COMP534: Project 4: Bitonic sort using a GPU with CUDA

Dejan Grubisic

May 6, 2020

Abstract

This project implements the Bitonic sort, discovered by Ken Batcher [1], on the GPU using Nvidia's CUDA model. This algorithm is based on sorting network techniques that enable high utilization of parallelism in the GPU. In this project, we used Nvidia Tesla V100-SXM2-32GB graphic card and run the sorting experiments on the arrays with size from 1 to 2^{26}. Based on the results from experiments we will discuss execution time and throughput metrics.

1. **The problem statement and a short motivation for why solving this problem is useful.**

   Bitonic sort represents one of the fastest sorting networks with time complexity $O(log^2(n))$ and $O(n \cdot log^2(n))$ comparisons for the arrays of size power of two. This algorithm is, therefore, faster than comparison-based sorting algorithms. On the other hand, it is not cost-optimal compare to comparison-based algorithms as the number of comparisons is $log(n)$ times bigger, but it is cost-optimal compare to serial implementation of the bitonic sort.

   The idea behind bitonic sort is to sort a bitonic sequence, which consists of exactly one increasing and decreasing sequence in a circular array. If we can construct such a mechanism, we can sort an array, by applying this principle recursively up to single elements. There are several known implementation of such networks. The network we used for this project is shown in Figure 1.

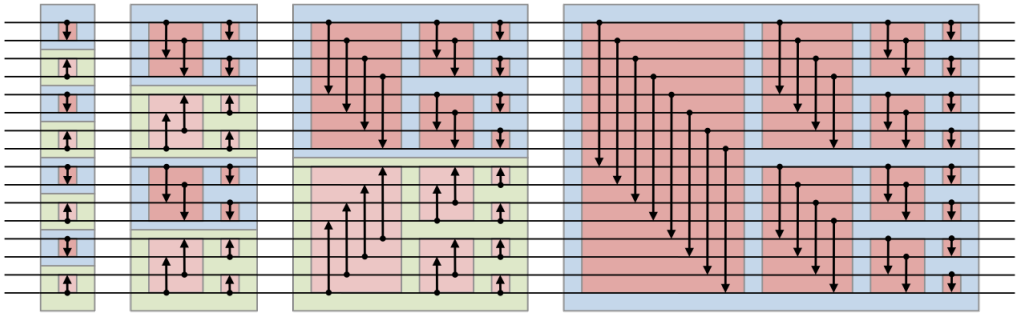

Figure 1: Bitonic sort network

Bitonic sorting network consists of an alternating sequence of sorters that take one bitonic sequence on their input and produce and produce the sorted array on their output. Two sorted arrays in opposite direction create a new bitonic sequence with two times bigger length, which is the input for the next sorter. Following this principle, we will create a sorted sequence for the whole array. On the upper figure, we can see described the pattern in which decreasing sorters are colored in blue while increasing sorters are colored in green. From this figure, it is also easy to see that there is $log(N)$ comparison stages and in each stage, there is $log(N)$ comparators which result in $O(log^2(n))$ time complexity.

2. **The details of your approach and an explanation of how/why this approach solves your problem**

The bitonic sort is implemented in the Nvidia Cuda programming model for the Tesla V100-SXM2-32GB graphics card, by using maximum one array of 4kB of shared memory(1000 values) and one array of 4MB GPU memory(1M values). Depending on the number of the array we are sorting, the execution is performed completely in shared memory for the arrays if the length is less than 1000, in one batch on the GPU if the length is less than 1M and in multiple batches, if the array is bigger than that. These three cases will be discussed separately.

Arrays that fits inside shared memory At the beginning of the program, we initialize the array with random numbers and allocate the same amount of memory for the GPU. After that, we send random array from host to device and initialize allocated memory using the function *cudaMemcpy*. As allocated array fits inside shared memory we can sort the array in a single stage. For that, we call function *bitonicSortShared* with thread blocks of size batchSize and half of arrayLength threads in each block.

```
bitonicSortShared<<<batchSize, arrayLength/2>>>(d_array, arrayLength, dir);
```

This function tightly resembles the pattern in Figure 1 in which each thread compares two values in each comparator colored in red. On the beginning each thread copy two values inside the shared memory, which is associated with the thread block so that all threads from the block could read and write in it. The pattern implementation is given in the listing below.

```
for (uint size = 2; size <= arrayLength; size <<= 1){
        uint ddd = dir ^ ((threadIdx.x & (size / 2)) != 0);

        for (uint stride = size / 2; stride > 0; stride >>= 1){
            cg::sync(cta);
            uint pos = 2 * threadIdx.x - (threadIdx.x & (stride - 1));
            Comparator(
                s_key[pos +      0],
                s_key[pos + stride],
                ddd
            );
        }
    }
```

Outer for-loop determines the size of the comparator at each stage, while the inner loop determines the distance between comparing values inside a comparator. The outer loop also determines the direction of each comparison inside a block in variable *ddd*. Variable *pos* determines the the starting value for each comparator based on *thread_id*. As one thread shouldn't proceed further until all threads in the block finish their comparison there is a synchronization after each step. When position and stride are calculated, Comparator just compares the appropriate values in direction *ddd*. On the end, each thread copies back two values into GPU memory that will be forwarded to the CPU.

Arrays that fits inside GPU memory If the array can not fit inside the shared memory we are performing multiple kernel stages inside GPU. Now, the first step is to sort every consecutive two blocks of the array of size equal to shared memory in opposite directions (Figure 2). In the next step, we perform the bitonic merge operation on the consecutive blocks. By doing so we create a bitonic sorter twice bigger and we put larger elements from two subarrays in the upper block while the lower elements go to lower block in the GPU memory. By sorting these two arrays in descending order we get the sorted sequence of length 2 x shared_memory. As each of subarrays can fit in

shared_memory, we are doing sorting there. The same procedure can be applied to the next pair of subarrays, with sorting the values in ascending order. After we finish this we are having a bitonic sequence of size 4 x shared_memory. Following this pattern, we are going to sort the whole array.

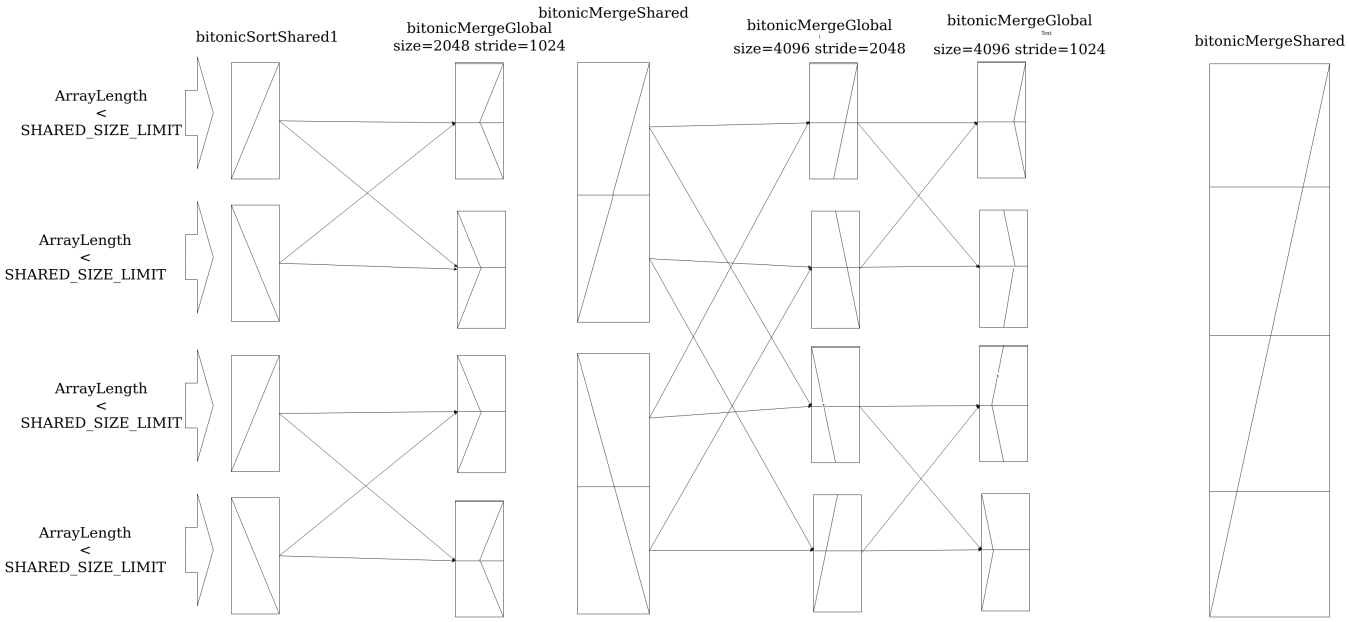

Figure 2: Conceptual sorting network for array lenght equal to 4 x shared memory

Arrays that exceeds GPU memory If we cannot store the whole array in GPU memory, we have to sort and merge piece by piece that fits in memory. In the principle, this is very similar to the approach in Figure 2, but this time instead of shared memory size we have GPU memory size. So, the first step is to divide the array into pieces of GPU memory size and to sort them alternately into ascending and descending order.

The merge step differs from the in-GPU approach, as we cannot merge two subarrays larger than GPU memory size all at once. Instead, we need to split them into pieces of sizes of size $\frac{1}{2}$ GPU memory we can store the piece of both subarrays and merge it on the GPU. Once we send two subarrays to GPU, we perform a bitonic merge by comparing the first and second half. After this operation, the bigger/lower elements will reside in the upper or lower part, depending on the merge direction. In the last step, we just copy these parts back to original arrays on their starting position.

To validate the results we just perform the comparisons in the CPU between each consecutive value in the array. If all values with a higher index are bigger or smaller than the value with a lower index, we consider the array sorted.

3. **A description of the experiments you conducted. Give the quantitative and qualitative analysis of your results.**

   Experiments had been conducted for the arrays of length from 1 to 2^{26} on the Nvidia Tesla V100-SXM2-32GB GPU. The execution time(seconds) and throughput(million elements/second) are shown in Figure 3. From these figures, we can see that execution time increases sharply for an array length bigger than 2^{20}. This is exactly how big is our GPU memory. Once the array exceeds this dimension we need to perform the bitonic sort in several batches. The main performance bottleneck lays is the transfer of memory between CPU and GPU. With doubling the array size, we need to perform also about twice more memory transfers that make the major impact on the performance.

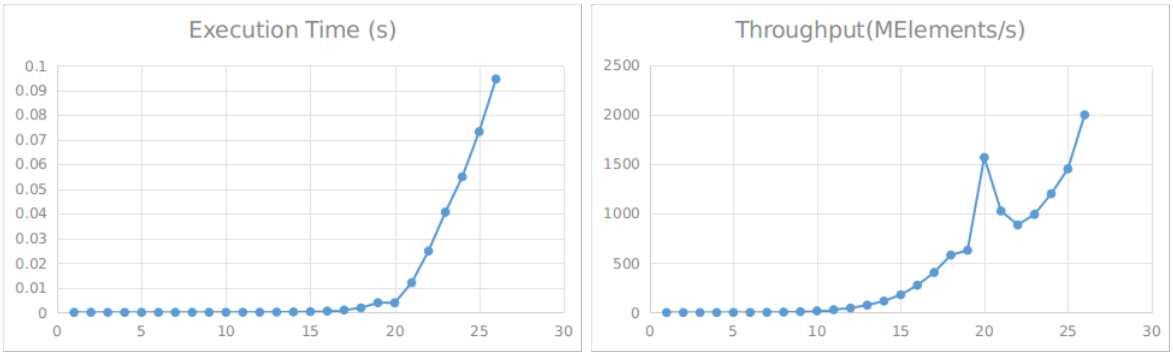

Figure 3: Execution time and Throughput for different array size $\log_2$ scale

   The throughput is calculated by dividing the number of elements with execution time. From Figure 3 we can see that throughput has a pick for array length 2^{20} and that it grows after that. Until we reach the pick, we just need to perform one memory transfer forth and back and to perform the sorting on the GPU. As GPU computation is fast $O(log^2(n))$, array size growth is is bigger than execution time. When array size exceeds the GPU memory size we need to perform additional memory transfers that increase execution time, which decreases throughput. On the other hand with increasing array length increases the amount of computation on GPU that becomes dominant compare to memory transfer, which increases throughput. This behavior confirms once again that time complexity less than $O(N)$, as throughput has an increasing tendency.

4. **Conclusion** This project implements and analyses the CUDA implementation of the bitonic sort. In this report, we showed the idea behind the bitonic sort, it's implementation on the GPU for various array sizes and we showed discussed obtained results.

   Bitonic sort represents one of the fastest sorting algorithms with time complexity of $O(log^2 N)$, if we can perform $\frac{N}{2}$ comparison at the time. With a high number of available threads, GPU programming enables us to highly utilize given parallelism.

   From shown results, we can conclude that GPU executes extremely efficiently when we can fit all array into GPU memory. Once, array size exceeds the bitonic sort must be performed in several batches that move memory back and forth between CPU and GPU which increases execution time. Even with this, the execution time increases linearly with doubling the array size. The throughput, therefore, has also an increasing tendency with a pick at the point when array fills GPU memory, as we have efficient GPU computation and only one memory transfer forth and back.

References

[1] https://en.wikipedia.org/wiki/Bitonic_sorter