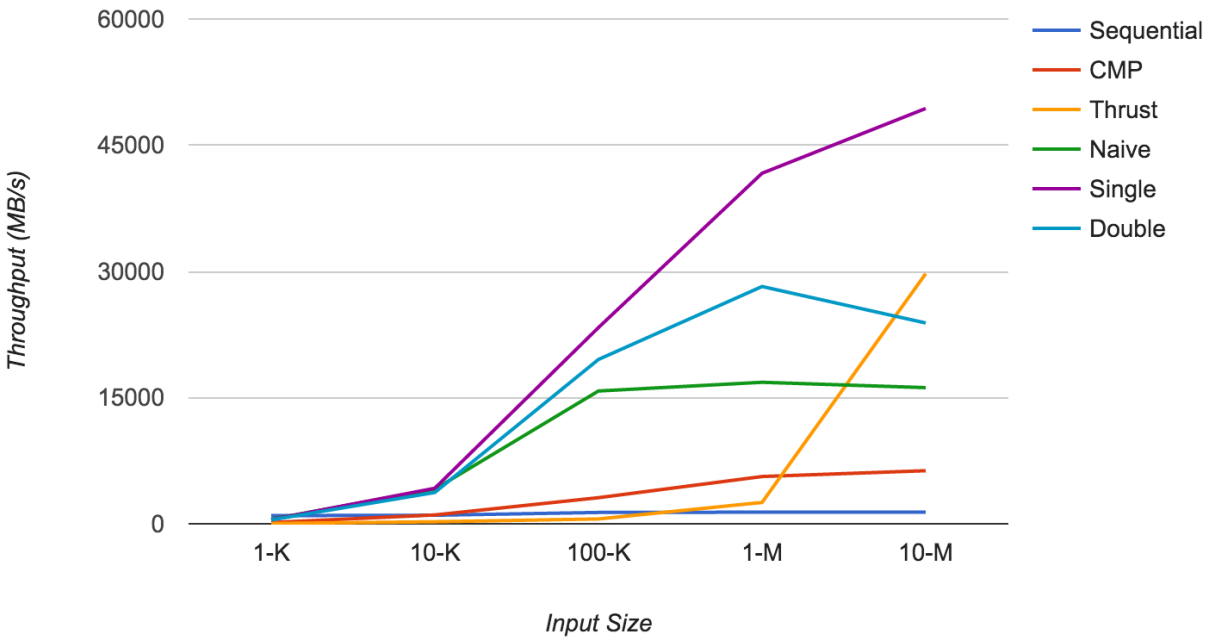

Figure 6.3: Throughput on GTX 980

In naive parallel merge, both the co-rank function and merge function run on the global memory. Due to the nature of the co-rank function, in which each thread is performing a

binary search for its own input range, the global memory accesses are not coalesced. The global memory accesses for merge function are not coalesced either. For these reasons, although naive parallel merge is faster than thrust merge for certain input size, it still underutilizes the memory bandwidth offered by GPU and therefore did not achieve the optimal overall performance.

In single buffer parallel merge, we use shared memory as a scratch pad. There are two benefits to using shared memory: (1) Global memory accesses are coalesced when we fill shared memory. This will better utilize the memory bandwidth on GPUs. (2) There will be fewer global memory requests. Using shared memory as scratch pad, the thread level co-rank function runs on shared memory, and the merge function reads the inputs from shared memory. In this way, we convert many expensive global memory reads to cheap and fast shared memory reads. Therefore, single buffer parallel merge outperforms naive parallel merge as expected.

In double buffer parallel merge, the benefits of using shared memory are preserved. Moreover, we utilize all the data loaded into shared memory, while in single buffer parallel merge, half of the input data are wasted. We expect that double buffer parallel merge will outperform single buffer parallel merge. However, the results in figures 6.2 and 6.3 show the opposite outcome. The reasons are: (1) In double buffer parallel merge, each thread merges fewer number of elements due to the limited shared memory size, so the overhead for searching could dominate the actual merge. (2) We observed that double buffer parallel merge used more registers than single buffer parallel merge, and caused the occupancy to drop. As a result, double buffer parallel merge is not as fast as single buffer parallel merge.

All the GPU implementations we have are significantly faster than sequential merge for large input sizes. Among the three implementations we have, single buffer parallel merge has the best performance. In figures 6.4 and 6.5 we show the speedup of single buffer parallel merge over thrust merge implementation.

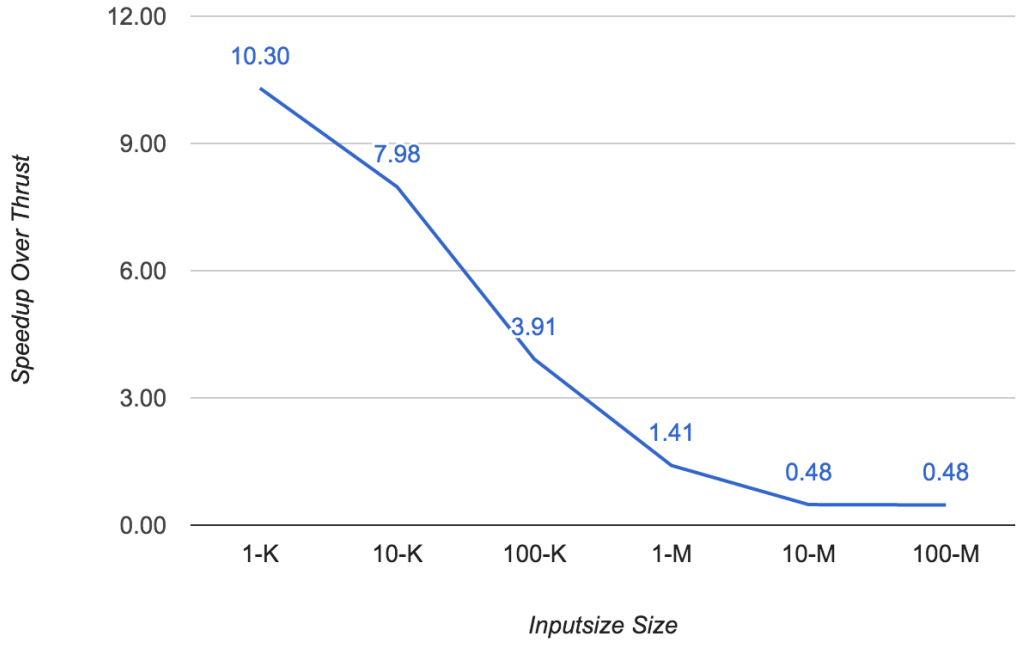

Figure 6.4: Speedup on Titan-Z

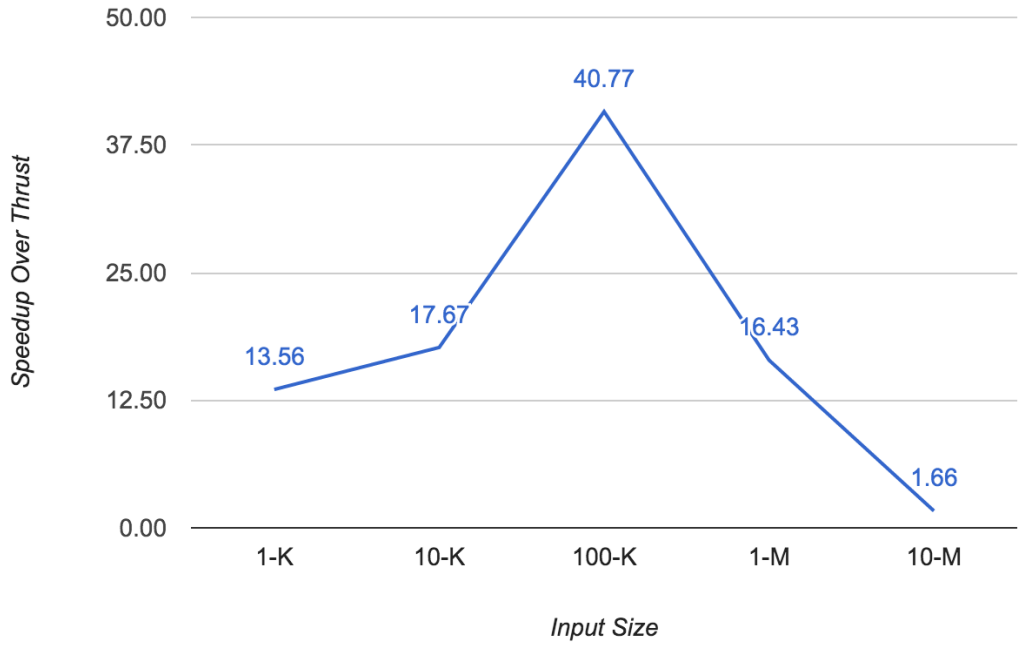

Figure 6.5: Speedup on GTX 980

On Titan-Z, single buffer parallel merge can achieve up to 10x speedup compared to thrust merge implementation. On GTX 980, single buffer parallel merge can achieve up to 40x speedup compared to thrust merge implementation.

Single parallel merge has a larger speedup on GTX 980 than on Titan-Z. The reason is that thrust merge is not specifically optimized for Maxwell Architecture and therefore its performance is far from optimal on this platform.

In Figures 6.6 we show the performance improvement after applying the optimizations we present in Chapter 5. Reducing number of calls to co-rank function and changing code divergence to memory divergence contribute another 1.5x speedup for single buffer parallel merge on Titan-Z.

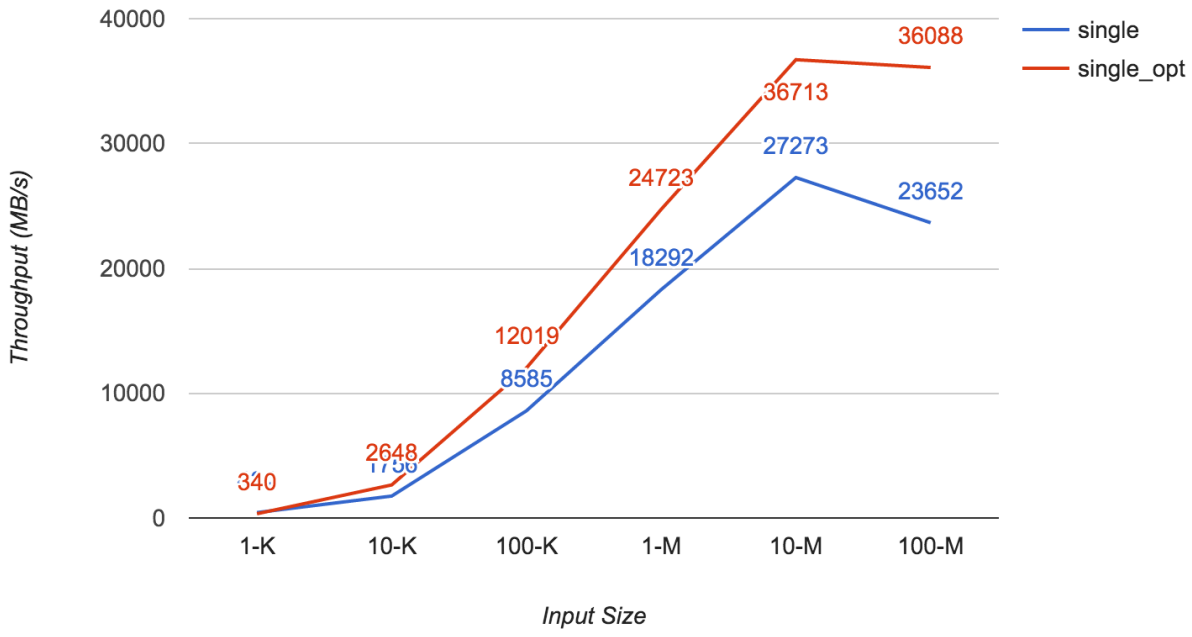

Figure 6.6: Performance Improvement after Optimizations

Chapter 7

Conclusion

In this thesis, we implemented three versions of parallel merge on GPUs with different levels of GPU-specific optimizations: naive parallel merge, single buffer parallel merge and double buffer parallel merge.

We set up the auto tunable parameters, and chose the best tuning parameters for different input sizes and GPU architectures. Evaluations on Titan-Z and GTX 980 showed that single buffer parallel merge had the best performance among the three versions.

All the GPU implementations are significantly faster than sequential merge for large input sizes. Single buffer parallel merge achieved up to 10x and 40x speedup on Titan-Z and GTX 980 respectively compared to thrust merge implementation.

References

[1] C. Siebert and J. L. Träff, "Perfectly load-balanced, optimal, stable, parallel merge," arXiv preprint arXiv:1303.4312, 2013.

[2] N. Fauzia, L.-N. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on GPUs," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2015, pp. 12–22.

[3] D. B. Kirk and W. H. Wen-mei, *Programming Massively Parallel Processors. A Hands-on Approach*. Newnes, 2012.

[4] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007, pp. 407–420.

[5] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications," *Multimedia, IEEE Transactions on*, vol. 15, no. 2, pp. 279–290, 2013.

[6] O. Green, R. McColl, and D. A. Bader, "GPU merge path: a GPU merging algorithm," in *Proceedings of the 26th ACM International Conference on Supercomputing*. ACM, 2012, pp. 331–340.