Two Phase Multiway Merge Sort and new GPU merging algorithm on GPU

Kevin Yang, 50541650
[1]Computer Science, Arkansas state University, Jonesboro, Arkansas, USA

Abstract - *Graphics Processing Units (GPUs) have become ideal candidates for the development of fine-grain parallel algorithms as the number of processing elements per GPU increases. In addition to the increase in cores per system, new memory hierarchies and increased bandwidth have been developed that allow for significant performance improvement when computation is performed using certain types of memory access patterns. Merging two sorted arrays is a useful primitive and is a basic building block for numerous applications such as joining databases queries, merging adjacency lists in graphs, and set intersection. An efficient parallel merging algorithm partitions the sorted input arrays into sets of non-overlapping sub-arrays that can be independently merged on multiple cores. For optimal performance, the partitioning should be done in parallel and should divide the input arrays such that each core receives an equal size of data to merge. Traditional sort algorithm requires that the file/data must fit in the main memory to be sorted. The two-phase multiway merge sort (TPMMS) algorithm provide sort file/data larger than the main memory. That is more closer to reality situation such as using in big data.*

1 Algorithm

In this section, I use two algorithms to implement the code runing on GPU. TPMMS has two Phase, the Phase 1 is typically divide an array into main memory and sort it. After that store the array back to hard disk array. I will provide the detail explanation in the later section. As to the Phase 1, I use new GPU merging path algorithm, and separate main memory array into array *A*, and array *B*. After we sort each sub-array, merge into array *C.* In Phase 2, it is nothing but compare each sub-array and find the minimal data for ascending sorting. However, before we discuss the new merging algorithm, I need to talk about the traditional algorithm of merging.

1.1 Serial Merging and Merge Path

Serial merging follows a well-known algorithm; it is necessary to present it due to a reduction that is presented further in this section. Given arrays A, B, C as defined earlier, a simplistic view of a decreasing-order merge is to start with the indices of the first elements of the input arrays, compare the elements, place the larger into the output array, increase the corresponding index to the next element in the array, and repeat this comparison and selection process until one of the two indices is at the end of its input array. Finally, copy the remaining elements from the other input array into the end of the output array. In part of research paper, the author suggests treating the sequential merge as though it is a path that moves from the top-left corner of an $|A| \times |B|$ grid to the bottom-right corner of the grid. The path can move only to the right and downwards. The reasoning behind this is that the two input arrays are sorted. This ensures that if $A[i] > B[j]$, then $A[i] > B[j']$ for all $j' > j$.

In this way, performing a merge can be thought of as the pro-cess of discovering this path through a series of comparisons. When $A[i] >= B[j]$, the path moves down by one position and we copy $A[i]$ into the appropriate place in *C*. Otherwise when $A[i] < B[j]$, the path moves right, and we copy $B[j]$ into the appropriate place in *C*. We can determine the path directly by doing comparisons. Similarly, if we determine a point on the path through a means other than doing all comparisons that lead to that point, we have determined something about the outcomes of those comparisons earlier in the path. From a

pseudo-code point of view, when the path moves to the right, it can be considered taking the branch of the condition when $A[i] \leq B[j]$, and when the path moves down, it can be thought of as not taking that branch. Consequently, given this path the or- der in which the merge is to be completed is totally known. Thus, all the elements can be placed in the output array in parallel based on the position of the corresponding segment in the path (where the position in the output array is the sum of row and column indices of a segment).

We can observe that sections of the path correspond to sections of elements from at least one or both of the input arrays and a section of elements in the output array. Here each section is distinct and contiguous. Additionally, the relative order of these workspaces in the input and output arrays can be determined by the relative order of their corresponding path sections within the overall path. In Figure below, $A[1] = 13$ is greater than $B[4] = 12$. Thus, it is greater than all $B[j]$ for $j \geq 4$. We mark these elements with a *'0'* in the matrix. This translates to marking all the elements in the first row and to the right of $B[4]$(and including) with a *'0'*. In general if $A[i] > B[j]$, then $A[i'] > B[j]$ for all $i' \leq i$. Figure below, $A[3] = 10$ is greater than $B[5] = 9$, as are $A[1]$ and $A[2]$. All elements to the left of $B[4]$ = are marked with a *'1'*. The same inequalities can be written for B with the minor difference that the *'0'* is replaced with *'1'*.

		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		16	15	14	12	9	8	7	5
A[1]	13	1	1	1	0	0	0	0	0
A[2]	11	1	1	1	1	0	0	0	0
A[3]	10	1	1	1	1	0	0	0	0
A[4]	6	1	1	1	1	1	1	1	0
A[5]	4	1	1	1	1	1	1	1	1
A[6]	3	1	1	1	1	1	1	1	1
A[7]	2	1	1	1	1	1	1	1	1
A[8]	1	1	1	1	1	1	1	1	1

Figure: Illustration of the Merge Path concept for a non-increasing merge. The first column (in blue) is the sorted array A and the first row (in red) is the sorted array B. The orange path (Merge Path) represents comparison decisions that are made to form the merged output array. The black cross diagonal intersects with the path at the midpoint of the path which corresponds to the median of the output array.

Observation 1: The path that follows along the boundary between the *'0's* and *'1's* is the same path mentioned above which represents the selections from the input arrays that form the merge as is depicted in Figure above with the orange, heavy stair-step line. It should be noted here that the matrix of *'0's* and *'1's* is simply a convenient visual representation and is not actually created as a part of the algorithm (i.e. the matrix is not maintained in memory and the comparisons to compute this matrix are not performed).

Observation 2: Paths can only move down and to the right from a point on one diagonal to a point on the next. Therefore, if the cross diagonals are equally spaced diagonals, the lengths of paths connecting each pair of cross diagonals are equal.

1.2 GPU Challenges

To achieve maximal speedup on the GPU platform, it is necessary to implement a platform-specific (and in some cases, card-specific) algorithm. These implementations are architecture-dependent and in many cases require a deep understanding of the memory system and the execution system. Ignoring the architecture limits the achievable performance. For example, a well-known performance hinderer is bad memory access (read/write) patterns to the global memory. Further GPU-based applications greatly benefit from implementation of algorithms that are cache aware. For good performance, all the Streaming Processors (SPs) on a single Streaming Multi-processors (SM) should read/write sequentially. If the data is not sequential (meaning that it strides across memory lines in the global DRAM), this could lead to multiple global memory requests which cause all SPs to wait for all memory requests to complete. One way to achieve efficient global memory use when non- sequential access is required is to do a sequential data read into the local shared memory incurring one memory re- quest to

the global memory and followed by "random" (non-sequential) memory accesses to the local shared memory.

An additional challenge is to find a way to divide the workload evenly among the SMs and further partition the workload evenly among the SPs. Improper load-balancing can result in only one of the SPs out of the eight or more doing useful work while others are idle due to bad global memory access patterns or divergent execution paths (if-statements) that are partially taken by the different SPs. For the cases mentioned, where the code is parallel, the actual execution is sequential. It is very difficult to find a merging algorithm that can achieve a high level of parallelism and maximize utilization on the GPU due to the multi-level parallelism requirements of the architecture. In a sense, parallelizing merging algorithms is even more difficult due to the small amount of work done per each element in the input and output. The algorithm that is presented in this paper uses the many cores of the GPU while reducing the number of requests to the global memory by using the local shared memory in an efficient manner.

2 GPU Merge Path

In this section we discuss a new algorithm for the merging of two sorted arrays into a single sorted array on a GPU. In the previous section we explained Merge Path and its key properties.

2.1 GPU Partitioning

Algorithm 1 Pseudo code for parallel Merge Path algorithm with an emphasis on the partitioning stage.

```
A_diag[threads] ⇐ A_length
B_diag[threads] ⇐ B_length
for each i in threads in parallel do
  index ⇐ i * (A_length + B_length) / threads
  a_top ⇐ index > A_length ? A_length : index
  b_top ⇐ index > A_length ? index − A_length : 0
  a_bottom ⇐ b_top
  // binary search for diagonal intersections
  while true do
    offset ⇐ (a_top − a_bottom)/2
    a_i ⇐ a_top − offset
    b_i ⇐ b_top + offset
    if A[a_i] > B[b_i − 1] then
      if A[a_i − 1] ≤ B[b_i] then
        A_diag[i] ⇐ a_i
        B_diag[i] ⇐ b_i
        break
      else
        a_top ⇐ a_i − 1
        b_top ⇐ b_i + 1
      end if
    else
      a_bottom ⇐ a_i + 1
    end if
  end while
end for
for each i in threads in parallel do
  merge(A, A_diag[i], B, B_diag[i], C, i * length/threads)
end for
```

W-Wide Binary Search: In this approach we fetch *w* consecutive elements from each of the arrays *A* and *B*. By using CUDA block of size *w*, each SP / thread is responsible for fetching a single element from each of the global arrays, which are in the global memory, into the local shared memory. This efficiently uses the memory system on the GPU as the addresses are consecutive, thus incurring a minimal number of global memory requests. As the intersection is a single point, only one SP finds the intersection and stores the point of intersection in global memory, which removes the need for synchronization. It is rather obvious that the work complexity of this search is greater than the one presented in Merge Path, which does a regular sequential search for each of the diagonals; however, doing w searches or doing 1 search takes the same amount of time in practice as the additional execution units would otherwise be idle if we were executing only a single search. In addition to this, the GPU architecture has a wide memory bus that can bring more than a single data

element per cycle making it cost-effective to use the fetched data.

In essence for each of the stages in the binary search, a total of w operations are completed. This approach reduces the number of searches required by a factor of *w*. The complexity of this approach for each diagonal is: $Time = O(log(n) - log(w))$ for each core and a total of $Work = O(w \cdot log(n) - log(w))$.

2.2 GPU Merge

The merge phase of the original Merge Path algorithm is not well-suited for the GPU as the merging stage is purely sequential for each core. Therefore, it is necessary to extend the algorithm to parallelize the merge stage in a way that still uses all the SPs on each SM once the partitioning stage is completed. For full utilization of the SMs in the system, the merge must be broken up into finer granularity to enable additional parallelism while still avoiding synchronization when possible. This paper research present the approach on dividing the work among the multiple SPs for a specific workload.

For the sake of simplicity, assume that the sizes of the partitions that are created by the partitioning stage are significantly greater than the warp size, *w*. Also, we denote the CUDA thread block size using the letter *Z* and assume that $Z \geq w$. For practical purposes $Z = 32$ or 64; however, anything that is presented in this subsection can also be used with larger *Z*. Take a window consisting of the *Z* largest elements of each of the partitions and place them in local shared memory (in a sequential manner for performance benefit). *Z* is smaller than the local shared memory, and therefore the data will fit. Using a *Z* thread block, it is possible to find the exact Merge Path of the *Z* elements using the cross diagonal binary search. Given the full path for the *Z* elements it is possible to know how to merge all the *Z* elements concurrently as each of the elements are written to a specific index. The complexity for finding the entire *Z*-length path requires $O(log(Z))$ time in general iterations. This is followed by placing the elements in their respective place in the output array. Upon completion of the *Z*-element merge, it is possible to move on to unmerged elements by starting a new merge window whose top-left corner starts at the bottom-right-most position of the merge path in the previous window.

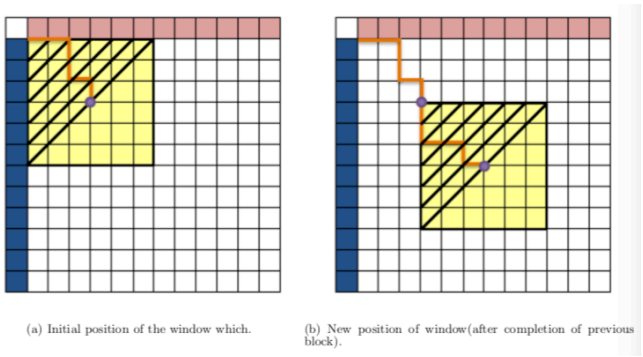


Figure: Diagonal searches for a single window of one SM. (a) The window is in its initial position. Each SP does a search for the path. (b) The window moves to the farthest position of the path and the new diagonals are searched.

This can be seen in Figure above where the window starts off at the initial point of merge for a given SM. All the threads do a diagonal search looking for the path. Moving the window is a simple operation as it requires only moving the pointers of the sub-arrays according to the *(x, y)* lengths of the path. This operation is repeated until the SM finishes merging the two sub-arrays that it was given. The only performance requirement of the algorithm is that the sub-arrays fit into the local shared memory of the SM. If the sub-arrays fit in the local shared memory, the SPs can perform random memory access without incurring significant performance penalty. To further offset the overhead of the path searching, we let each of the SPs merge several elements.

2.3 Complexity analysis of the GPU merge

Given *p* blocks of *Z* threads and *n* elements to merge where n is the total size of the output array, the size of the partition that each of the blocks of threads receives is *n/p*. Following the explanation in the previous sub-section on the movement of the sliding window, the window moves a total of *(n/p)/Z* times for that partition. For each window, each thread in the block performs a binary diagonal search that is dependent on the block size Z. When the search is complete, the threads copy their

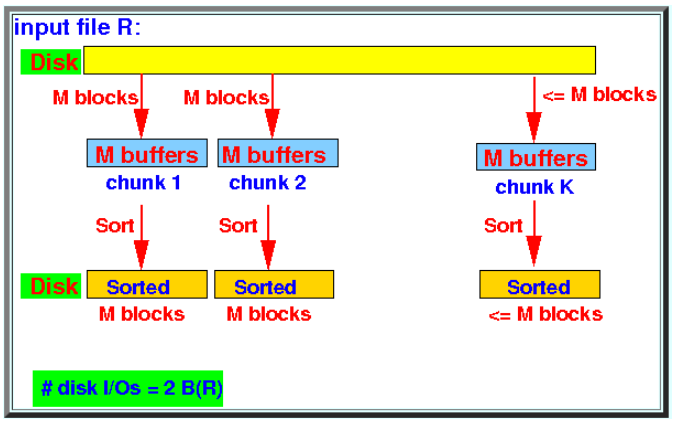
resulting elements into independent locations in the output array directly. Thus, the time complexity of merging a single window is O(log(Z)). The total amount of work that is completed for a single block is O(Z · log(Z)). The total time complexity for the merging done by a single thread block is O(n/(p · Z) · log(Z)) and the work complexity is *O(n/p·log(Z))*.

For the entire merge the time complexity stays the same, $O(n/(p \cdot Z) \cdot log(Z))$, as all the cores are expected to complete at the same time. The work complexity of the entire merge is $O(n \cdot log(Z))$. The complexity bound given for the GPU algorithm is different than the one given in previous section for the cache efficient Merge Path. The time complexity given by Odeh et el. is O(n/p + n/Zˆ · Zˆ), where Zˆ refers to the size of the shared memory and not the block size. It is worth noting that the GPU algorithm is also limited by the size of the shared memory that each of the SMs has, meaning that Z is bounded by the size of the shared memory. While the GPU algorithm has a higher complexity bound, I will show the code in the results section that the GPU algorithm offers significant speedups over the parallel multicore algorithm.

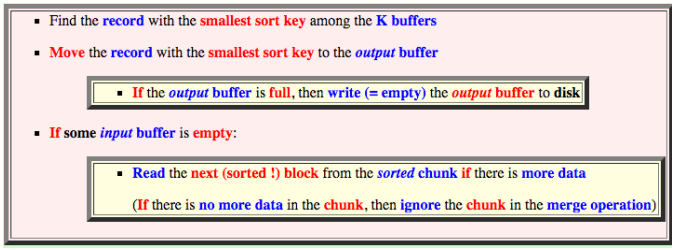
2.4 TPMMS Algorithm

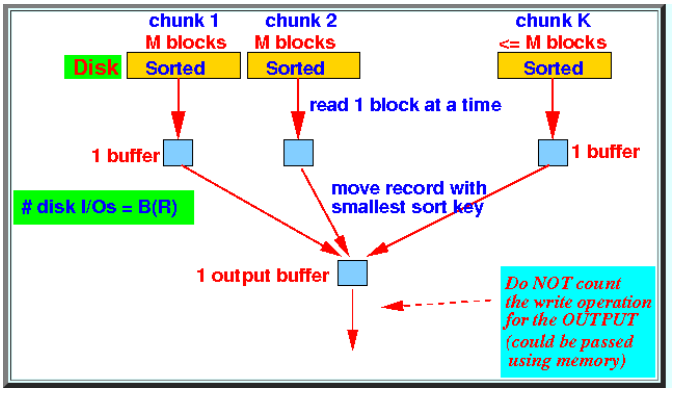
Suppose: There are M buffers available for storing the file data (1 buffer can hold 1 data block)

Phase 1: Divide the input file into chunks of M blocks each. Sort each chunk individually using the M buffers. Write the sorted chunks to disk.



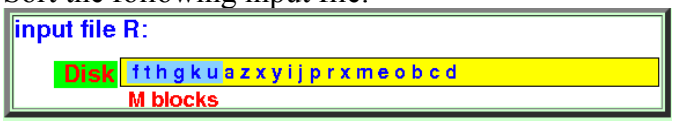
Phase 2: Divide the M buffers into *M-1* input buffers, and 1 output buffer. Use the *M-1* input buffers to read the *K* sorted chunks (1 block at a time). Use 1 output buffer to merge sort the *K* sorted chunks together into a sorted file as follow:



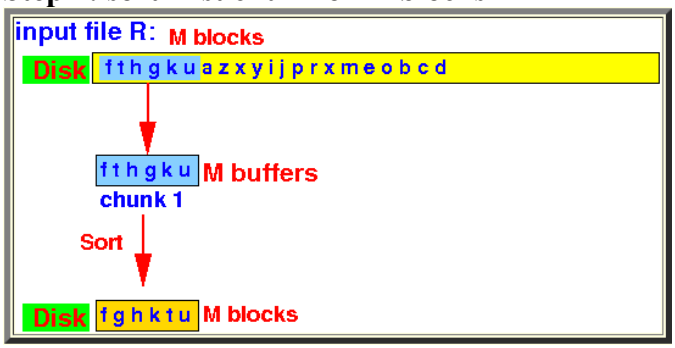


2.5 Example of TPMMS

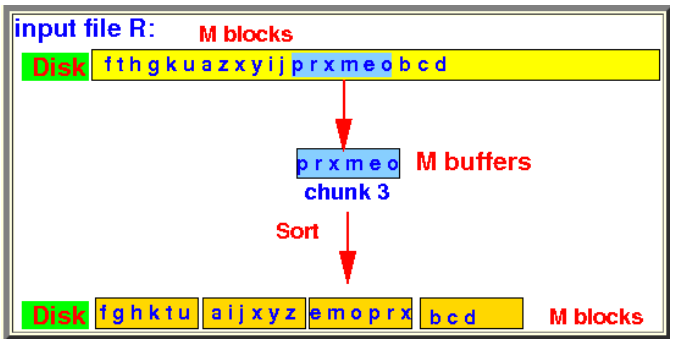
Sort the following input file:



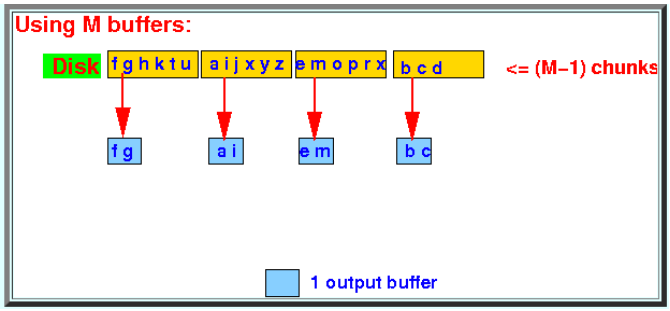
Phase 1:
Step 1: sort first chunk of M blocks



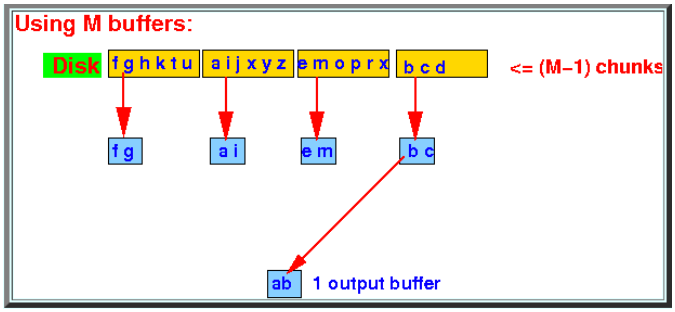
Step 2: sort second chunk of M blocks and so on



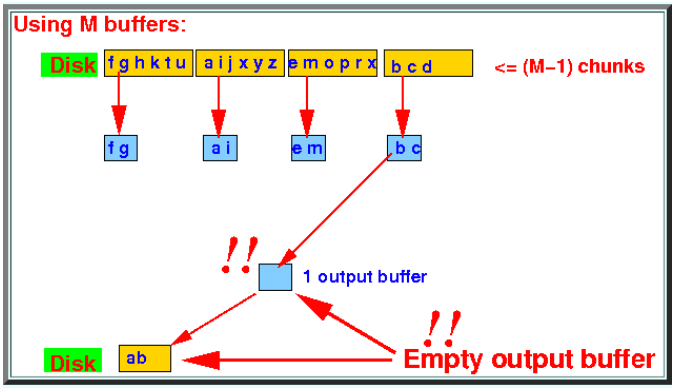
Phase 2:
Use 1 buffer to read each chunk and use 1 buffer for output, and read each chunk 1 block at a time



Move the smallest element to the output buffer and so on



When output buffer is full, empty it for re-use. Keep compare each block until all chunk is empty. Then output to the Disk.



3 Conclusion

The code I show later offers this as a starting point for comparison. However, what I discover is the performance is not very good, and this is due to the fact that two phase multiway merge sort and new merge algorithm fundamentally have a data-dependent access sequence, and so it is (more) difficult to arrange for coalesced access on the GPU. The authors of the paper propose to mitigate this problem by first reading the data (in a coalesced fashion) into shared memory, and then having the algorithm work on it out of shared memory, where the penalty for disorganized access is less.

I propose a different approach:
1. Arrange the sequential merge algorithm so that each element of A and B need only be read once
2. Arrange the storage of A, B, and C in column-major form as opposed to the more "natural" row-major storage that one might consider. This is effectively transposing the storage matrices for A, B, and C vectors. This allows for an improvement in coalesced access, as the GPU threads navigate their way through the merging operation on their individual A and B vectors. It's far from perfect, but the improvement is substantial.

Admittedly, despite I use a different approach to implement this project, it does not speed up as I expect. For this point, the reason may be not enough parallel executing, because of too much dependency.

HOST:

```
Total 240000 elements in HD array

CPU (get time of day) time: 37.461037
```

Column-Merge:

```
Total 24000 elements in HD array


GPU (get time of day)  time: 20.651079

GPU (cuda event) time: 20.651079
```

Row-Merge:

```
Total 24000 elements in HD array

GPU (get time of day)  time: 17.723797

GPU (cuda event) time: 17.723797
```

For further improvement, I tried to use shared memory, peer to peer, and UVA. Unfortunately, it does not compile correctly for some uncertain problems, I will also list the code in the later section. Finally, this algorithm, TPMMS, is not quite fit with the final project's goal, nonetheless, I already spent much time on picking up C programming language and also debugging the host part. Even I added the implement of new merging algorithm to optimize parallelism in this code, it is not take full advantage of GPU properties.

4 Code

Code separate 3 parts: HOST, Column-Merge, and Row-Merge. Each part of framework pretty similar except the GPU part. Therefore, I will list each distinct part as below.

Main framework (in host side, inside also include cuda part):

```
/*
**(a) Memory size for data: M records (Mem[M]) where M is a constant greater than or equal to 4
**(b) B records/blocks, where B is a constant greater than or equal to 2
**(c) A big sized hard disk (HD[100000])
**(d) HD[0] is save for an integer to indicate the number of records initially
**ALGORITH:
**1/ Use 2PMMS to sort the data in Mem and then move to/from HD
**2/ Use parallelism to do the pahse one which is sorting each sublist
**3/ Store those sublist into HD and doing phase two to finish merge part
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>
#include <sys/time.h>

#define NUM_SETS 4
#define DSIZE 2
typedef int mytype;
#define cmp(A,B) ((A)<(B))
#define nTPB 512
#define nBLK 128
#define MEMSIZE 8
#define RECORDS 24
#define FIXBLOCK 4
#define ARRSIZE 1000000
#define USECPSEC 1000000ULL

int main()
{
    // simulate HD array as hard disk and Mem array as Memory
    mytype HD[ARRSIZE], Mem[MEMSIZE];
    // pointer run in host
    mytype *ch_a, *ch_b, *ch_c;
    // pointer run in device
    mytype *d_a, *d_b, *d_c;

    // allocate variable size in memory
    ch_a = (mytype *)malloc((DSIZE*NUM_SETS/2)*sizeof(mytype));
    ch_b = (mytype *)malloc((DSIZE*NUM_SETS/2)*sizeof(mytype));
    ch_c = (mytype *)malloc(DSIZE*NUM_SETS*sizeof(mytype)*2);

    // cuda: allocate variable size in memory
    cudaMalloc(&d_a, ((DSIZE*NUM_SETS+1)/2)*sizeof(mytype));
    cudaMalloc(&d_b, ((DSIZE*NUM_SETS+1)/2)*sizeof(mytype));
    cudaMalloc(&d_c, DSIZE*NUM_SETS*sizeof(mytype)*2);

    // represent total elements
    HD[0] = RECORDS;
    printf("\nTotal %d elements in HD array\n", HD[0]);
```

```
	// random number upper bound = 10000
	int upper = 10000;
	// random number lower bound = 0
	int lower = 0;
	// generate random numbers and store them into HD
array
	for (int i = 1; i < RECORDS+1; i++)
	{
		HD[i] = (rand()% (upper-lower +1)) + lower;
	}

	// get current time
	cudaEvent_t start_time, stop_time;
	cudaEventCreate(&start_time);
	cudaEventCreate(&stop_time);
	unsigned long gtime = dtime_usec(0);
	cudaEventRecord(start_time, 0);
//declare a variable which represent how many sublist we
have
	int sublist = RECORDS / MEMSIZE;
	//a counter that tarck our current sublist
	int count = 0;
	//a pointer represent begining index from HD
	int start = 1;
	//a pointer represent store index location in HD
	int store = RECORDS +1;

	//read data from HD to Mem, and sort the data. After
that, store the data back to HD
	while (count < sublist)
	{
		//start from HD begining index to the Mem begining
index
		for (int i = start, j = 0; j < MEMSIZE; i++, j++)
		{
			Mem[j] = HD[i];
		}

		// separate those unsorted data into 2 part
		for (int i = 0; i < MEMSIZE/2; i++)
		{
			ch_a[i] = Mem[i];
		}
		for (int i = 0, j = MEMSIZE/2; j < MEMSIZE; i++, j++)
		{
			ch_b[i] = Mem[j];
		}

		//bubble sort
		for (int i = 0; i < MEMSIZE/2-1; i++)
		{
			for (int j = 0; j < MEMSIZE/2-i-1; j++)
			{
				if (ch_a[j] >= ch_a[j+1])
				{
					int temp = ch_a[j];
					ch_a[j] = ch_a[j+1];
					ch_a[j+1] = temp;
				}
			}
		}
		for (int i = 0; i < MEMSIZE/2-1; i++)
		{
			for (int j = 0; j < MEMSIZE/2-i-1; j++)
			{
				if (ch_b[j] >= ch_b[j+1])
				{
					int temp = ch_b[j];
					ch_b[j] = ch_b[j+1];
					ch_b[j+1] = temp;
				}
			}
		}

	  // pass those array into Device
		cudaMemcpy(d_a, ch_a,
((DSIZE*NUM_SETS)/2)*sizeof(mytype),
cudaMemcpyHostToDevice);
		cudaMemcpy(d_b, ch_b,
((DSIZE*NUM_SETS)/2)*sizeof(mytype),
cudaMemcpyHostToDevice);

		int arrsize = (DSIZE*NUM_SETS)/2;
		sort<<<1, 1>>>(d_a, d_b, arrsize);

		cudaDeviceSynchronize();

		cudaMemcpy(ch_a, d_a,
((DSIZE*NUM_SETS)/2)*sizeof(mytype),
cudaMemcpyDeviceToHost);
		cudaMemcpy(ch_b, d_b,
((DSIZE*NUM_SETS)/2)*sizeof(mytype),
cudaMemcpyDeviceToHost);

		cudaMemcpy(d_a, ch_a,
((DSIZE*NUM_SETS)/2)*sizeof(mytype),
cudaMemcpyHostToDevice);
		cudaMemcpy(d_b, ch_b,
((DSIZE*NUM_SETS)/2)*sizeof(mytype),
cudaMemcpyHostToDevice);

		// call kernel
		row_merge<<<nBLK, nTPB>>>(d_a, d_b, d_c,
NUM_SETS, DSIZE);

		// the kernel is guaranteed to finish (and the output
from the kernel will find a waiting standard output queue),
before the application is allowed to exit.
		cudaDeviceSynchronize();
```

```
            // pass the result back to Host
            cudaMemcpy(ch_c, d_c,
DSIZE*NUM_SETS*2*sizeof(mytype),
cudaMemcpyDeviceToHost);

            // restore those array value into memory
            int m_idx = 0, c_idx = 0, r_idx = 0;
            while (m_idx < 2*DSIZE*NUM_SETS)
            {
                  Mem[m_idx] = ch_c[c_idx*NUM_SETS+r_idx];
                  c_idx+=2;
                  if (c_idx == DSIZE*2)
                  {
                        c_idx = 0;
                        r_idx++;
                  }
                  m_idx++;
            }


            //after we sort the data, put it back to HD
            for (int i = 0, j = store; i < MEMSIZE; i++, j++)
            {
                  HD[j] = Mem[i];
            }

            // count represent how many times of iteration
            ++count;
            // next begining index in HD for unsorted data
            start = (MEMSIZE * count) +1;
            // next begining index in HD for sorted data
            store = RECORDS + (MEMSIZE * count) +1;
      }

 //========== phase two ===========//

      // how many blocks for each input and output memory
buffer
      int block = MEMSIZE / FIXBLOCK;
      // count the total IO for the terminal point
      int outputIO = RECORDS;

      if (outputIO % block != 0)
      {
            outputIO = RECORDS / block +1;
      }
      else
      {
            outputIO = RECORDS / block;
      }

      //tracking how many IO we have done
      int counter = 0;
      //store each pointer for HD
      int sublistarr[sublist*2];

      //even index represent the starting location of each
sublist
      for (int i = 0, j = 0; i < sublist*2; i+=2, j++)
      {
            sublistarr[i] = RECORDS + (MEMSIZE * j) +1;
      }
      //odd index represent the ending location of that sublist
      for (int i = 1; i < sublist*2; i+=2)
      {
            sublistarr[i] = sublistarr[i-1] + MEMSIZE;
      }

      //a pointer represent HD index when we finish phase
two
      int result = RECORDS + (MEMSIZE * sublist) +1;
      //represent memory index
      int memindex = 0;

      //store the first block from each sublist into memory
      for (int i = 0, y = 0; i < sublist; i++, y+=2)
      {
            for(int j = 0, k = sublistarr[y]; j < block; memindex++,
j++, k++)
            {
                  Mem[memindex] = HD[k];
            }
      }

      //each index's number represent the initial number of
the index
      int countarr[FIXBLOCK];

      for (int i = 0; i < FIXBLOCK; i++)
      {
            countarr[i] = block * i;
      }
  //loop for total times of output data from Mem to HD
      while (counter < outputIO)
      {
            //a loop for each sublist with comparison the min
number and store two min numbers to HD when violate the
condition
            while (countarr[FIXBLOCK-1] < MEMSIZE)
            {
                  //compare each index's number
                  if (Mem[countarr[0]] <= Mem[countarr[1]] &&
Mem[countarr[0]] <= Mem[countarr[2]])
                  {
                        Mem[countarr[FIXBLOCK-1]] =
Mem[countarr[0]];
                        countarr[0]++;
                        sublistarr[0]++;
                        countarr[FIXBLOCK-1]++;
```

```
                }
                else if (Mem[countarr[1]] <= Mem[countarr[0]]
&& Mem[countarr[1]] <= Mem[countarr[2]])
                {
                      Mem[countarr[FIXBLOCK-1]] =
Mem[countarr[1]];
                      countarr[1]++;
                      sublistarr[2]++;
                      countarr[FIXBLOCK-1]++;
                }
                else if (Mem[countarr[2]] <= Mem[countarr[0]]
&& Mem[countarr[2]] <= Mem[countarr[1]])
                {
                      Mem[countarr[FIXBLOCK-1]] =
Mem[countarr[2]];
                      countarr[2]++;
                      sublistarr[4]++;
                      countarr[FIXBLOCK-1]++;
                }
  //if the block is full and also the pointer of HD is less than
the ending location
                if (countarr[0] == block && sublistarr[0] <
sublistarr[1])
                {
                      for (int i = 0, j = sublistarr[0]; i < block; i++,
j++)
                      {
                            Mem[i] = HD[j];
                      }
                      countarr[0] = 0;
                }
                else if (countarr[0] == block && sublistarr[0] >=
sublistarr[1])
                {
                      for (int i = 0; i < block; i++)
                      {
                            Mem[i] = INT_MAX;
                      }
                }
                if (countarr[1] == (block*2) && sublistarr[2] <
sublistarr[3])
                {
                      for (int i = block, j = sublistarr[2]; i < block*2;
i++, j++)
                      {
                            Mem[i] = HD[j];
                      }
                      countarr[1] = block;
                }
                else if (countarr[1] == (block*2) && sublistarr[2]
>= sublistarr[3])
                {
                      for (int i = block; i < (block*2); i++)
                      {
                            Mem[i] = INT_MAX;
                      }
                }

                if (countarr[2] == (block*3) && sublistarr[4] <
sublistarr[5])
                {
                      for (int i = (block*2), j = sublistarr[4]; i <
(block*3); i++, j++)
                      {
                            Mem[i] = HD[j];
                      }
                      countarr[2] = block*2;
                }
                else if ( countarr[2] == (block*3) && sublistarr[4]
>= sublistarr[5])
                {
                      for (int i = block*2; i < block*3; i++)
                      {
                            Mem[i] = INT_MAX;
                      }
                }
          }

          for (int i = block*3, j = result; i < MEMSIZE; i++, j++)
          {
                HD[j] = Mem[i];
          }
          countarr[FIXBLOCK-1] = block*3;
          result += block;
          counter++;
     }
  gtime = dtime_usec(gtime);
     cudaEventRecord(stop_time, 0);
     cudaEventSynchronize(stop_time);
     float elapsedTime;
     cudaEventElapsedTime(&elapsedTime, start_time,
stop_time);

printf("\nGPU (get time of day)  time: %f\n",
gtime/(float)USECPSEC);
     printf("\nGPU (cuda event) time: %f\n", &elapsedTime);
     free(ch_a);
     free(ch_b);
     free(ch_c);
     cudaFree(d_a);
     cudaFree(d_b);
     cudaFree(d_c);

     return 0;
}
```

Merging Algorithm: Smerge

```
template <typename T>
```

```
__host__ __device__ void smerge(const T * __restrict__  a,
const T * __restrict__ b, T * __restrict__ c, const unsigned
length_a, const unsigned length_b, const unsigned stride_a =
1, const unsigned stride_b = 1, const unsigned stride_c = 1)
{
    unsigned length_c = length_a + length_b;
    unsigned nc = 0;
    unsigned na = 0;
    unsigned nb = 0;
    unsigned fa = (length_b == 0);
    unsigned fb = (length_a == 0);
    T nxta = a[0];
    T nxtb = b[0];

    while (nc < length_c)
    {
        if (fa)
        {
            c[stride_c*nc++] = nxta;
            na++;
            nxta = a[stride_a*na];
        }
        else if (fb)
        {
            c[stride_c*nc++] = nxtb;
            nb++;
            nxtb = b[stride_b*nb];
        }
        else if (cmp(nxta,nxtb))
        {
            c[stride_c*nc++] = nxta;
            na++;
            if (na == length_a)
            {
                fb++;
            }
            else
            {
                nxta = a[stride_a*na];
            }
        }
        else
        {
            c[stride_c*nc++] = nxtb;
            nb++;
            if (nb == length_b)
            {
                fa++;
            }
            else
            {
                nxtb = b[stride_b*nb];
            }
        }
    }
}
```

Phase 1: Sort

```
__global__ void sort(int *a, int *b, int array_size)
{
    int idx = threadIdx.x;

    for (int k = 0; k < array_size-1; k++)
    {
        for ( idx = 0; idx < array_size-k-1; idx++)
        {
            if (a[idx] >= a[idx+1])
            {
                int temp = a[idx];
                a[idx] = a[idx+1];
                a[idx+1] = temp;
            }
        }
    }
    for (int k = 0; k < array_size-1; k++)
    {
        for ( idx = 0; idx < array_size-k-1; idx++)
        {
            if (b[idx] >= b[idx+1])
            {
                int temp = b[idx];
                b[idx] = b[idx+1];
                b[idx+1] = temp;
            }
        }
    }
}
```

Row-merge:

```
template <typename T>
__global__ void row_merge(const T * __restrict__ a, const T
* __restrict__ b, T * __restrict__ c, int array_size, int length)
{
    int idx=threadIdx.x+blockDim.x*blockIdx.x;

    while (idx < array_size)
    {
        int sel = idx * length;
        smerge(a+sel, b+sel, c+(sel*2), length, length);
        idx += blockDim.x*gridDim.x;
    }
}
```

Column-Merge:

```
template <typename T>
__global__ void column_merge(const T * __restrict__ a,
const T * __restrict__ b, T * __restrict__ c, int array_size, int
length, int stride_a, int stride_b, int stride_c)
{
	int idx = blockDim.x * blockIdx.x + threadIdx.x;
	while (idx < array_size)
	{
		smerge(a+idx, b+idx, c+idx, length, length, stride_a,
stride_b, stride_c);
		idx += blockDim.x*gridDim.x;
	}
}

long long dtime_usec(unsigned long long start)
{
	timeval tv;
	gettimeofday(&tv,0);
	return((tv.tv_sec*USECPSEC) +tv.tv_usec) - start;
}
```

Get Time Of Day:

```
long long dtime_usec(unsigned long long start)
{
	timeval tv;
	gettimeofday(&tv,0);
	return((tv.tv_sec*USECPSEC) +tv.tv_usec) - start;
}
```

Shared memory for phase 1 sorting:

```
__global__ void sort(int *a, int *b, int array_size)
{
	int tx = threadIdx.x;
//	 __shared__
	int a_data[4],  int b_data[4];
	unsigned int i = blockDim.x * blockIdx.x + tx;
	a_data[tx] = a[i];
	b_data[tx] = b[i];

	for (int k = 0; k < array_size-1; k++)
	{
		for ( tx = 0; tx < array_size-k-1; tx++)
		{
			if (a_data[tx] >= a_data[tx+1])
			{
				int temp = a_data[tx];
				a_data[tx] = a_data[tx+1];
				a_data[tx+1] = temp;
			}
		}
	}
	for (int k = 0; k < array_size-1; k++)
	{
		for ( tx = 0; tx < array_size-k-1; tx++)
		{
			if (b_data[tx] >= b_data[tx+1])
			{
				int temp = b_data[tx];
				b_data[tx] = b_data[tx+1];
				b_data[tx+1] = temp;
			}
		}
	}
	__syncthreads();
}
```

5 References

[1] Mathcs.emory.edu—Cheung: Courses554 (http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/4-query-exec/2-pass=TPMMS.html)

[2] Oded Green, Robert McColl, David A. Bader: GPU Merge Path—A GPU Merging Algorithm