PARALLEL MERGE ON GPU

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 array has a key. An ordering relation denoted by $\leq$ is defined on the keys. Array A and B has m and n elements respectively. m and n don't have be to 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 B, and is sorted by the ordering relation.

We applied several GPU specific optimizations to the parallel merge algorithm. The optimizations include coordinating the memory access pattern, making full use of the shared memory and reducing the thread divergence. Out implementation achieves more than 10x speedup compared to sequential merge, and at most 20x speedup compared to thrust merge implementation.

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

Table of Contents

List of Figures

Chapter 1

Introduction

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 B has m and n elements respectively. m and n don't have be to 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 B, and is sorted by the ordering relation.

Figure 1.1 gives an example of mering two arrays of integers.

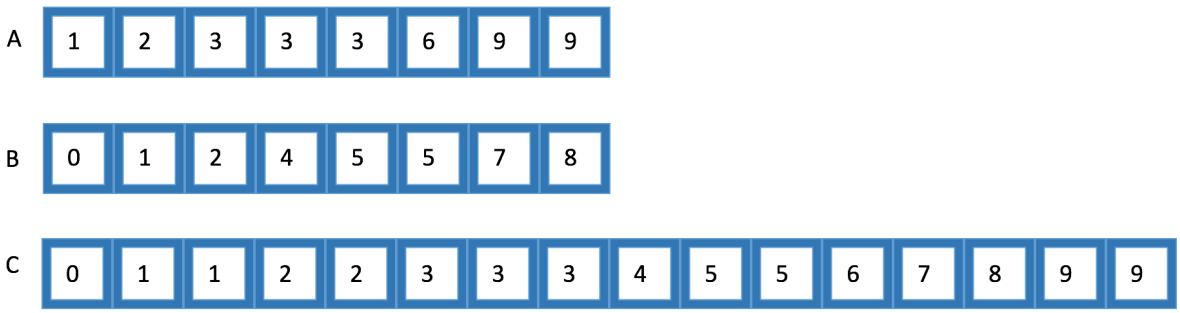

Figure 1.1: Merge Example

Sequentially merging two sorted arrays has been solved for a long time. Listing 1.1 shows the sequential code implemented in C++. The time complexity of this implementation is $O(m + n)$.

```
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 remainging 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 1.1: Sequential Merge Implementation in C++

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

As single-chip multiprocessor(CMP) becomes more and more popular, parallel merging algorithm is also developed to exploit the performance brought by single-chip multiprocessor. In the paper "*Perfectly load-balanced, optimal, stable, parallel merge*" [1], Siebert et al.

proposed a parallel merge algorithm. With p processing elements, the time complexity of merge could be reduce from $O(m+n)$ to $O(\frac{m+n}{p} + \log min(m,n))$. This parallel merge algorithm can be implemented on single-chip multiprocessor using openMP with minimum effort, and achieve considerable speedup compared to to sequential merge.

However, a direct implementation of this parallel merge algorithm on GPU will result in suboptimal performance due to the architecture difference between GPU and single-chip multiprocessor. To exploit the massive parallelism on GPU, we need to coordinate the memory access pattern, make full use of the shared memory and reduce the thread divergence. This thesis proposes a novel GPU implementation for merging two sorted arrays. To the best of our knowledge, **thrust library** is the fastest GPU merge implementation. Our implementation achieves 5x speedup compared to the thrust library for certain input size.

The rest of the paper is organized as follows: Motivation for improving the performance of merge is presented in Section 2. GPU architecture and global memory coalescing are introduced in Section 3. The parallel merge algorithm[1] is described in Section 4. Section 5 describes the GPU implementation of the parallel merge algorithm and all the optimizations. Evaluation is present in Section 6. Finally, Section 7 concludes.

Chapter 2

Motivation

Merge is a very important operation. It is used as a subroutine by many popular algorithm and applications such as merge sort and database operations. As a frequently used subroutine, the performance of merging is critical.

The time complexity of sequentially merging two sorted arrays is $O(m+n)$. As the need for high performance computing grows, single-chip multiprocessor becomes more and more popular. However, most existing sequential algorithm couldn't fully utilize the computing resource on single-chip multiprocessor. To exploit the performance of single-chip multiprocessor, parallel algorithm must be developed.

Siebert et al.[1] proposed a parallel merge algorithm. With p processing elements, the time complexity of merge could be reduce from $O(m+n)$ to $O(\frac{m+n}{p} + \log min(m,n))$ [1]. This algorithm can be implemented on single-chip multiprocessor using openMP with minimum effort, and achieve considerable speedup compared to sequential merge.

Compared to CPU, GPU has more cores and larger memory bandwidth. For data parallel applications, GPU has much better performance than CPU. Merge can also run on GPU. To our best knowledge, thrust library has the fastest GPU merge implementation. However, its throughput is still far from the memory bandwidth.

We implement Siebert's parallel merge algorithm(we call it Naive Parallel Merge) and compared its performance to the thrust library. The memory throughput for different input size is shown in figure 2.1.

Although naive parallel merge is suboptimal due to the lack of GPU specific optimizations, we observe that for some input size(from 1k to 1M), its performance is better than thrust library.

This motivates us to do further optimizations for naive parallel merge, and find a better

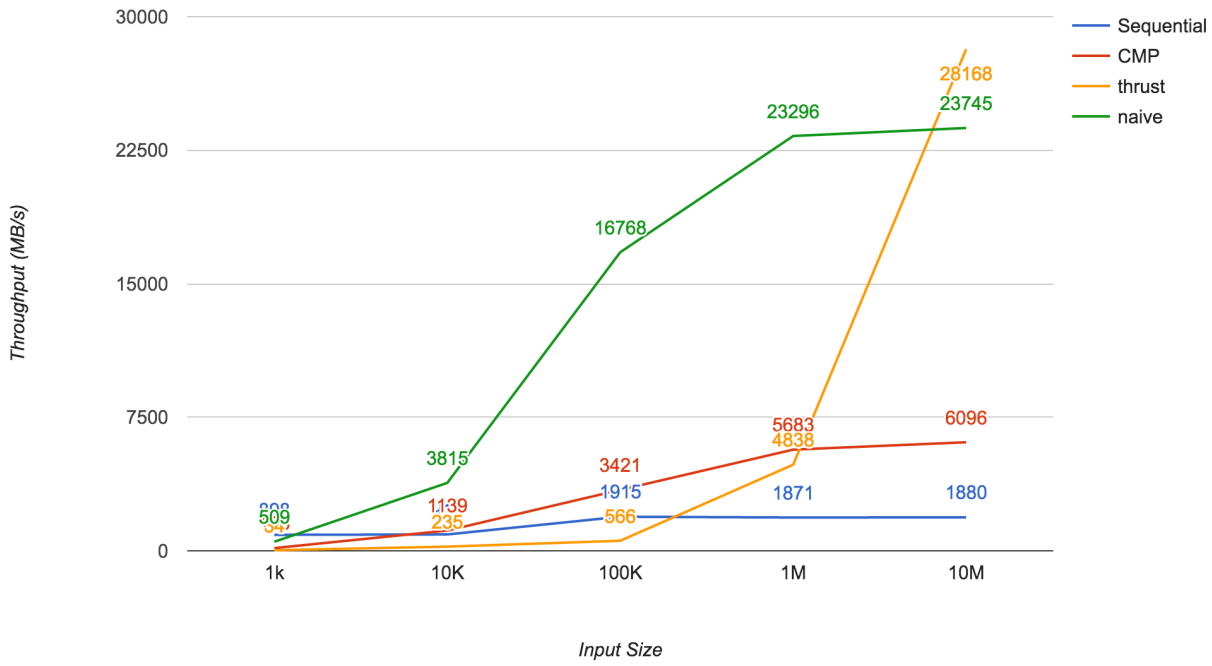

Figure 2.1: Performance of Thrust Merge and Naive Parallel Merge on GTX980

GPU merge implementation that outperforms thrust library.

Chapter 3

GPU Architecture and Global Memory Coalescing

3.1 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 utilize the huge number of cores on 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 48KB. Registers are the fastest but have the smallest size.

3.2 Baseline Architecture

The CPU and GPUs and their specifications are listed as follows:

- CPU: Intel Core i7-4960HQ Processor, 2.6GHz
- GPU: Nvidia Titan-Z, Kepler Architecture
- GPU: Nvidia GTX980, Maxwell Architecture

3.3 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 3.1 shows an example of global memory reference.

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

Listing 3.1: 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 3.1), 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 3.1, 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 the shared memory as the scratch pad. This approach requires complex code restructuring, and is one of the GPU specific optimization that we use to improve the performance of parallel merge.

Chapter 4

Parallel Merging Algorithm

In their paper "*Perfectly load-balanced, optimal, stable, parallel merge*"[1], Siebert et al. proposed a parallel merge algorithm.

In the parallel merge algorithm, each processing element will calculate the output range it is going to produce, and use that output range as the input to a **co-rank function** to identify the corresponding input range that generate the output. Finally, each processing element will call the sequential merge function to do the merge independently in parallel.

4.1 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 are unique. j and k define the prefixes of A and B needed to produce the prefix of C of length i. 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 number of elements in the input arrays. Figure 4.1 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.

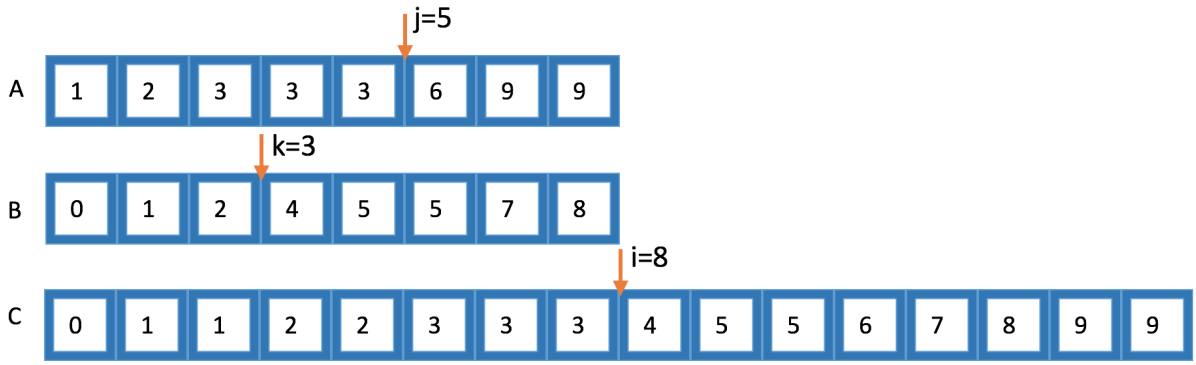

Figure 4.1: Co-rank Example

The pseudo code to find the co-rank from A, m, B, n and i is also given in their paper. In listing 4.1, 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 4.1: Original Co-rank

4.2 Overall Parallel Merge

The co-rank function provides a simple and efficient way of performing 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 will calculate 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 range of the input arrays this processing element needs to merge sequentially.

Listing 4.2 shows the pseudo code for the 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 4.2: Pseudo Code for Parallel Merge Algorithm

Figure 4.2 shows an example of the parallel merge process. In this example, there are two processing elements($p = 2$). These two processing elements are going to do the task $C[16] = merge(A[8], B[8], \leq)$ collaboratively in parallel.

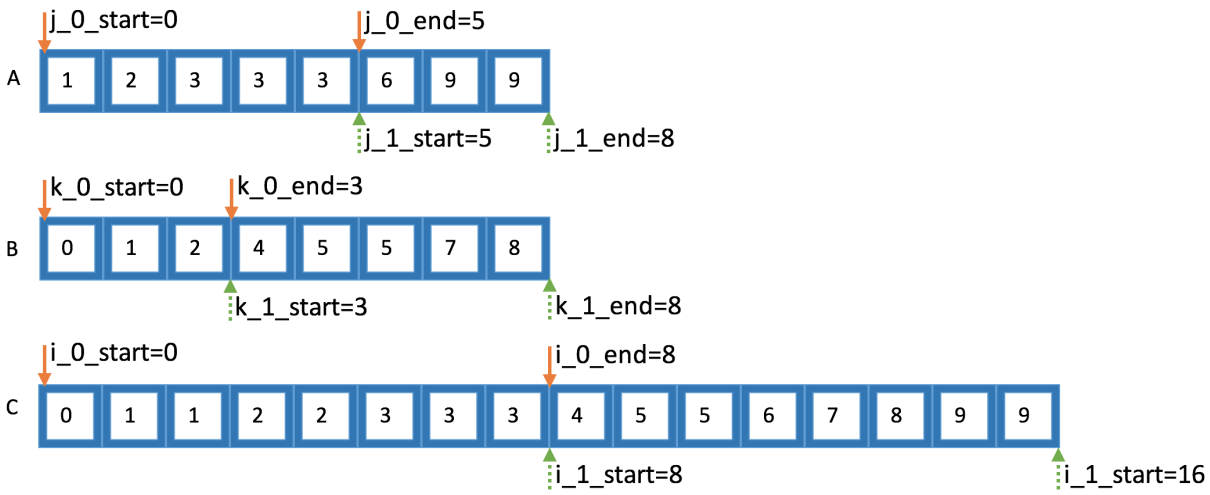

Figure 4.2: Parallel Merge Algorithm Example

For p_0(the solid red arrows), $r = 0$, $i\_start = 0$, $i\_end = 8$. It 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(the dash green arrows), $r = 1$, $i\_start = 8$, $i\_end = 16$. It is going to produce $C[8, ..., 15]$. After running the co-rank function, p_1 knows the input ranges: $j\_start = 5$, $j\_end = 8$, $k\_start = 3$, $k\_end = 8$. 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 part of 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.

4.3 Implementation on CPU

We implement the parallel merge algorithm on single-chip multiprocessor using openMP. The CPU we are using has 8 threads. Each thread first calculates the output range it is going to produce. Then it identifies the corresponding input ranges using the co-rank function. Finally, does the merge in parallel by calling the sequential merge. Ideally, the speedup of an 8 thread CPU could achieve 8. In reality, the 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 speed up 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 can't 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 speed up can achieve 5x instead of 8. Figure 4.3 shows the performance of CMP parallel merge over sequential merge.

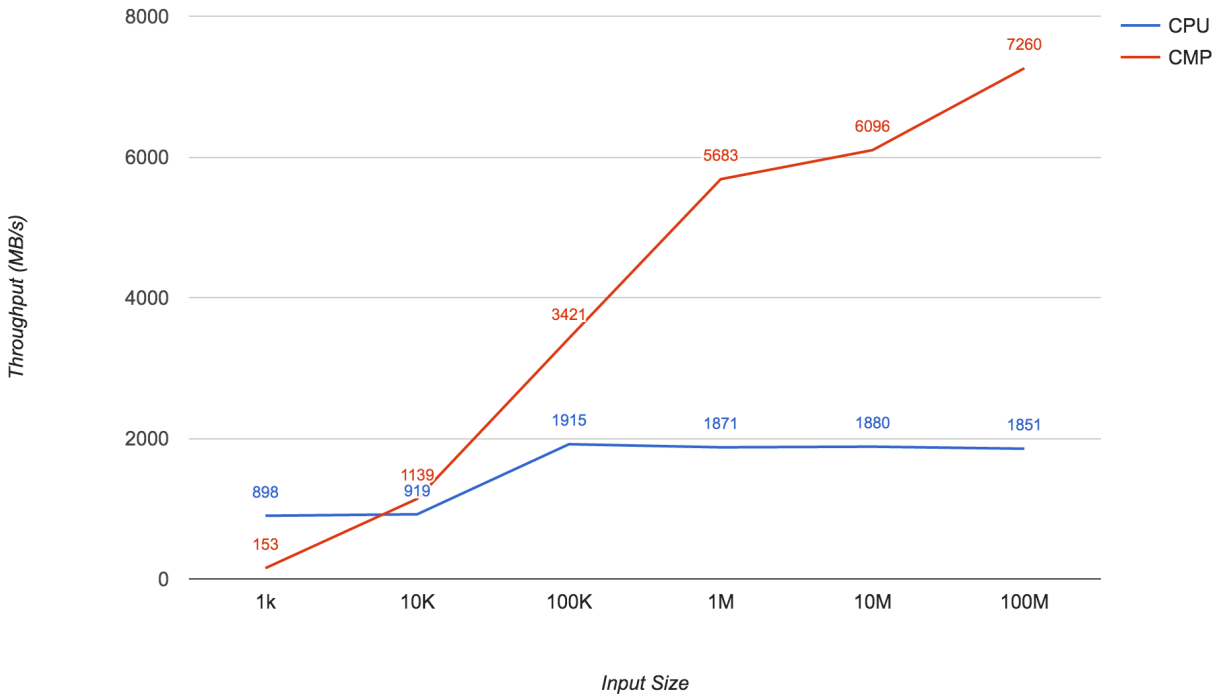

Figure 4.3: CMP Merge and Sequential Merge

Chapter 5

Parallel Merging Algorithm Implementation on GPU

Compared to single-chip multiprocessor, GPU has more threads and larger memory bandwidth for the purpose of massive parallelism. However, a direct translation of an parallel algorithm that suit on single-chip multiprocessor may not run efficiently on GPU. To explore the massive parallelism on GPU, 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.

As the first step, we implement this parallel merge algorithm on GPU without any GPU specific optimization. We name it **naive parallel merge**. More details about naive parallel merge is described in section 5.1.

However, the naive parallel merge didn't incorporate any GPU specific optimizations, which is critical to improve the application performance that runs on GPU. We implement an optimized 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**. More details about single buffer parallel merge is described in section 5.2.

In single buffer parallel merge, we only consume a 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 name it **double buffer parallel merge**. We describe more details about double buffer parallel merge in section 5.3.

5.1 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 id(r) and total number of threads(p)

using $t\_id = blockIdx.x * blockDim.x + threadIdx.x$ and $t\_num = gridDim.x * blockDim.x$ respectively. Based on t_id and t_num, 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 can start to merge independently by calling the sequential merge function to in parallel and write the result to $C[\,]$ in device global memory. Finally, we copy $C[\,]$ from device global memory back to the host.

Figure 5.1 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 will run in parallel and produce the result for the entire output array $C[\,]$.

Figure 5.1: Naive Parallel Merge

The performance of naive parallel merge on titan-z is shown in Figure 5.2.

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 doing a binary search for its own input index, the memory access pattern is not coalesced. The memory access pattern for merger function is not coalesced neither. For these reasons, although naive parallel merge is faster than thrust merge for certain input size, it still under utilizes the memory bandwidth offered by GPU and therefore didn't achieve the optimal overall performance.

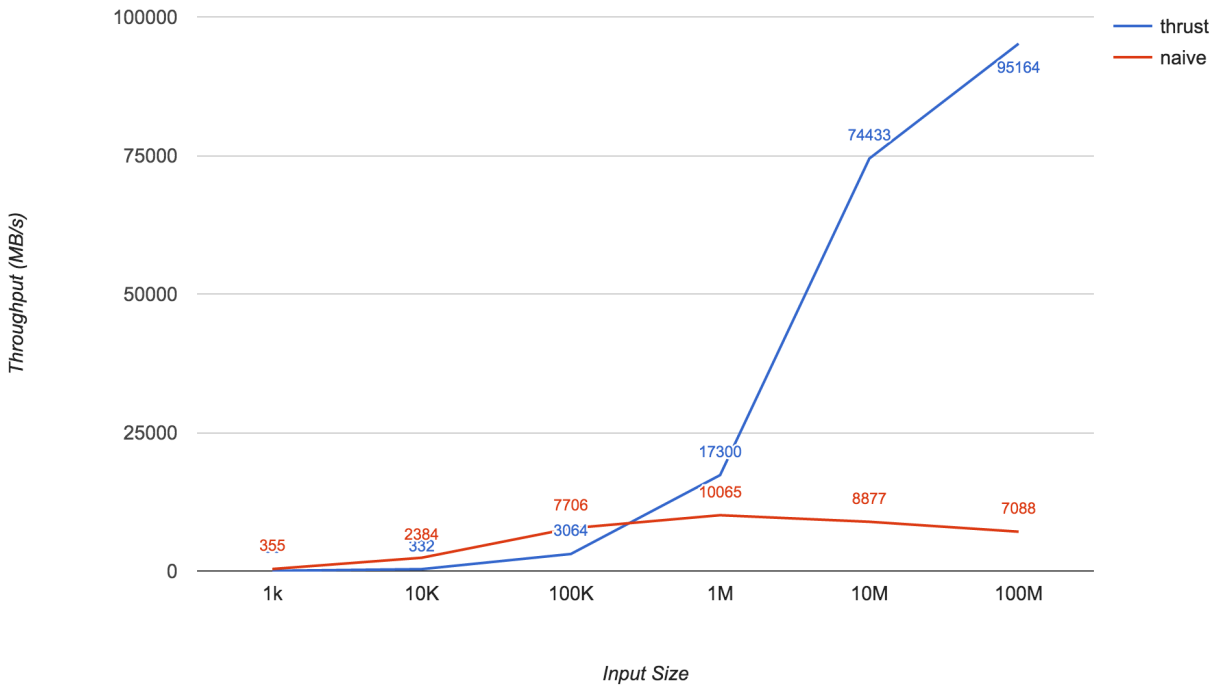

Figure 5.2: Performance of Naive Parallel Merge on Titan-Z

5.2 Single Buffer Parallel Merge

The memory access patter in naive parallel merge is not coalesced. To improve the memory throughput on GPU, coalesced global memory access pattern is critical. For this reason, we use shared memory on GPU to 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. In the block level, all the threads in the same block does the same searching. Each thread calculates the block id 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 out 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, the 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 the 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 a half of the data we load into the shared memory.

Then we run the co-rank function in 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 id and total number of threads in the block using $t\_id = threadIdx.x$ and $r\_num = blockDim.x$ 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 can start their work independently and call 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. Figure 5.3 shows the first iteration of single buffer parallel merge. The solid orange box is the block level run of co-rank function. The dash 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.

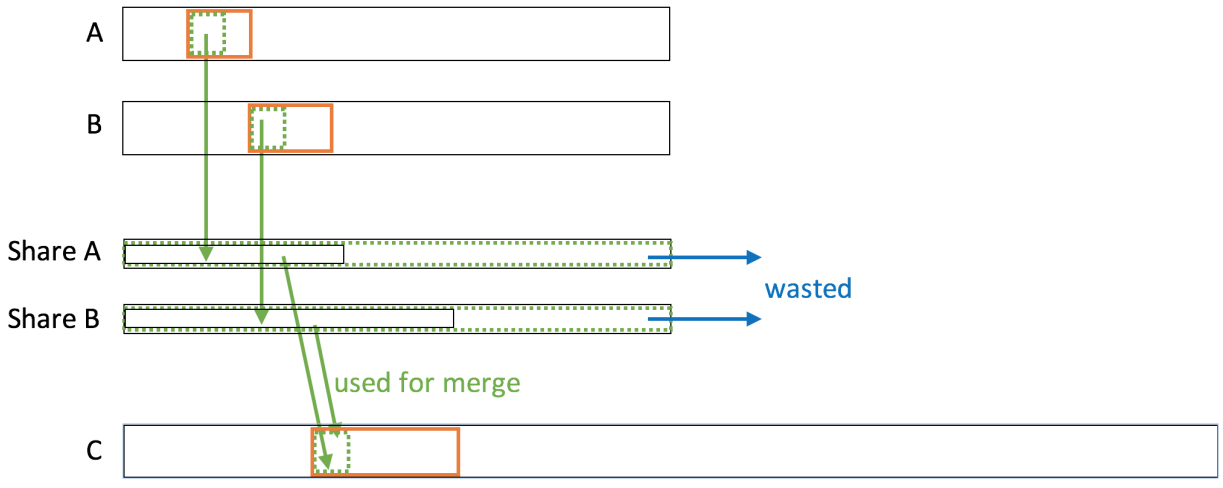

Figure 5.3: First Iteration of Single Buffer Parallel Merge

In the next iteration, we will load the data we haven't merged into the shared memory, and do the same merge. Figure 5.4 shows the second iteration of single buffer parallel merge. The dash red box is the second iteration of the thread level run of co-rank function.

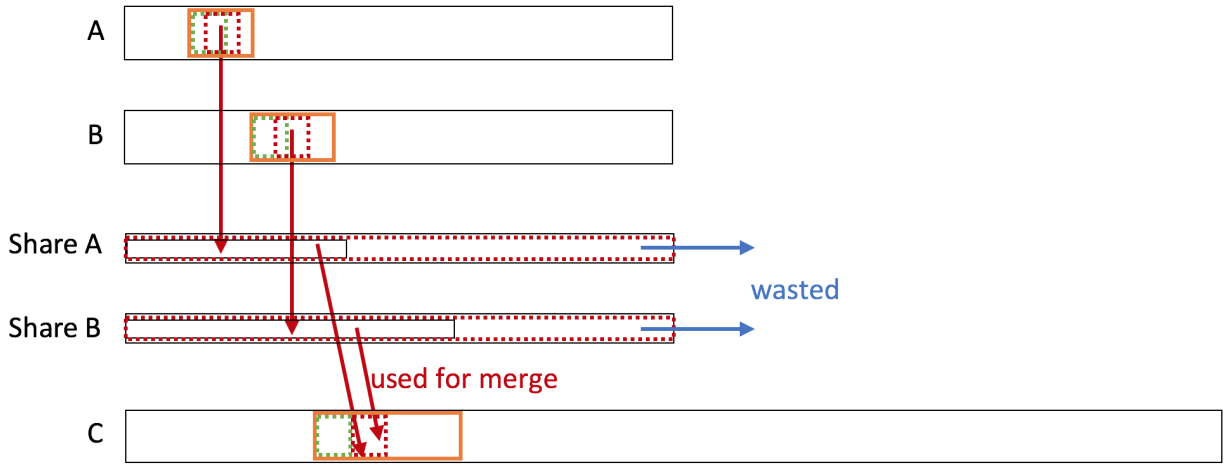

Figure 5.4: Second Iteration of Single Buffer Parallel Merge

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

In single buffer parallel merge, we fill the shared memory in a coalesced pattern so that there will be fewer global memory requests. The thread level co-rank function runs on shared memory. The merge function reads the input from shared memory, and writes to global memory. Because we convert many expensive global memory read to cheap shared memory read, the single buffer parallel merge could better utilize the memory bandwidth and hence outperform the naive parallel merge by 5x.

The performance of single buffer parallel merge is evaluated in Figure 5.5.

5.3 Double Buffer Parallel Merge

In single buffer parallel merge, we only consume a half of the data we load into the 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 the similar to single buffer parallel merge except the way how we use shared memory. Co-rank function is run in two levels, block level and thread level. In the block level, all the threads in the same block does the same searching. Each thread calculates the block id 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 out 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.

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 5.6 shows the initialization process.

In each iteration later, 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 don't load from global memory. Figure 5.7 and 5.8 show an example 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 didn't load from the global memory to shared memory.

If there are less than x elements in either shared array A or shared array B, we will load another x elements the into shared memory. Figure 5.9 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 in thread level. The co-rank function is run on shared memory. Each thread calculates the thread id and total number of threads in the block using $t\_id = threadIdx.x$ and $r\_num = blockDim.x$ 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 can start their work independently and call 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(fill the entire solid orange box).

Notice that the memory access pattern of double buffer parallel merge is coalesced. In double buffer parallel merge, we utilize all the data we load into shared memory, while in single buffer parallel merge, we waste a half of the data we load into shared memory. We expect that double buffer parallel merge will outperform single buffer parallel merge. However, in experiments, we observed that double buffer used more registers(39) than single buffer parallel merge(31), and cause the occupancy to drop. As a result, the double buffer parallel merge is not as fast as single buffer parallel merge.

The performance of double buffer parallel merge is evaluated in Figure 5.10.

5.4 Further Optimizations

5.4.1 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. One for the start point of the output range, and one for the end point of the output range. Figure 5.11 gives a 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.

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 4.2, 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 5.12. 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, 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 5.12) and reduce the number of calls by a half.

5.4.2 Change Code Divergence to Memory Divergence

Code divergence also hurts the performance of GPU application. When the threads in the same warp take different branches for an $if - else$ statement, code 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 minimum amount of code divergence to achieve high performance. In parallel merge, we are concerned about two functions: merge and co-rank.

- Merge

  Listing 1.1 shows the original sequential merge code. It uses a while loop to do the merge. It first merges A and B to C, and than copy the remaining A or B to C. We can see that the original merge has code divergence due to **if** and **else** statements inside the while loop. To remove the code divergence, we use **selection** to replace the **if** and **else** statements. Listing 5.1 shows the sequential merge code after removing code divergence. After this optimization, we effective replace all the code divergence with memory divergence.

- Co-rank

  Listing 4.1 shows the original code for co-rank function. The code divergence also comes from the **if-else** statements. We replace the **if** and **else** by **selection**. The resulting code is shown in Listing 5.2.

```
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

```
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

After removing code divergence, we see some speedup.

Here is a figure.

5.5 Tuning parameters

We set the number of blocks, the block dimension, and shared memory size as our tuning parameters.

- **number of blocks:** When we change the number of blocks, we are also changing the amount of work for each block. When we increase the number of blocks, we create more parallelism that could be scheduled on different SM. Meanwhile, the amount of work for each block is decreasing. We want the amount of work for each block to be large enough to amortize the block launch overhead. We choose the number of blocks 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 SM. We choose from 1024B, 2048B and 2560B.

We enumerate all the combinations of tuning parameters, record their running time, and choose the one that has the best performance.

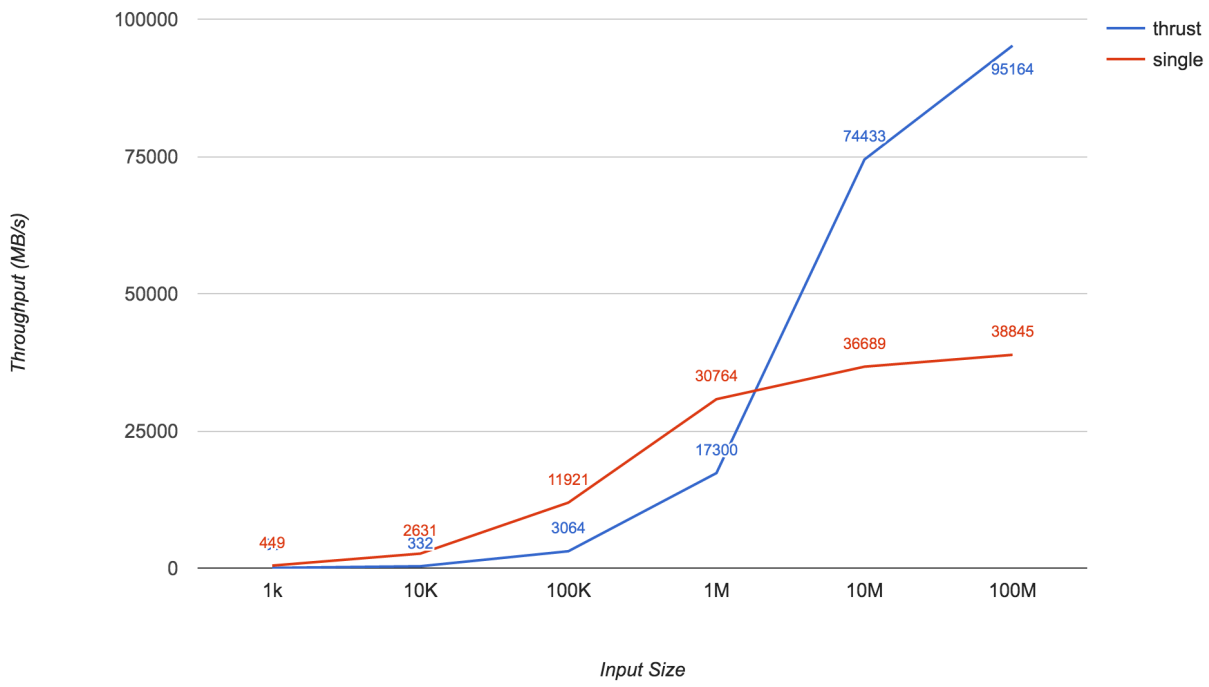

Figure 5.5: Performance of Single Buffer Parallel Merge on Titan-Z

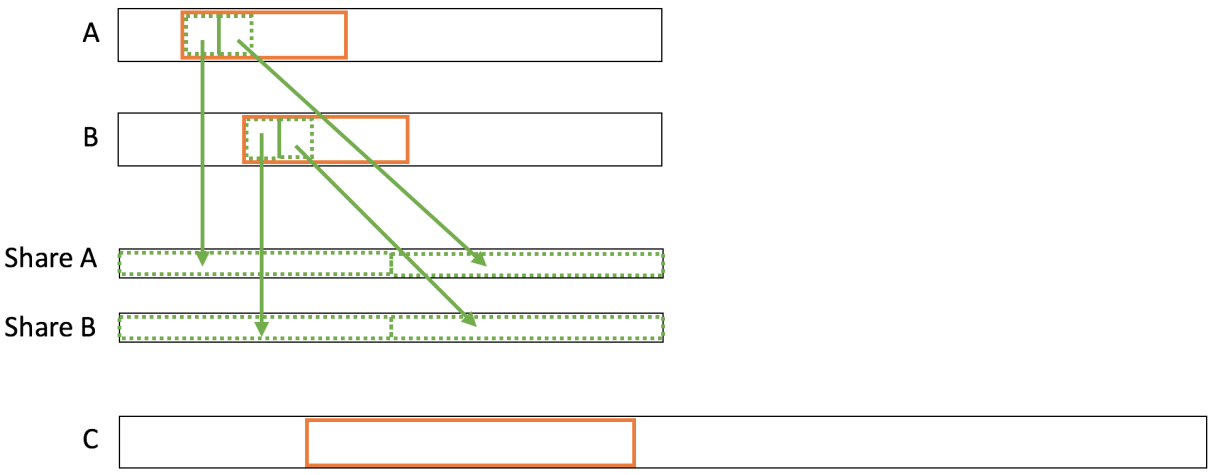

Figure 5.6: Initialization of Double Buffer Parallel Merge

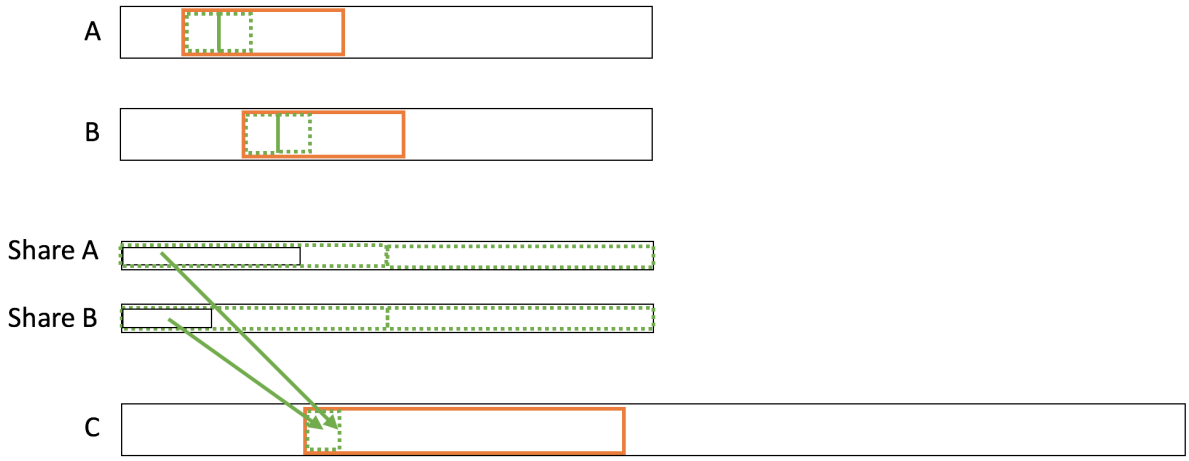

Figure 5.7: First Iteration of Double Buffer Parallel Merge

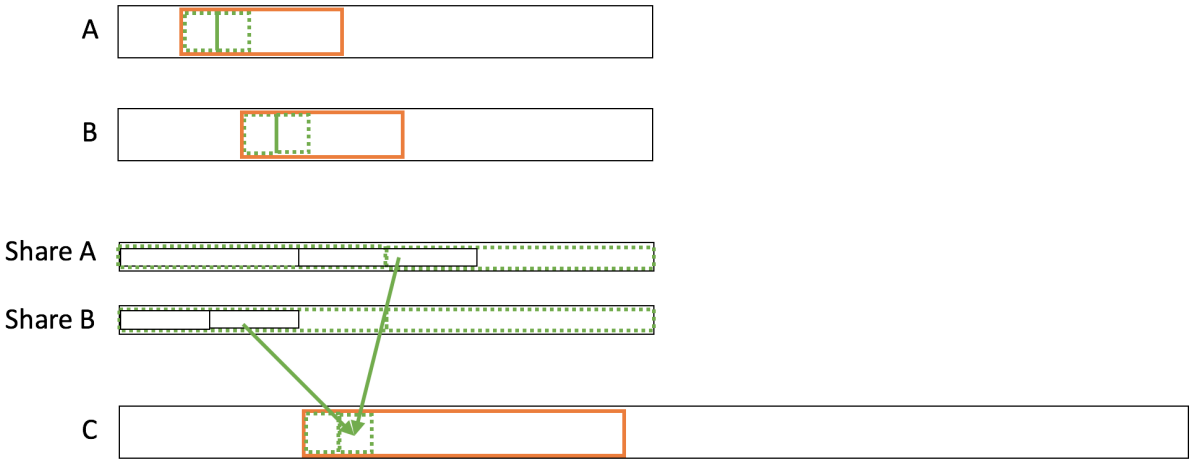

Figure 5.8: Second Iteration of Double Buffer Parallel Merge

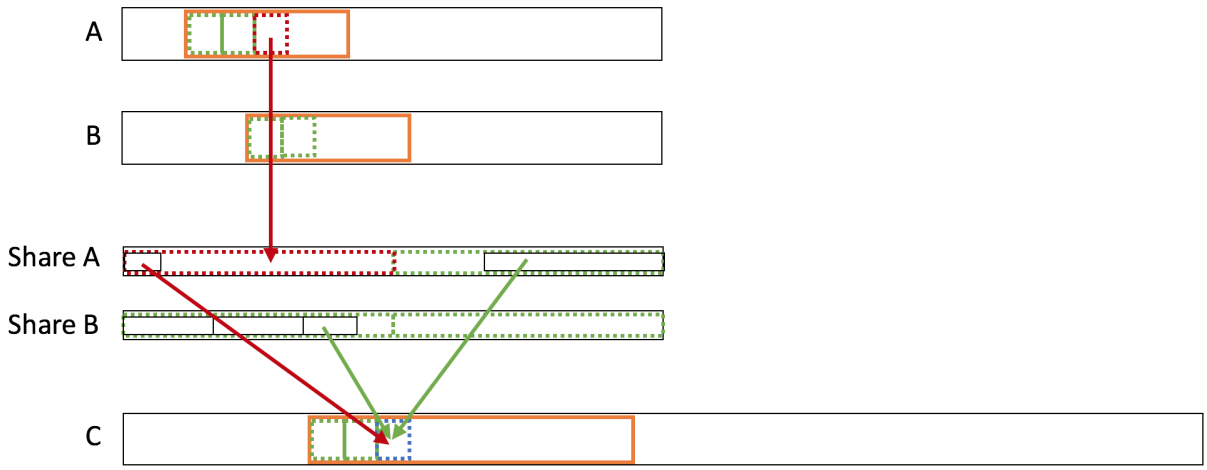

Figure 5.9: Third Iteration of Double Buffer Parallel Merge

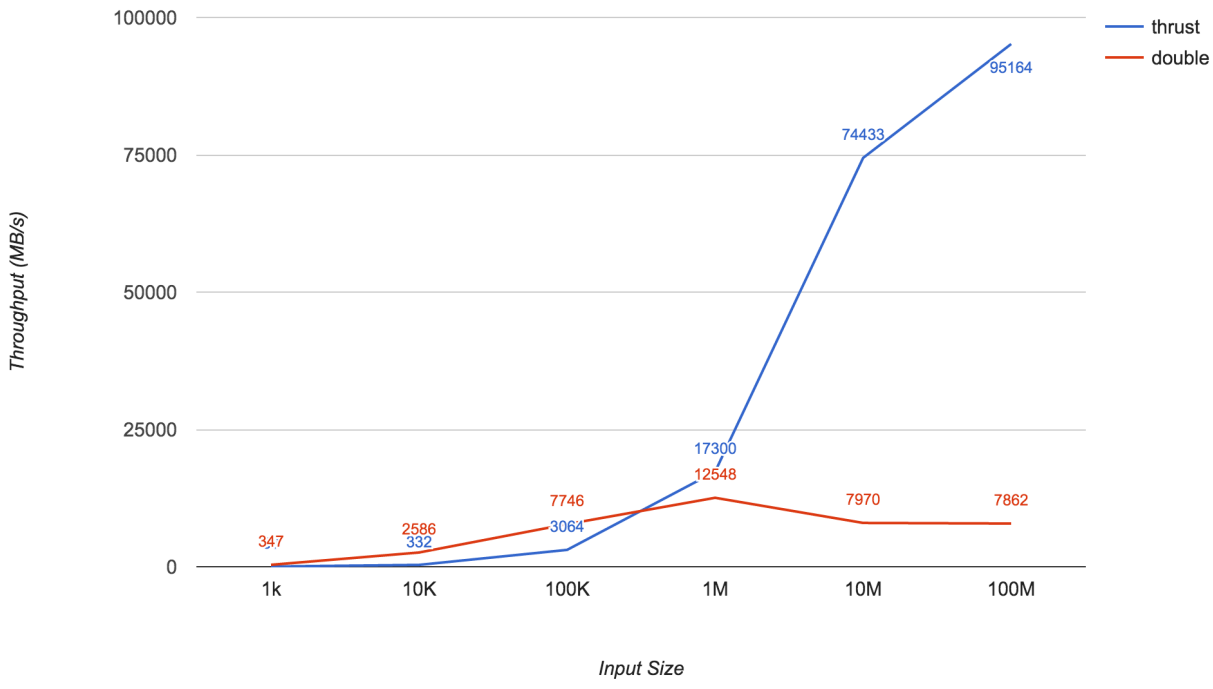

Figure 5.10: Performance of Double Buffer Parallel Merge on Titan-Z

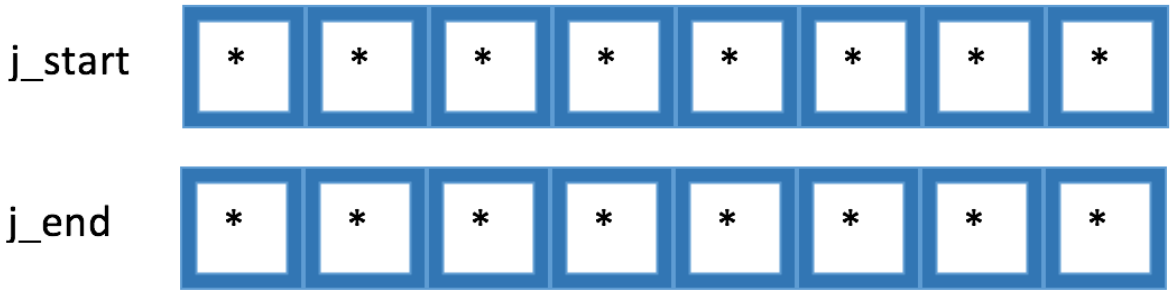

Figure 5.11: Number of Calls to Co-rank Function

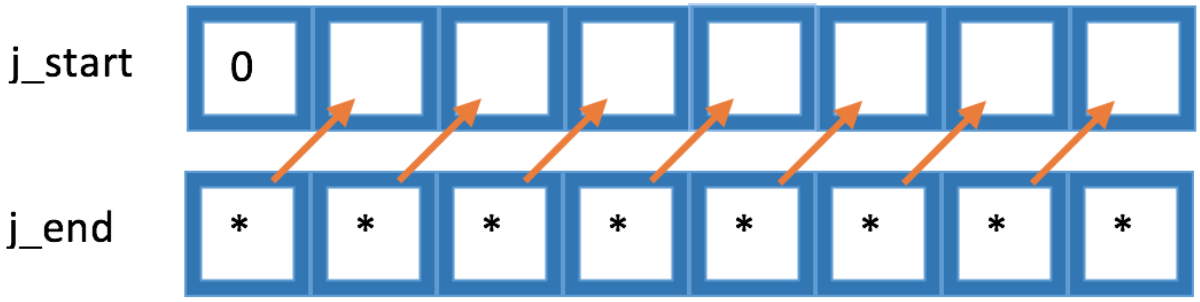

Figure 5.12: Number of Calls to Co-rank Function after Optimization

Chapter 6

Evaluation

We put the performance of all the parallel merge implementations on Titan-Z in Chapter 5 in Figure 6.1.

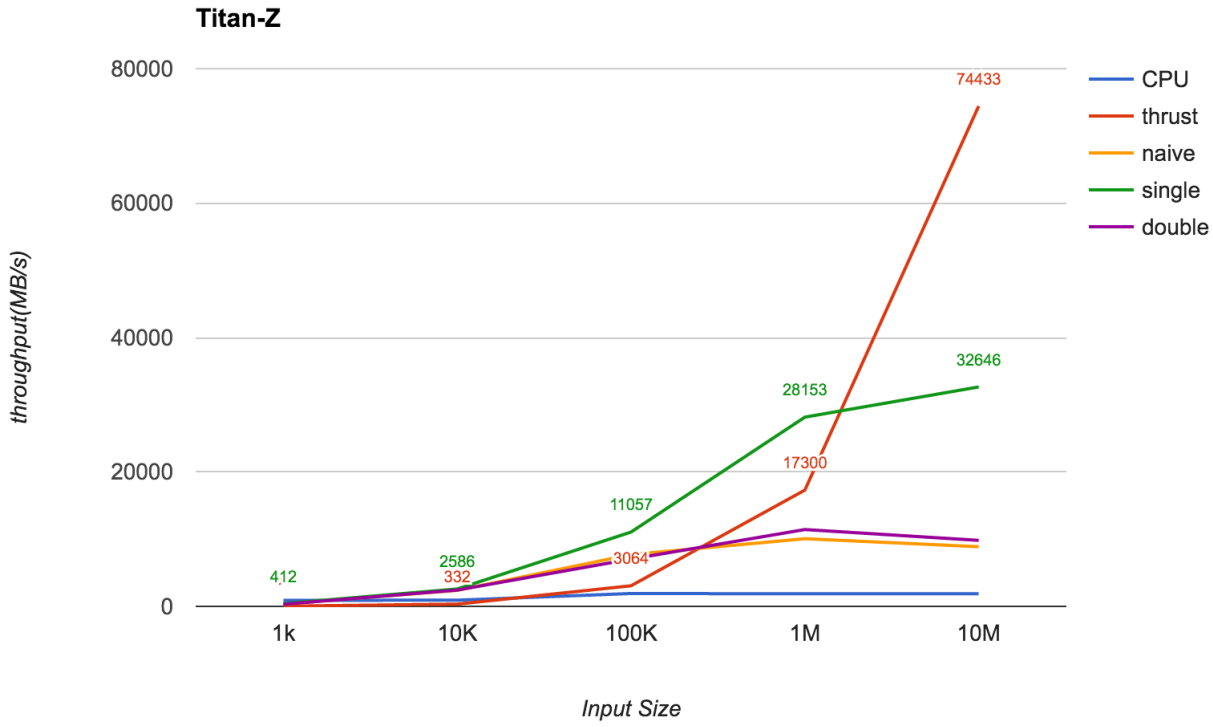

Figure 6.1: Performance on Titan-Z

Among the three implementations we have, single buffer parallel merge has the best performance. Compared to thrust merge implementation, single buffer parallel merge can achieve up to 3x speedup for input size between 10k to 1M.

We also test the portability of our implementations on a Nvidia GTX980 GPU(Maxwell Architecture). Figure 6.2 shows the performance.

Single buffer parallel merge outperform the other two implementations as well. Compared to thrust merge, single buffer parallel merge can achieve up to 20x speedup. The reason is that thrust merge is not specifically optimized for Maxwell Architecture and therefore its performance is far from optimal on this architecture.

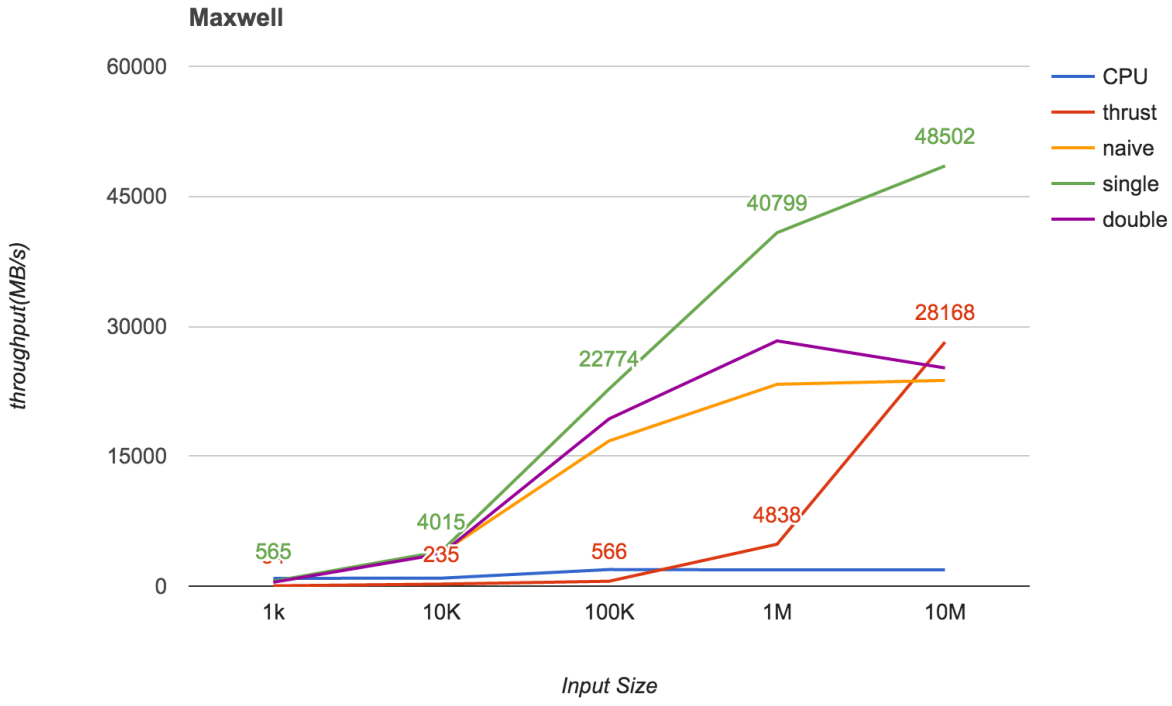

Figure 6.2: Performance on GTX980

Chapter 7

Conclusion

In this thesis, we implemented three versions of parallel merge algorithm on GPU: naive parallel merge, single buffer parallel merge and double buffer parallel merge.

We evaluate their performance on different GPUs. Among the three versions, single buffer parallel merge has the best performance. Compared to sequential merge, single buffer parallel merge achieves 10x speedup. Compared to thurst merge implementation, single buffer parallel merge achieves at most 20x speedup for certain GPU architecture.

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.