Report of Assignment 4

Hanzhang Song studentID:hs50

December 12, 2018

1 Algorithm

1.1 Divide Array into Subarrays

When the vector of numbers is larger than the memory of GPU. Let N denote the size of the vector, Mem_D denote the memory size of GPU. I will first divided the large vector into N/Mem_D parts. Because the size of vector and the size of GPU's memory are both powers of 2, now we have 2^p parts: [part_1, part_2, ... , $\text{part}_{N/\text{Mem}_D}$]. Each part's size can fit in the GPU memory, I use bitonic sorting algorithm to sort them seperately and let the even parts and odd parts have different sort order (ascending or descending). Then we will have an array like this:

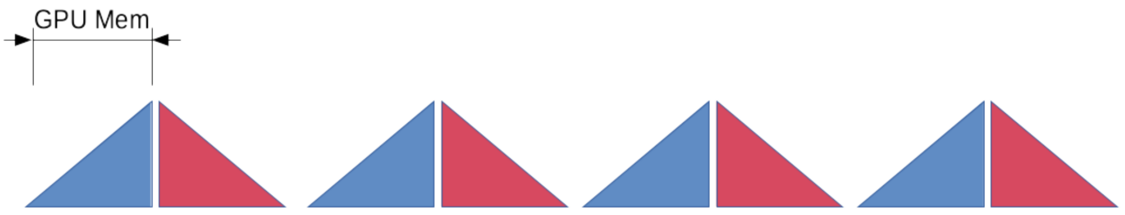

Figure 1: Sort even parts and odd parts with different directions.

1.2 Merge Large Subarrays

Next step is to use bitonic sorting algorithm to merge these large subarrays. Again, because the size of two subarrays is larger than GPU's memory, we should do the merging stage by stage. If the size of each subarrage is 2^k times of the size of GPU's memory, then we will have 2^{k+1} stages to merge them. In every stage, we process $1/2^{k+1}$ portion of the first subarray and the second subarray.

Figure2 can illustrate it more vividly. The size of subarray is 2 times of the GPU's memory and we will have 4 stages. Parts with same number are processed in the same stage. For every stage, we use the first half of GPU's memory to store a part of the first subarray, and use the second half the GPU's memory to store a part of the second subarray. Then let GPU do the exchange job according to bitonic sorting algorithm.

Now we have 2 bitonic sequences, we want to merge them too. Again they are still too large to fit in GPU's memory. So we need do the stage merge again until the subarray's size is small enough to fit in GPU's memory. This is a recursive call. And the code is like:

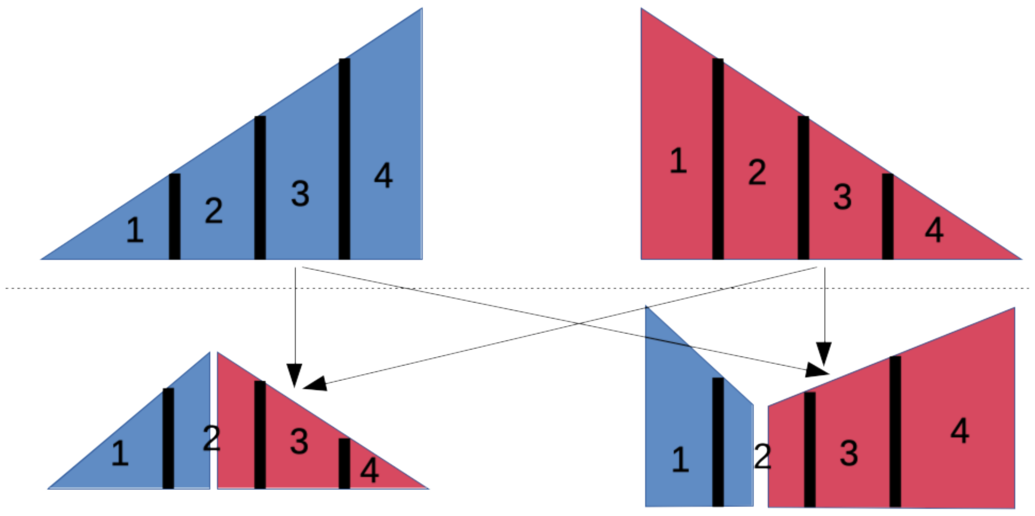

Figure 2: Stages of merge.

```
stage_merge(uint* h_array, uint array_length, uint* d_array, uint dir)
{
  if (array_length is small enough) {
    copy memory host to device
    CallGPUtoMergeSmallSubarrays<<<blocknum, threadnum>>>(...);
    copy memory device to host
  }
  else {
    for (stage_idx in [0,1,2,..., array_length/GPU_Mem]) {
      copy part of the first subarray to the first half memory of GPU
      copy part of the second subarray to the second half memory of GPU
      CallGPUtoExchangeElements<<<blocknum, threadnum>>>(...);
      copy memory from GPU back to the first subarray
      copy memory from GPU back to the first subarray
    }
    stage_merge(first_two_pair, array_length/2, d_array, dir);
    stage_merge(last_two_pair, array_length/2, d_array, dir);
  }
}
```

2 Details of Device Functions

The functions' name and what they do is listed as below:

- __global__ void bitonicSortShared1(...): Input is an unsorted array which can fit in GPU's memory. Every block is responsible for a part of the array. Blocks with even blockId and bolcks with oadd blockId will sort the subarray in different directions.
- __global__ void bitonicMergeGlobal(...): Input is two bitonic sequence, every thread is re-

sponsible for a pair of elements which respectively come from the two bitonic sequence. The index of the elements of the pair is according to the stride parameters.

- __global__ void bitonicMergeShared(...): Input is two bitnoic sequence whose size is small enough to fit in block's share memory. Do bitonic merge on these two sequence and output the sorted array.

I will dig into the three functions above to introduce how they work.

2.1 __global__ void bitonicSortShared1(...)

2.1.1 How does it work

All the blocks do exactly the same job except that blocks with even blockId and blocks with odd blockId will give out sorted array with different directions. A block first allocates a shared block array whose size is SHARED_SIZE_LIMIT, then the threads in this block will copy a subarray from the global array into this block shared array. Then run the bitonic sorting algorithm on the block shared array. Because this array is located on the block the threads do not need to visit global memory when run the bitonic sorting algorithm. The direction of the last merge is according to blockId. Then copy the block shared array to the global array.

2.1.2 Global memory access times and thread number

At the beginning, every block will visit the global memory once to read the subarrays and at the end every block will visit the global memory once again to write back the sorted subarrays. Because the bandwidth may be not enough for all blocks to read or write at the same time, some blocks may wait until other blocks finish their reading or writing.

In this assignment, I limit the block shared memory can only store 1024*sizeof(uint) elements. So compared with the size of shared memory we have plenty threads to used but we won't use them all because every thread is responsible for comparating two elements so the number of threads in a block is limited by the size of block shared memory.

2.2 __global__ void bitonicMergeGlobal(...)

2.2.1 How does it work

This function do not use block shared memory. Every thread reads a pair of elements from the global memory compares and exchanges them then writes back to the global memory. The index of the pair is (blockId*blocksize+threadId, blockId*blocksize+threadId+stride). The stride must be larger than SHARED_SIZE_LIMIT.

2.2.2 Global memory access times

Because every thread will access the globle memory once to read and once to write, so every block will access the global memory 2*threadnum times. So the bandwidth will be very busy, many threads will wait to read or write.

2.3 __global__ void bitonicMergeShared(...)

bitonicMergeShared and bitonicSortShared1 are very similar. They both allocate a block shared array and read from the global array and then run a bitonic sorting algorithm on the shared array.

The difference is that the input of bitonicMergeShared is two bitonic sequence. So we do not need to merge from size of 2, so bitonicMergeShared does not need the out loop in bionicSortShared1.

3 Timing

The average time for sorting an array of 2^{26} elements is 17339.1878906 ms.