Aristotle University of Thessaloniki
Department of Electrical and Computer Engineering
Parallel and Distributed Computer Systems

Nikolaos Kylintireas
January 2025

Source code: https://github.com/NicKylis/Cuda_sort

Cuda Bitonic Sort

1 Abstract

This project is the second part of an assignment for the Parallel and Distributed Systems class of the Aristotle University of Thessaloniki's Electrical and Computer Engineering department, under Professor Nikolaos Pitsianis. In the first part, a bitonic sorting algorithm was implemented based on the Message Passing Interface (MPI) library. Those results will be compared to the current algorithm's performance, which utilizes a Graphics Processing Unit (GPU) using the CUDA programming language.

2 Introduciton

The Bitonic Sort algorithm is a sorting algorithm that operates efficiently for parallel and distributed systems, leveraging the Bitonic sequence properties. In this project, a parallel implementation of Bitonic Sort using CUDA (Compute Unified Device Architecture), a programming language for NVIDIA's GPUs, has been developed to sort an array of integers distributed across the device.

At its core, CUDA employs a hierarchical structure to manage the execution of parallel tasks. This structure comprises three key elements: grids, blocks, and threads.

- Grid:

  Represents the total number of threads launched to execute a kernel function. It can be conceptualized as a collection of blocks, each operating independently.

- Block:

  A collection of threads that are executed together on a single Streaming Multiprocessor (SM) within a GPU. It serves as an intermediate level of thread hierarchy, residing below a grid and above individual threads.

- Thread:

  A single instance of a kernel function. It constitutes the fundamental unit of parallelism in the CUDA execution model.

To efficiently implement Bitonic Sort using CUDA, the algorithm is structured to take full advantage of this hierarchy. Each thread operates on specific portions of the data, while blocks manage shared memory for local computations. Grids coordinate the execution of blocks, enabling parallelism at scale. This hierarchical approach allows the GPU to sort large datasets efficiently by dividing the task into smaller, manageable chunks that are processed in parallel.

The specifics of the code, including the integration of shared memory, synchronization, and kernel execution, will be discussed later in detail, highlighting the strategies used to optimize performance.

3 Algorithm design

In this project, three different versions of the code were created (V0, V1, V2). In all of them, each block on the GPU is assigned a chunk of the overall array, and the blocks work independently to sort their assigned segments using a bitonic sorting algorithm. Afterwards, a multiway merge is performed to combine the results into a fully sorted array. The implementation steps are the following:

- Data Distribution:

  The array of size N is divided into smaller chunks of size equal to the number of threads per block (1024 by default). Each thread block is assigned one chunk of data, and the data is loaded into shared memory for efficient processing on the GPU.

- Local Sorting:

  Within each block, threads collaboratively perform a bitonic sort on the assigned data. Using shared memory ensures high-speed data access, enabling the GPU to sort these chunks efficiently. At this stage, the GPU processes all blocks independently, ensuring parallelism and high throughput.

- Multiway Merge

  After each block is sorted, the segments of the sorted arrays are managed by a merge function in a variety of ways in the three different versions of the algorithm.

The pseudocode below outlines the essential steps involved in sorting and merging the array.

Algorithm 1 Bitonic Sort with CUDA

```
1: Input: $N$: Size of the array, $array$: Array to sort
2: Output: Sorted array in ascending order
3: procedure BitonicSort($N, array$)
4:     Copy $array$ to GPU memory as $d\_array$
5:     Launch bitonicSortLocal kernel to sort blocks in parallel
6:     Copy sorted blocks from $d\_array$ back to $array$
7:     Initialize $chunkSize \leftarrow blockSize$
8:     while $chunkSize < N$ do
9:         for $start \leftarrow 0$ to $N$ in steps of $2 \cdot chunkSize$ do
10:            $mid \leftarrow start + chunkSize$
11:            $end \leftarrow \min(start + 2 \cdot chunkSize, N)$
12:            if $mid < end$ then
13:                Merge($array, start, mid, end$)
14:            end if
15:        end for
16:        $chunkSize \leftarrow chunkSize \cdot 2$
17:    end while
18: end procedure
19: procedure Merge($array, start, mid, end$)
20:    Merge $array[start \dots mid]$ and $array[mid \dots end]$ into a temporary array
21:    Copy the merged array back into $array[start \dots end]$
22: end procedure
```

3.1 V0: the first approach

This was the first implementation of the algorithm, with the main focus being the translation of the MPI project's code into CUDA. It presents a straightforward approach, implementing the basic components of the bitonic sorting algorithm (local sorting, calculation of the exchange partner, and elbow sort/merge).

After each block is sorted, the merging is handled by the GPU. However, this results in numerous memory copies between the GPU and host, many unnecessary communications and global synchronizations. For larger inputs ($N > 2^{16}$), this overhead becomes significant enough to heavily impact the execution time of the algorithm. Additionally, memory demands grow exponentially, making the algorithm impractical for very large arrays ($N > 2^{20}$), or even non-functional due to memory overflows.

3.2 V1: scaling and optimization

Addressing the issues above, V1 was created to reduce the overwhelming overhead for large inputs. This version features a more simple approach to the merging process, resulting in fewer global calls and therefore less memory allocated.

This is achieved by using 2 kernels instead of one in the previous solution. The first one (bitonicLocal) handles all the sorting within the blocks, and the second one (mergeGlobal) performs the final merging. This method proved much more efficient than the V0 version, but was not without setbacks. Although much faster than any other solution so far for medium sized arrays ($2^{10} < N < 2^{18}$), it still lacked the scalability to perform for larger array sizes ($N > 2^{20}$).

3.3 V2: Hybrid model / Optimal solution

Finally, V2 was developed as a hybrid model that utilizes both GPU and CPU resources to optimize performance. The key advantage of this approach lies in minimizing unnecessary memory copies and global synchronizations, which were significant bottlenecks in V0 and V1. Instead, more local memory on the GPU is utilized to handle sorting tasks, significantly reducing the reliance on global memory operations.

Moreover, V2 strategically delegates the final merging phase to the CPU. This allows the GPU to focus on high-throughput, parallel sorting tasks while offloading the almost serial merging operation to the CPU. By combining the strengths of both processing units, this hybrid solution achieves significantly improved scalability and execution time.

With fewer kernel calls and optimized memory usage, V2 delivers superior performance for both medium-sized arrays ($2^{10} < N < 2^{20}$) and larger datasets ($N > 2^{20}$). This makes it the most practical and efficient solution among all versions of the algorithm.

4 Execution Time

4.1 Time Complexity

In the sorting phase, the input is divided into chunks corresponding to the number of blocks available on the GPU. Each chunk is sorted independently within each block using bitonic sort, which operates with a complexity of $O(log^2 M)$, where M is the number of elements processed by each processing unit. Since M is a non-variable size (both blocksize and elements per thread numbers are static), with enough threads we can assume this is dominated by other elements.

The merging phase involves combining the sorted subarrays from each block. This is achieved through a series of parallel merging operations, where the total work done at each recursive level is linear in terms of the input size, O(N). The number of levels required to merge all blocks is logarithmic with respect to the number of blocks, specifically logB, where B is the total number of blocks.

Thus, the time complexity for the merging phase is:

$$T_{\text{merge}}(N, B) = O(N \log B)$$

For sufficiently large N, the logarithmic term $\log B$ grows slower than the linear term $O(N)$. Thus, the overall time complexity of the algorithm is dominated by the $O(N)$ term, and the algorithm behaves almost linearly in terms of execution time:

$$T(N) = O(N)$$

The practical verification of this claim, supported by experimental results, will be presented in the following section.

4.2 Performance Analysis and Results

The performance of the algorithm was evaluated on the same device used for the MPI-based implementation to ensure a fair comparison between the two approaches. The device is equipped with an RTX 3050 GPU featuring 4GB of VRAM. For the initial test, the input size N was set to 2^{20}. While both V0 and V1 exhibit notable challenges in matching the performance of the MPI algorithm, it is evident that V2 outperforms all other versions.

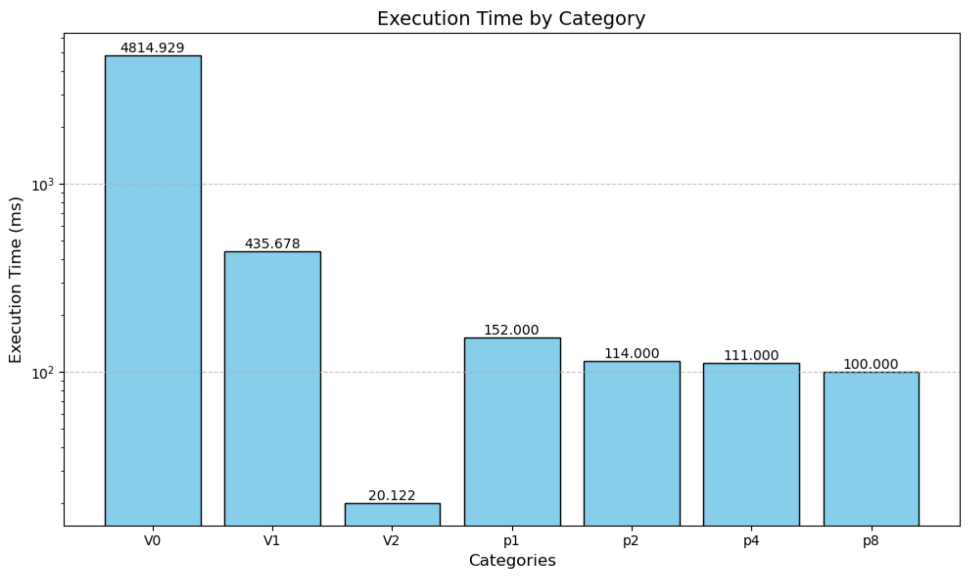

Figure 1: Comparison of execution time between multiple process numbers in the MPI code and all the versions of the CUDA code

In the second test, both the MPI algorithm, utilizing 4 processes, and the V2 solution were evaluated with increasingly larger values of N, specifically for exponents in the range of [20,28]. As shown in Figure 2, the V2 solution not only outperforms the MPI algorithm, but also maintains a near-linear execution time, as previously mentioned. This behavior further demonstrates the scalability of V2 for handling large array sizes efficiently.

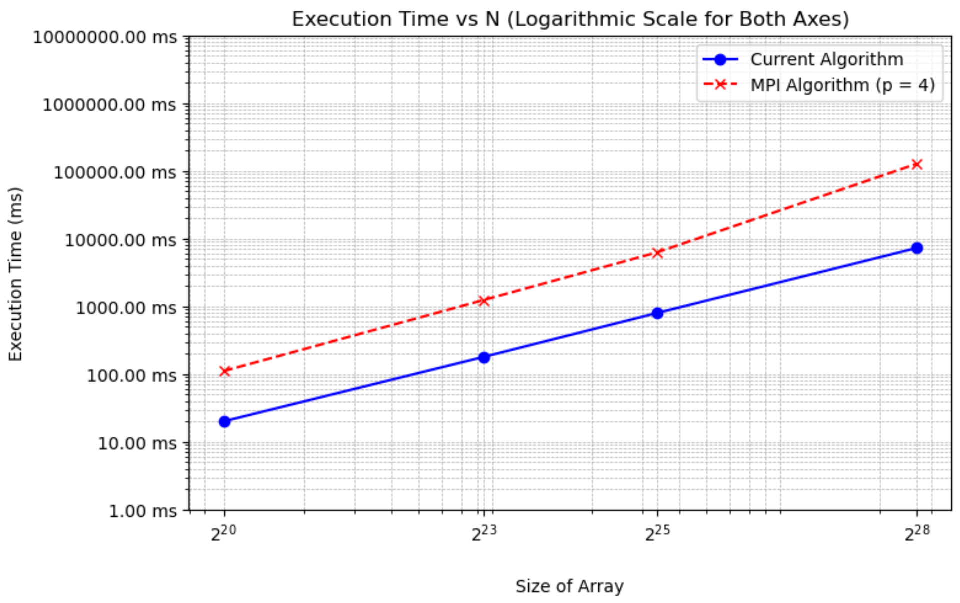

Figure 2: MPI algorithm (red) vs V2 (blue): execution time

All the aforementioned calculations, including those from the first and second tests, were performed within a WSL (Windows Subsystem for Linux) environment. As such, the execution times may be slightly slower than those observed in a native Linux setup.

5 Tools used

The following list contains all the tools used for this project:

- C programming language/compilers/standard libraries
- Visual Studio Code
- Windows Subsystem for Linux (WSL)
- CUDA drivers
- Jupyter notebook for diagrams
- ChatGPT for the plots
- Github Copilot for syntax errors
- Latex with Overleaf
- source control with git/github