Report of Architecture and Platforms for Artificial Intelligence, Module 1

Giuseppe Tanzi 0001058210

July 2023

Introduction

Sorting is a fundamental operation in computing, essential for numerous applications across various domains. Traditional sorting algorithms, while providing accurate results, can be quite time-consuming for large datasets.
The existing sequential sorting algorithms, such as bubble sort, quick sort, radix sort, and merge sort, follow step-by-step procedures that perform comparisons and swaps to gradually organize the data. While these algorithms are conceptually simple to implement, their performance tends to degrade with larger input sizes.
Parallel sorting algorithms, on the other hand, aim to overcome these limitations by dividing the dataset into smaller portions that are processed independently in parallel. This approach allows parallel algorithms to exploit the massive parallelism offered by modern GPUs, resulting in significant performance improvements for sorting operations.
In this report, I will focus on the implementation and analysis of a parallel sorting algorithm in CUDA C. I will evaluate the performance and efficiency of my parallel sorting algorithm,, comparing its execution time with some sequential sorting algorithms.

Proposed solution

I have developed a solution that combines the principles of Radix Sort and Merge Sort. The algorithm leverages the advantages of Radix Sort's non-comparative nature and Merge Sort's divide-and-conquer strategy to achieve efficient parallel sorting.
Merge Sort, a widely used sequential sorting algorithm, follows a divide-and-conquer approach. It recursively divides the input array into smaller subarrays, sorts them individually, and then merges them back together to obtain the final sorted output. Merge Sort's time complexity of $O(n \log(n))$ makes it an attractive choice for sorting tasks.
On the other hand, Radix Sort is a non-comparative sorting algorithm that operates on the digits of the elements being sorted. It distributes the elements into different buckets based on their least significant digit, repeatedly redistributing them based on subsequent digits until the array is sorted. Radix Sort's time complexity of $O(d * (n + k))$ makes it particularly efficient for sorting integers or strings, where d represents the number of digits, n is the number of elements, and k is the range of values.
My solution builds upon the efficiency of Radix Sort by utilizing it as the base sorting algorithm within Merge Sort.

Algorithm

Initially, the input array is divided into partitions, with each partition assigned to a separate thread or processor. Each thread independently applies Radix Sort to its assigned partition. This step ensures that each partition is individually sorted.
Once the partitions are sorted, a parallel merging step is performed. This step combines the sorted partitions in a manner similar to the merging phase of the sequential Merge Sort algorithm. By merging the partitions in parallel, the algorithm effectively combines the benefits of both Radix

Sort and Merge Sort, achieving efficient sorting performance.

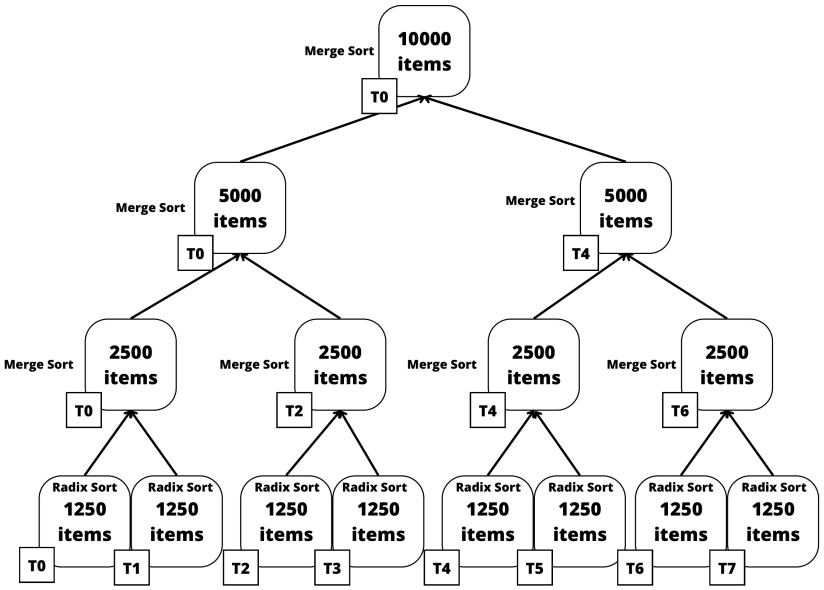


Figure 1: Proposed algorithm

The parallel sorting algorithm follows a multi-step process to efficiently sort an array using parallel computation, with the additional requirement that the number of threads must be a power of 2:

1. **Partitioning**: The input array is divided into multiple partitions, with each partition assigned to a separate thread for processing. The number of partitions is equal to the number of available threads, which is a power of 2.
2. **Sequential Radix Sort**: Each thread performs a sequential radix sort on its assigned partition. This step ensures that the elements within each partition are sorted correctly.
3. **Parallel Merge Sort**: After the sequential radix sort, the parallel merge sort begins. It operates in multiple levels, with each level consisting of pairs of partitions sorted by one of the two threads that performed the sequential radix sort.
    - **Merge Step**: In each level of the merge sort, pairs of partitions are merged by combining the elements from both partitions and sorting them. This process repeats for each level, merging larger and larger portions of the array until the entire array is sorted.

To compute the number of levels in the merge sort based on the number of threads, we can use the following formula:

$$\#LevelsOfMergeSort = \log_2(\#Threads) \tag{1}$$

The formula leverages the fact that at each level of the merge sort, the number of partitions (and hence the number of threads) is halved. Taking the logarithm base 2 of the number of threads gives us the number of times we can halve the number of partitions until we reach a single sorted array.
For example, as you can see in Figure 1, if there are 8 threads available, the number of levels would be:

$$\#LevelsOfMerge = \log_2(\#Threads)$$
$$\#LevelsOfMerge = \log_2(8) = 3$$

This means that the merge sort would have three levels of merging, with pairs of partitions being merged at each level until a single sorted array is obtained.

Parallel algorithm

The parallel sorting algorithm begins by dynamically determining the partition size to enable efficient parallel processing. It calculates the optimal configuration by considering the available hardware resources, such as the number of threads, number of blocks, partition size and other necessary parameters. The configuration function intelligently divides the input array into smaller partitions based on the input size and hardware capabilities, ensuring that each partition can be processed by a separate thread. Additionally, the number of threads is set to a power of 2 to meet the algorithm's requirements.
Next, the algorithm checks the configuration to determine whether a single block or multiple blocks are required for sorting:

- In the case of a single block, the radix sort phase and merging sort phase are combined into a single kernel.
- For scenarios involving multiple blocks, the sorting process is divided into two distinct phases:
    1. First, the entire array undergoes the radix sort phase, utilizing all the necessary blocks.
    2. After completing the radix sort phase, essential indexing and offset information are computed for each participating block in the merging phase. This information is crucial for determining the specific segment of the array that each block will handle during the merging process. The merging phase involves launching the merge kernel for each block. Within each block, individual threads collaborate to merge their respective segments of the array. Consequently, at the end of the loop, multiple lists sorted by each thread within each block are merged.
        - In cases where multiple blocks are involved in the merging phase, a final merging step of the sorted lists, ordered by each block, is executed using a single block.

Shared Memory

Two versions of this parallel sorting algorithm have been tested: one utilizing only global memory and the other incorporating shared memory when available. Shared memory is utilized strategically in various stages of the parallel sorting algorithm. It is employed to optimize memory access and facilitate data sharing among threads within each block. The algorithm checks if shared memory is available and enabled before utilizing it. When there is only one block involved, shared memory is used to combine the radix sort and merging phases into a single operation, enhancing performance by minimizing data movement. In scenarios with multiple blocks, shared memory aids in improving the efficiency of the radix sort phase by allowing faster access to shared data. If multiple blocks are engaged in the merging phase, shared memory may be used to perform the final merge step, optimizing the sorting process by minimizing memory accesses and enhancing thread cooperation. By leveraging shared memory intelligently, the algorithm aims to improve overall performance during parallel execution by optimizing memory access patterns and promoting efficient data sharing among threads.

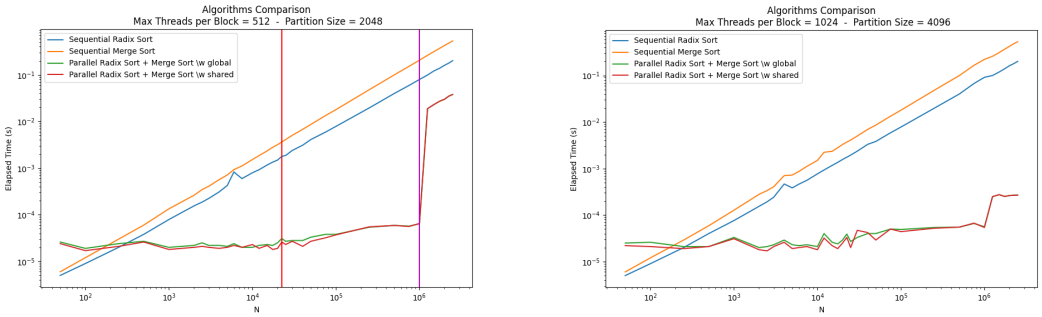
Experiments and Results

The evaluation of the performance involved analyzing the efficiency and scalability of the parallel sorting algorithm compared to sequential radix sort and merge sort approaches. To assess the algorithm's performance, various input data sizes were considered, ranging from 50 elements to 2,500,000 elements. The primary focus was to evaluate the execution times and compare the parallel algorithm's performance to the sequential methods. Additionally, the algorithm's parallel scalability was examined to understand its effectiveness in handling increasingly larger input sizes. During the evaluation, several constants were employed to configure the behavior of the algorithm. These constants are as follows:

- *Warp size*: set to 32, representing the number of threads per warp in a CUDA block. This value is determined based on the hardware architecture and characteristics.

- *Max number of threads per block*: varied for each plot, indicating the maximum number of threads that can be assigned to a single CUDA block. It sets an upper limit on the number of threads that can work together within a block, ensuring efficient utilization of resources and maximizing parallelism.
- *Max size of a thread block*: set to 65535, defining the maximum number of CUDA blocks that can be used in the parallel algorithm. It provides an upper limit on the number of blocks that can be employed for parallel processing, considering hardware limitations and constraints.
- *Partition size*: varied for each plot, indicating the initial size of each partition for parallel processing. The partition size is initially set to this predefined value. However, it is dynamically adjusted based on the input size and the hardware limitations:
  - for smaller input sizes, the partition size remains unchanged.
  - for larger input sizes, it is computed an optimal partition size to ensure efficient utilization of threads and blocks.
- *Min value* and *Max value*: define the range of numbers contained within the input array. These constants establish the minimum and maximum values, respectively, for the dataset being sorted. They are set to 0 and 65535 respectively.

By modifying the values of *Max number of threads per block* and *Partition size* for each plot, the algorithm's scalability and efficiency can be assessed under varying hardware configurations and input sizes. The subsequent analysis of the performance plots will provide valuable insights into the impact of these constants on the algorithm's execution time, parallel scalability, and shared memory utilization.



(a) Behavior with more than one CUDA block

(b) Opmital performance

Figure 2: Performance analysis

As one can see from the plots, the execution time of the sequential algorithms increases gradually as the input size grows. They serve as a baseline for comparing the performance of the parallel algorithms. For what concerns the parallel algorithm, the execution time is slightly higher with small sizes than the sequential algorithms due to the overhead of parallel processing. However, as the input size increases, the parallel algorithm starts to outperform the sequential methods.
In Figure 2a we can see the behaviour of the parallel algorithm when it uses more than one CUDA block. The vertical red line at 22,500 elements indicates the threshold where shared memory is no longer utilized, causing both parallel algorithms to perform similarly. Beyond this point, the introduction of shared memory no longer contributes to performance improvement. The magenta line at 1,000,000 elements signifies the transition to using more than one CUDA block for parallel processing.
Notably, once multiple blocks are employed, a rapid increase in execution time is observed, indicating potential scalability challenges. This rapid increase in execution time when using more than one block suggests the presence of potential bottlenecks in the algorithm's scalability. Factors contributing to this phenomenon may include memory access and communication overhead, load imbalance between blocks, and kernel launch overhead.

In the Figure 2b only a single block is utilized. There is no need for inter-block communication or synchronization, resulting in reduced overhead and improved efficiency. The use of a single block in the parallel algorithms eliminates inter-block communication overhead and enables more efficient memory access, leading to improved performance. It highlights the importance of considering the trade-off between the number of blocks and the associated communication overhead when designing parallel algorithms.
Overall, the Figure 2 demonstrates the effectiveness of the parallel sorting algorithms, showcasing their superior performance compared to sequential approaches, particularly for larger input sizes. The shared memory optimization contributes to improved execution times until a certain input size is reached, after which the algorithms seamlessly adapt to using global memory. This adaptability ensures efficient resource utilization and consistent performance across different input sizes.