Homework 4 (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 algorithm (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 Step 2, each tile is sorted by BSITR, with a tile size ***Ts (e.g. = 256, 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.

Idea of HPMS: (Note here: this is NOT an optimal parallel merge sort algorithm)

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

tile0 tile1 tile2 tile3 tile4 tile5 tile6 tile7

blk0 blk1 blk2 blk3 → iteration 0

Enlarged tile0 and tile1 and 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 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 4 threads. Thread t0 computes rank(a0, C) and rank(b0, C), where C is the resultant array that contains the merged tile0 and tile1.

The rank() operations can be done by using binary search, as we learned in class. You can do binary search in global memory of the CUDA device.

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

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 is almost as twice as tile0 or tile1 in iteration 0. The reason we do this because we cannot continually increase the thread block size to accommodate to the growing tiles. 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.

Requirements:

1. Following the ideas presented above about HPMS and BSITR.
2. Randomly populate your input key array with integer number between 1 to 10000. Your kernel is required to take int *d_srcVal as another parameter. I.e. inputs and output are key array and value array. You are required to output the sorted keys and sorted values associated with each key. You can initialize each element in the input **value** array to the index of the key with which a value associates. In this way, you can produce a sorted index array.
3. The number of elements in the input array **N** CANNOT be limited by the maximal number of blocks in the grid. You have to use a 2D grid configuration, because if you use 2D grid, the maximal number of blocks in the grid would be up to 65535 * 65535. (The maximal number of blocks per dimension is 65535.) However, you have to be very careful when mapping a 2D array of 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 algorithm with CPU sequential merge sort program, and fill out the following chart.
6. Add a simple Makefile to compile your code into target hw4.
7. 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 step 2 when sorting **individual** tile in parallel, tile size of dataset is twice as the size of thread block, shown in the provided kernel. However, when we merge sorted tiles in parallel, size of tile is doubled after each iteration.

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.