Homework 5 (Project 1)
Parallel Merge Sort using CUDA

Technical Summary

Based upon the lecture regarding merge sort on GPU, you are required to use the advanced Binary Search Based Intra-Tile sorting kernel (BSITR), which has been included into the startup package. After each tile sorted, you are required to implement a Hybrid Parallel Merge Sort (HPMS) that combines the Naïve Inter-Tile Merge Algorithm (NTMS) and the BSITR.

After the step 2 in GPU merge sort, each tile is sorted by BSITR, with a tile size ***Ts (e.g. = 256, or 512 or 1024),*** we have to merge all theses sorted tiles to produce the final entire sorted array. We discussed the Naïve Inter-Tile Merge Algorithm on GPU (NTMS). But NTMS is very inefficient because it is coarse-grain and number of threads used in merging decreases geometrically as the algorithm proceeds.

Idea of NTMS: (Note here, each tile initially has size of up to **Ts**. The last tile could be less than **Ts**)

|--------|--------|--------|--------|--------|--------|--------|--------|

tile0 tile1 tile2 tile3 tile4 tile5 tile6 tile7

thr0 thr1 thr2 thr3 → iteration 0

|----------------|----------------|----------------|----------------|

tile10 tile11 tile12 tile13

thr0 thr1 → iteration 1

In NTMS, thread 0 will merge tile0 and tile1, thread 1 will merge tile2 and tile3 in **first** iteration, so on so forth. In the **second** iteration, thread 0 will process up to **2 * Ts** pairs of elements when merging tile10 and tile11. Likewise, the same is performed for thread 1 when merging tile12 and tile13.

NTMS is very inefficient. Instead, we propose and use the following idea HPMS for the **third step** of GPU merge sort. **Idea of HPMS: (Note here: this is NOT the most optimal parallel merge sort algorithm either.)**

|--------|--------|--------|--------|--------|--------|--------|--------| (In the left diagram, entire input array contains 8 tiles, with each tile sorted already.)

tile0 tile1 tile2 tile3 tile4 tile5 tile6 tile7

blk0 blk1 blk2 blk3 → iteration 0

Enlarged tile0 and tile1 and thread block0: (a0 to a7 belong to tile0 and b0 to b7 is in tile1)

a0 a1 a2 a3 a4 a5 a6 a7 b0 b1 b2 b3 b4 b5 b6 b7

t0 t1 t2 t3 t4 t5 t6 t7

Here in this simple example, we assume each tile has up to 8 elements and each thread block has 8 threads. Thread

t0 computes rank(a0, C) and rank(b0, C), where C is the resultant array that contains the merged tile0 and tile1. Similarly, thread t1 computes rank(a1, C) and rank(b1, C), and thread t_i computes rank(a_i, C) and rank(b_i, C). The rank() operations can be performed by using binary search, as we learned in class. You can do binary search in **global memory** of the CUDA device, since all tiles are stored in the global memory. In the HPMS kernel, no shared memory is needed.

|---------------|---------------|---------------|---------------|

tile10 tile11 tile12 tile13

blk0 blk1 → iteration 1

In this iteration 1, we keep the **same thread block size** as with iteration 0, while tile10 has doubled the size of tile0 (or tile1) in the iteration 0 (tile10 here is a result of merging tile0 and tile1 in the iteration 0). The reason we do this is because we cannot continually increase the thread block size to accommodate the growing tile size. To address this problem, we make each thread in a block process **multiple adjacent elements** in the tiles to be merged, without increasing the thread block size across different iterations.

Enlarged tile10 and tile11 and thread block0 are shown below in the iteration 1. a0 to a15 belong to tile10 and b0 to b15 is in tile11.

a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 b10 b11 b12 b13 b14 b15

t0 t1 t2 t3 t4 t5 t6 t7

Here, the thread block size is still 8, same as in the previous iteration. However, the region to be merged is as twice larger than in the previous iteration, (Tile size in the iteration 1 is 16, but previously it was 8 in the iteration 0). To merge tile10 and tile11, each thread has to do **twice much work** as it did in the previous iteration. That is, Thread t0 computes rank(a0, D) and rank(b0, D), and rank(a1, D) and rank(b1, D), where D is the resultant array that contains the merged tile10 and tile11. Likewise, thread t1 computes rank(a2, D) and rank(b2, D), and rank(a3, D) and rank(b3, D) and thread t_i computes rank(a_{2*i}, D) and rank(b_{2*i}, D), and rank(a_{2*i+1}, D) and rank(b_{2*i+1}, D). Each thread performs four rank() operations in this iteration 1.

In the **next** iteration 2, the thread block size is the same, 8, but each thread will perform eight rank() operations after the region to be merged is doubled after iteration 1. By following the same pattern, in the last iteration of the HPMS, there will be one full active thread block to merge two really big tiles, generating the entire sorted array.

Requirements:

1. Following and implement the ideas presented above about HPMS (required to implement by yourself) and BSITR (provided).
2. Randomly populate your input key array with integer number between 1 to 10000. In addition to a parameter **int *d_srcKey**, your kernel is required to take **int *d_srcVal** as another parameter. I.e. input and output are key array and value array, as we learned in the class. You are required to output the sorted keys and sorted values that are associated with each key. You can initialize each element in the input **value** array with the index of the key. In this way, you will produce a sorted index array.
3. The size of input array **N** will NOT be limited by the maximal number of thread blocks in the grid. You have to use a 2D grid configuration to create more than 65535 thread blocks, otherwise the number of thread blocks is limited to 65535 in a 1D grid. But if you use 2D grid, the maximal number of blocks in the grid would be up to 65535 by 65535 blocks. (The maximal number of blocks **per dimension** is 65535.)

However, you have to be very careful when mapping a 2D array of thread blocks to 1D input data.

4. The size of the input array **N** has to be a command-line argument. The thread block size is another command-line argument.
5. Compare the parallel implementation with a CPU sequential merge sort program, and fill out the following chart.
6. Add a simple Makefile to compile your code into target hw5.
7. Pleas try a small input array first in order to verify the correctness of your parallel code.

If there is anything special you'd like to point out about your implementation, include a brief paragraph or bullet list of features below your chart. Note here, in the second step of GPU merge sort, when sorting **individual** tile in parallel (with BSITR), tile size of dataset is twice as the size of a thread block, as shown in the provided kernel. However, when we merge the sorted tiles in parallel with HPMS (the third step of GPU merge sort), size of a tile is doubled after each iteration, but the thread block size is kept the same.

N, the size of input 1D array	2000000	4000000	8000000	8000000	8000000	16000000	16000000	1600000
threadPerBlock	512	512	256	512	1024	256	512	1024
Time cost of Serial CPU (ms) T1								
Time cost of CUDA (ms) T2								
Speedup of Parallel code compared with Serial code (T1 / T2)								

Submission

Wrap up all your source code and your result report into a single zip file, submit the single zip file through EWU canvas.