Parallel Sorting in CUDA C

Architectures and Platforms for Artificial Intelligence

Riccardo Paolini
Master's Degree in Artificial Intelligence, University of Bologna
riccardo.paolini5@studio.unibo.it

Abstract

In this project two sorting algorithms, Bitonic and Merge sort, have been re-implemented in their parallel versions using the CUDA extension for the C programming language. Details about the parallel implementations are given in this report as well as the performance measured against their sequential versions.

1 Introduction

Sorting is one of the most popular problems in computer science, and many algorithms have been proposed over the decades. However, traditional sequential sorting algorithms can be prohibitively slow for large sequences, leading to the development of parallel sorting algorithms that can leverage the power of parallel computing architectures to achieve faster sorting speeds.

One such architecture is CUDA (Compute Unified Device Architecture), developed by NVIDIA for their Graphics Processing Units (GPUs). CUDA enables developers to write parallel programs in a C-like language, allowing for efficient use of the massive parallelism offered by modern GPUs.

This report aims to investigate bitonic and merge parallel sorting algorithms implemented using CUDA C. We will compare the performance of these algorithms against their sequential versions on large sequences.

2 Bitonic sort

Bitonic sorting is based on the creation of bitonic sequences that are sequences of elements $x_0, .., x_i, .., x_n$ in which we can find an index i such that the elements on the left of x_i are monotonically non-decreasing and the ones on the right of x_i are monotonically non-increasing (Bhargav). For instance, the array [4, 6, 11, 27, 22, 17, 10, 1] is a bitonic sequence. In fact, its monotonicity changes at 27: $4 \leq 6 \leq 11 \leq 27$ and $27 \geq 22 \geq 17 \geq 10 \geq 1$]. The original implementation of this algorithm is recursive, but we will use the iterative one, since stack problems may occur in CUDA during recursive calls.

The algorithm requires $\log_2 n$ steps, $s \in \{1, 2, .., \log_2 n\}$, to sort the whole array. In each of these steps it deals with $\frac{n}{2^s}$ bitonic sequences of 2^s elements each. These sequences are ordered in multiple stages by comparing, and eventually swapping, the elements at a distance d which depends on the current stage. In particular, d is halved at each stage going from $\frac{2^s}{2}$ down to 1. The direction of the ordering is dependent on both the stage and the step. Once the ordering of these sub-sequences is terminated the following step starts, these process continues until the whole sequence is sorted. A visual description of the algorithm is provided by Figure 1.

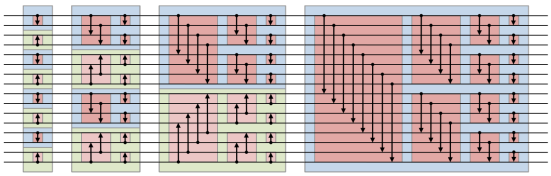

Figure 1: Bitonic sort: the **steps** are colored in blue or green, while the **stages** are in red.

All in all the bitonic sort has a time complexity of $O(n \log^2 n)$ but ideally all the operations within the same stage can be perfectly parallelized using $\frac{n}{2}$ processors bringing down the complexity to $O(log^2 n)$.

Pros This algorithm is extremely easy to be parallelized since the operations belonging to the same stage are independent from each other. Plus, each thread will receive the same amount of work to be done, this means that synchronization among threads and blocks will not produce large overheads.

Cons It can only process sequences with a number of elements which is a power of two. Therefore we have to consume additional memory to store dummy values. This is the only way to handle sequences of arbitrary size.

3 Merge sort

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

This procedure is usually performed by recursively halving the arrays until we get subarrays containing only one element and then merging them back in pairs up to the original size. However, it is possible to do this procedure iteratively by directly merging together single elements of the original array and continuing merging these pairs up to the original size as shown in Figure 2.

The algorithm requires $\log_2 n$ merging steps, $s \in \{1, 2, .., \log_2 n\}$. At each step $\frac{n}{2^s}$ merges are performed. Every merge operation requires 2^s comparisons between pairs of elements in the two arrays to be joined.

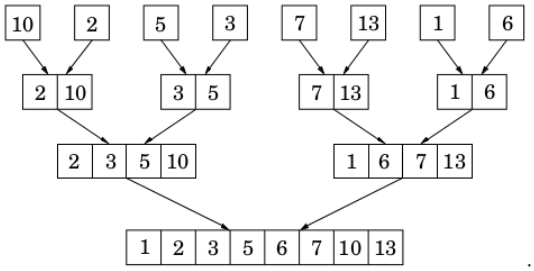

Figure 2: Merge sort

The merge sort has a time complexity of $O(n \log n)$ which candidates this algorithm to be one of the best sequential sorting algorithms. However, the way in which the merges are performed constitutes an important bottleneck for the parallel version since in the original implementation the comparisons must be executed sequentially not allowing multiple threads to be exploited.

Pros This algorithm is conceptually really simple and, ideally, it is also simple to be parallelized. Its sequential version is already very efficient.

Cons The merge procedure constitutes a huge bottleneck in the final steps, when there are few pairs of very long sequences to be joined. In fact, in its original version only one processor works on the merge procedure, so it is not possible to distribute the work among processors and use all the available computing power.

4 Parallel Bitonic sort

The idea behind the parallelization of bitonic sort is to partition the data according to the current step, in fact it is possible to notice that at each step the distance of the furthest comparison doubles.

While this distance is less than or equal to $4{\cdot}1024$ elements we can make all the computations inside the *shared memory*. Indeed the maximum number of bytes that can be allocated in shared memory is $12 \cdot 1024 \cdot 4 = 49152$B meaning that only $12 \cdot 1024$ integers can be stored locally, however we have to work with powers of two, so we will store at most $8 \cdot 1024$ elements in the *shared memory*.

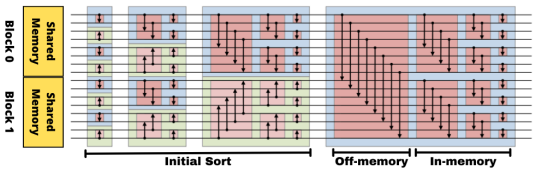

Figure 3: Sorting steps: an **initial sorting** is performed among the data fitting the shared memory, then in the following steps the first stages are performed **off-memory** and the last ones are performed **in-memory**.

Initial sort During these initial steps the data is transferred inside the *shared memory* of the different blocks, then it is processed by the threads belonging to each block. In particular, the $i\text{-}th$ thread will compare, and eventually swap, the elements located at positions that are multiples of the *block size* offset by i (i.e., $\{i, i + block\_size, i + 2 \cdot block\_size, ..\}$). Moreover, threads are synchronized after each stage while it is not necessary to synchronize blocks since we are working on local data. For this reason these first ordering steps are computed inside the same *kernel*.

Off-memory sort When the initial sorting has been performed we reach a stage in which it is not convenient anymore to store the data inside the *shared memory* given that is not reusable. Thus, operations are performed directly on *global memory* and it becomes really important to access the memory as efficiently as possible exploiting *memory coalescing*. This can be achieved by having

consecutive threads accessing consecutive memory locations. This time we need to synchronize at the block level, therefore a new *kernel* is started at each stage. However, only a few steps are computed off-memory; in fact, as soon as we reach a stage within the current step where the data can be reused in subsequent stages, it starts being processed locally again.

In-memory sort When we work in *shared memory*, we perform the same operations as in the initial sorting, but in this case we perform only the remaining stages of the current step before moving off-memory again.

5 Parallel Merge sort

In this case parallelization is much harder to be achieved efficiently. It is not possible to merge two subarrays by iteratively comparing pairs of values because this procedure must be accomplished sequentially and does not allow the use of multiple threads. So, the merge function is replaced by a binary search that can be performed simultaneously by multiple threads, each thread uses a different element as a target for the search.

Merge procedure The idea is that, given an element x_i belonging to the first of the two subarrays to be merged, we count the number of elements in the second subarray which are smaller than x_i, then the final position of x_i will be $i + count$. Differently, given an element y_i of the second subarray, we count the number of elements in the first subarray which are smaller than or equal to y_i. The fact that, in the second case, we have to count also the numbers that are equal to the current element is necessary to get unique indexes when some number is present in both arrays.

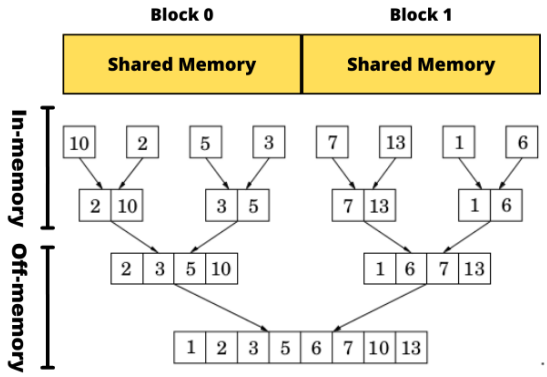

Figure 4: Sorting steps: the first steps are performed **in-memory** and the last ones are performed **off-memory**.

In-memory sort Initially, we split the original array in different blocks such that each block contains $6 \cdot 1024$ elements. We use only half of the memory because the other half is occupied by an auxiliary array that is required to store intermediate results. Then, we start the merging process from subarrays containing only one element. As before each threads is responsible for the executions concerning the elements at positions that are multiples of the *block size* plus an offset dependent on the thread number. After each reordering step we have to synchronize threads and copy the data from the auxiliary array back to the original one. The size of the subarrays is doubled at each step until they contain $6 \cdot 1024$ elements. At his point we start using *global memory*.

Off-memory sort From now on we will always use the *global memory*. The procedure is equal to the one we use when the work is done using *shared memory* but, as before, we ensure that subsequent threads access contiguous areas of memory to make accesses more efficient. Block synchronization is required, so a new *kernel* is invoked for each step.

6 Program evaluation

This section will report the method used to evaluate the program and the results obtained.

To test the sorting algorithms, arrays of different lengths were created. In particular, the number of elements ranges from 1024 to 512 million, all of them are randomly generated integers. Given the capacity of the available GPU, 3005MB, it is not possible to test larger arrays.

To verify the correctness of the parallel versions, their results are compared with those of the respective sequential version but this happens up to 1 million elements (in Figure 5 and 6 sequential algorithms are tested up to 32 million elements), after this threshold, for time reasons, I only checked whether the output arrays are monotonically increasing.

Performances The metric used to compare performance between different algorithms and versions is *throughput*, which, in our case, represents the number of elements sorted per second. Then, the *speed-up* will simply be the ratio between the time required by the CPU and the time required by the GPU to complete the sorting (Harris et al.).

As can be seen from Figure 5, the CPU implementation shows continuously decreasing through-

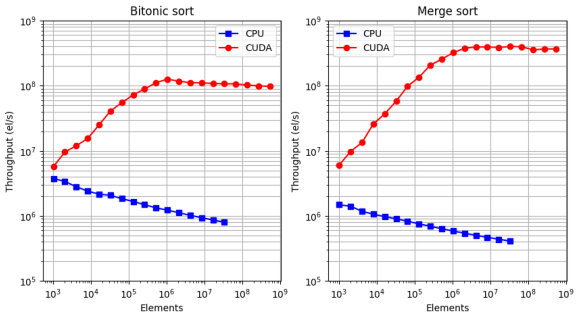

Figure 5: Throughput comparison.

put, while the GPU implementation shows increasing throughtput up to a certain number of input elements and then stabilizes at much higher rates than the sequential version. Moreover, we can observe that parallel merge sort outperforms parallel bitonic sort by almost four times when the maximum throughput is reached. The fact that the sequential version of merge sort is so inefficient is due to the re-implementation carried out to make the parallel version efficient.

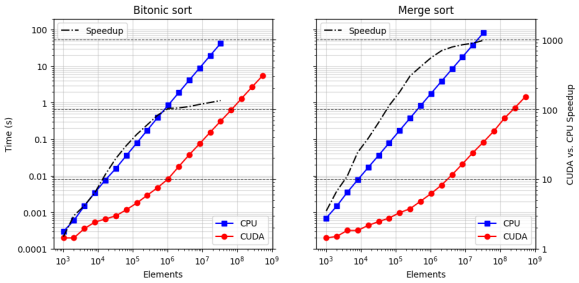

Figure 6: Time comparison and speed-up.

In Figure 6, it is possible to notice that parallel algorithms have a sub-linear complexity up to a certain number of elements and then, as expected, the time complexity tends toward the theoretical one of the sequential algorithm since the number of blocks and threads cannot increase indefinitely. The maximum speed-up obtained is about one undred for bitonic sort and one thousand for merge sort. Although the speed-up of merge sort is an order of magnitude greater, it is only four times faster than bitonic sort.

All in all, if we need to sort arrays containing billions of elements, we have to use GPUs with more memory capacity, and if we want to scale performance even more, we would require GPUs with more CUDA cores so as to maintain sub-linear complexity for longer.

7 Conclusion

In this project, it was demonstrated how GPUs can be exploited to run efficient parallel programs using the CUDA extension for C. Specifically, bitonic and merge sort were parallelized by trying to modify the original code as little as possible, but changing the parts that made the workload more distributed and balanced, while also keeping in mind the efficiency of memory accesses.

The experiments and their results showed the effectiveness of parallel programming in scaling the performances of specific algorithms. But, often times it is necessary to revisit or completely change some parts of the original (sequential) algorithms and this is one of the main challenges in working with such programming paradigm. Another major drawback on the hardware side is that we are limited by the memory of the device to store data.

In any case, the use of parallel programming and related hardware may be extremely important in a future where we will need more and more computing power and where processors will not improve at the rate of previous decades.

8 Links to external resources

Link to Github repo: https://github.com/Paulaner99/Parallel_Sorting_CUDA

References

Nikhil Bhargav. https://www.baeldung.com/cs/bitonic-sort.

Mark Harris, Shubhabrata Sengupta, and John D. Owens. Chapter 39. Parallel Prefix Sum (Scan) with CUDA.