GPU Computing Course 2020-2021

Comparison between a sequential bitonic sort and the parallel GPU-Warpsort

Stace Briggith Bravo[1] and Michele Leva[1]

[1] Universitá degli Studi di Milano
Dipartimento di Informatica "Giovanni Degli Antoni"
Via Celoria 18 - 20133 Milano (MI)
stacebriggith.bravo@studenti.unimi.it, michele.leva1@studenti.unimi.it

January 14, 2022

Abstract

In this paper, we implemented the GPU - Warpsort created by *et al.* Ye et al. (2010) and tested its effectiveness compared to the sequential bitonic sort Grossi (2021).

The algorithm mainly consists of a bitonic sort followed by a merge sort and achieves high performance by efficiently mapping the sorting tasks to GPU architectures. Firstly, we take advantage of the synchronous execution of threads in a warp to eliminate the barriers in bitonic sorting network. We also provide sufficient homogeneous parallel operations for all the threads within a warp to avoid branch divergence. Furthermore, we implement the merge sort efficiently by assigning each warp independent pairs of sequences to be merged and by exploiting totally coalesced global memory accesses to eliminate the bandwidth bottleneck.

Assuming a starting array of **N** values with no doubles, where **N** is a power of 2, our results suggest that our GPU-based sort can be even 100% faster than the sequential counterpart with **N** sufficiently high.

Keywords Sorting Algorithm, Many-Core, GPU, CUDA, Bitonic Network, Merge Sort

1 Introduction

In this paper we compare the efficacy of the sequential bitonic sort to the GPU warpsort created by Ye et al. (2010).

The algorithm we implemented is mainly composed of two parts: Firstly, we divide the input sequence into equal sized subsequences and use a bitonic network to sort each of them. After that, a merge sort follows to merge all small subsequences into the final result. In order to map the sorting tasks to the GPU architecture efficiently, our algorithm takes advantage of the synchronous execution of threads in a warp, and organizes the bitonic sort and merge sort using one warp as a unit, hence the name GPU-Warpsort.

The algorithm is implemented on the online platform Google Collab with a Tesla K80 GPU. Results from our tests indicate more than 100% higher performance than a sequential bitonic sort.

In the following sections the implementation of the algorithm will be explained in more detail, followed by tests and related results. Finally, we will present a discussion on the methodology used and the conclusions drawn from the work.

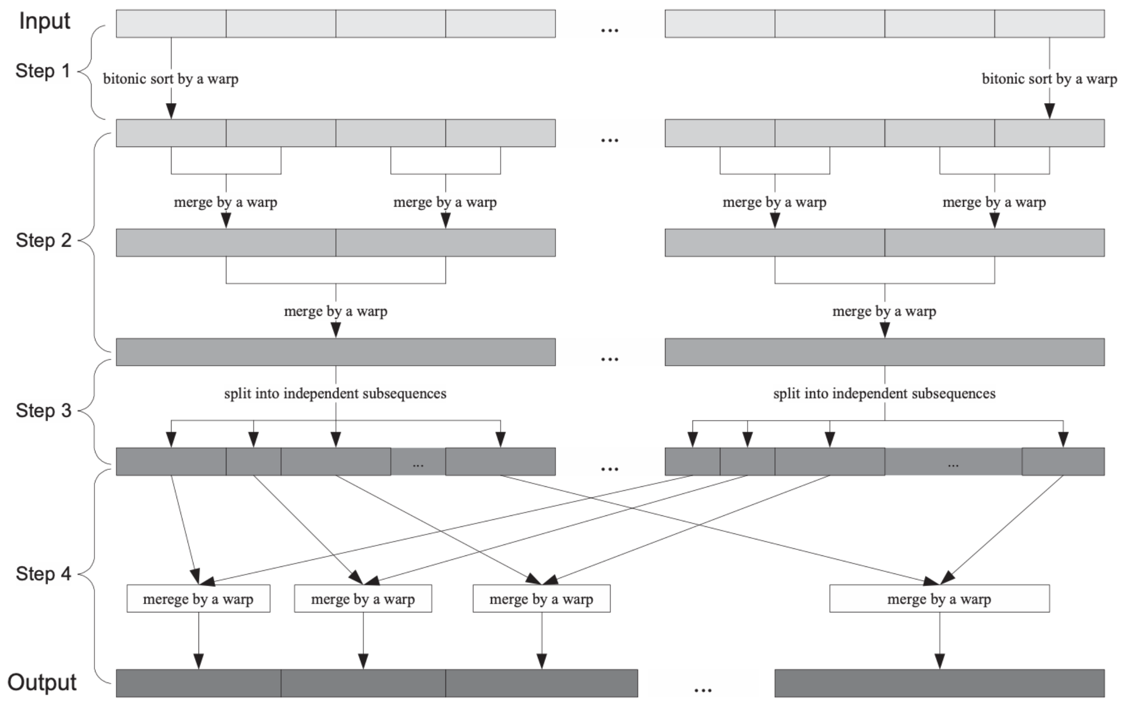

Figure 1: Overview of the sorting algorithm. It mainly consists of 4 steps as explained in the Implementation section

2 Implementation

Figure 1 gives an overview of the sorting algorithm we have implemented following the original paper. It mainly consists of 4 steps as follows:

1. Divide the input sequence into equal-sized subsequences. Each subsequence will be sorted by an independent warp using the bitonic network.
2. Merge all the subsequences produced in step 1 until the parallelism is insufficient.
3. Split the large subsequences produced in step 2 into small ones that can be merged independently.
4. Merge the small subsequences produced in step 3 by different warps or blocks independently.

In the following section a general implementation of the steps is explained, a more explicit explanation can be found in the original paper.

2.1 Barrier-Free Bitonic Sort

The first step of the algorithm divides the input array in small subsequences and each one is ordered with a bitonic network. Each stage of the bitonic sort reorders pairs of values in an ascending or descending order depending on the stage reached, and a global synchronization is needed between each stage. The algorithm uses a warp-based hardware synchronization instead of a block-based one removing the need of explicit barriers.

Each warp is then responsible for a subsequence, and each thread is able to reorder 2 pairs, the first in ascending order and the second in descending order. To avoid idle or divergent threads is necessary that all 32 threads of a warp work simultaneously, each executing 2 compare-and-swap on 4 different values of the subsequence. Each sequence in total then is composed of 128 values. As a consequence the dimension of the input array must be a multiple of 128.

2.2 Bitonic-Based Merge Sort

At the end of the previous step we have the input sequences divided in subsequences of 128 values in ascending order.

In this step each warp merges a pair of subsequences, and to achieve this uses a buffer in shared memory of dimension $\mathbf{T}$. $\mathbf{T}/2$ elements from the the two subsequences are loaded on the buffer composing a bitonic sequence. The last stage of a bitonic sort can then be used to sort the buffer in ascending order. Finally, the first $\mathbf{T}/2$ of the buffer are sent to the output and new $\mathbf{T}/2$ values are loaded on the buffer from one of the subsequences. This operation is repeated until all values from the two subsequences have been correctly ordered.

To order the **T** elements of the buffer, an algorithm similar to the one used in the previous step is used, but each thread must compare-and-swap only one pair in ascending order, so each of the 32 threads of a warp is responsible of 2 values. As a result, to avoid thread idling, the ideal dimension **T** of the buffer is 64.

To load the descending half of the buffer (that composes the bitonic sequence) from the global memory it would seem easier for each thread to fetch each value in the same (reverse) order, but this method causes uncoalesced memory access. So each thread access the global memory in ascending order and then reverses the loading on the buffer, resulting in coalesced access to the global memory and an elimination of bandwidth bottleneck.

The merge of pairs of two ordered subsequences is replicated until parallelism is insufficient, with each step doubling the size of the ordered subsequences. The parallelism will be recovered in the next steps.

2.3 Split into Small Tiles

In merge sort, the number of pairs to be merged will decrease geometrically. To avoid losing parallelism when there are no pairs of sequences to be merged, we divide the large subsequences produced in step 2 into smaller tiles that can be merged independently. We assume that we have **L** sequences of size n and each sequence will be divided into **S** subsequences. The paper suggests that these parameters should be larger than the number of SMs, in our case we have 16 SMs.

To obtain the splitters we call the method getSplitters() that has the funcion of sampling the input sequence randomly to get k-th sample elements. Then we sort the sample sequence and construct the splitters by selecting every k-th element of the sorted sample sequence.

In order to construct an even number of splitters, we sample the original input sequence randomly to get s*k sample elements. Then we sort the sample sequence and construct the splitters by selecting every k-th element of the sorted sample sequence. Choosing a larger k consumes more time, but produces better splitters. We obtain as output a linear and sorted array which contains the splitters.

Once we obtained the splitters we use them to save the subsequences' indexes from the input array for each row into a matrix. We use a block per row and a thread for each subsequence of the row. Each thread is tasked to find the starting index of the assigned subsequence.

2.4 Final Merge Sort

At the end of previous step we obtain **S** independent subsequences for each of the **L** rows. First, we control if the segments are smaller than 128, if they are we patch the start and end of each subsequence by inserting some special keys in order to avoid uncoalesced global memory access. Then we order each column executing the step 2.

As all the values of the subsequences of a column are in the same interval of numbers, thanks to the split of the previous step, we obtain, from the merge, an ordered sequence. After removing the special keys, these sequences can then be appended after the previous ones in the final output array, following the order of the columns.

We implemented this step with a for cycle on each of the **S** columns. For each column a sequence of GPU functions is used: the first applies the padding to the subsequences until their size reaches 128, the second copies the resulting segments in a buffer array, then the function used in the step 2 is called on the buffer to reorder it, and finally the last function copies the values from the buffer to the output array skipping the special keys.

At the end of this step the sort is complete.

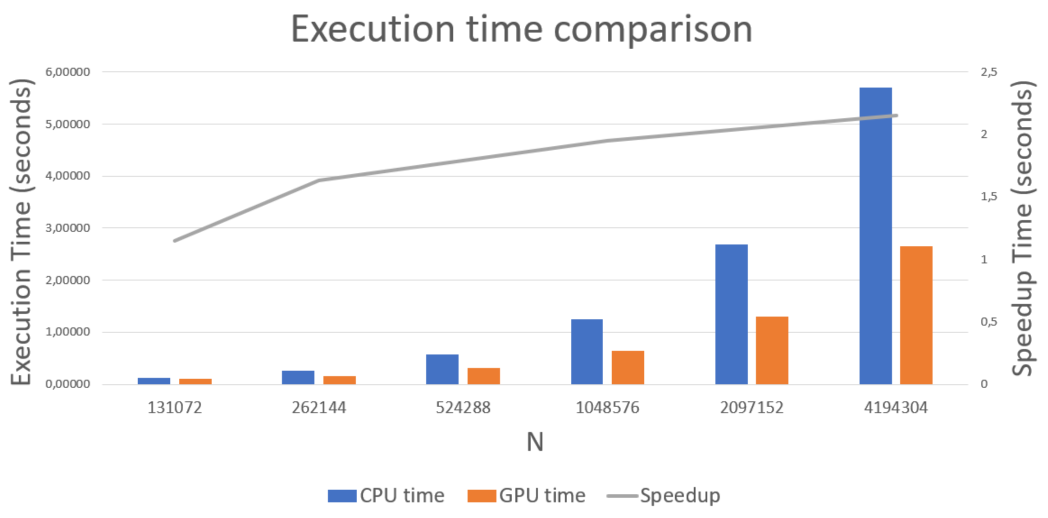

Figure 2: Comparison between average execution times of the implemented algorithm with the sequential bitonic sort algorithm.

3 Testing & Results

To evaluate the effectiveness of the algorithm we implemented, we tested it against a standard CPU bitonic sort Grossi (2021).

We measured both execution time and transfer time between GPU and host memory while the input array is filled with a uniform random distribution. Our tests consists of 100 computations of different input arrays of varying sizes for each algorithm, we recorded the most relevant average times that can be seen in Figure 2. Results attests that our algorithm is capable of sorting correctly sequences from 128 elements to more of 4 million elements. We evaluated that with an input array of size 131072 or greater the GPU-Warpsort gains ever-increasing speedups on the sequential counterpart, with a more than 100% speedup with N equal to 4194304.

4 Discussion

Our result state that our implemented algorithm is faster than a sequential bitonic sort but it is not faster than the original algorithm shown in the original paper by Ye et al. (2010). Not having the original implementation we are not able to certainly explain how our algorithm is slower than the original one. Although, we note that in their evaluation, differently than us, they do not take into consideration the memory transfer times.

Another consideration we have is on the determination of the **S** and **L** parameters used in the step 3 and 4 of the algorithm. **S** has to be big enough that the subsequences resulting from step 3 have size not greater than 128, in order to guarantee this **S** has to increase as the size of the input array **N** increases. As mentioned in the implementation of step 3, to determine the subsequences of each row, a block per row and a thread for each subsequence are used. This causes conflict with the previous prerequisite, as for bigger **N**s, **S** will eventually be greater than 1024. Our GPU is not able to support more than 1024 thread per block, so we had to increase the number of blocks used for each row.

We noticed also in our experiments that increasing **L** improves the speed of our algorithm as allows us to decrease **S** while maintaining the size of the subsequences matrix to be used in step 4. This is because in our implementation of the step, a for cycle is called **S** times (once for each column) on the host. This step is the speed bottleneck of our algorithm.

5 Conclusions

Our results demonstrate up to 100% better performance than the CPU bitonic sort.

We have also confirmed that efficient parallelism in GPU sort algorithms is achieved by using warp-based synchronization and coalesced memory access.

Further optimization works would surely include the for cycle of the last step mentioned above and experiments using a shared memory access instead of separate accesses for host and GPU data.

Acknowledgements

We thank Professor Giuliano Grossi for his support and guidance during the realization of the project.

References

Grossi, G. (2021). Lab 9 sort - bitonic sort cpu / gpu comparison. https://github.com/giulianogrossi/GPUcomputing/blob/master/lab9/sort/bitonic.cu.

Ye, X., Fan, D., Lin, W., Yuan, N., and Ienne, P. (2010). High performance comparison-based sorting algorithm on many-core gpus. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10.