Bitonic Sort with CUDA

Savvas Tzanetis
10889

January 20, 2025

Contents

Chapter 1

Abstract

This report is part of an assignment for the **Parallel and Distributed Systems** class of the Aristotle University's Electrical and Computer Engineering department.

This project implements *distributed sorting* using the **Bitonic Sort** algorithm with the use of the **CUDA** platform. The primary objective is to sort a dataset of $N = 2^q$ numbers (where $q \in N$). The implementation employs parallel processing to achieve efficient sorting, making it suitable for large-scale data sets.

The code for this project is available at this GitHub repository page.

Chapter 2

Serial Bitonic

To grasp the concept of distributed **Bitonic Sort**, it is essential to first understand its serial implementation. This foundational knowledge will provide the necessary context for comprehending the distributed version of the algorithm.

2.1 Explanation

Bitonic Sort is a sorting algorithm that operates by sorting and merging bitonic sequences. A bitonic sequence is defined as a sequence of numbers that first monotonically increases and then monotonically decreases (or vise versa), or can be cyclically rotated to exhibit this pattern. Note that sorted sequences are also bitonic. For example, the sequence $6, 4, 3, 1, 2, 5, 8, 7$ is bitonic because it can be rotated to $7, 6, 4, 3, 1, 2, 5, 8$.

The algorithm is has the following characteristics:

- $log(n)$ steps, where n is the total amount of numbers, with $log(step)$ comparison phases in each step.
- In each comparison phase, we swap elements using the **min-max** criteria. The min-max pattern is presented in the image bellow:

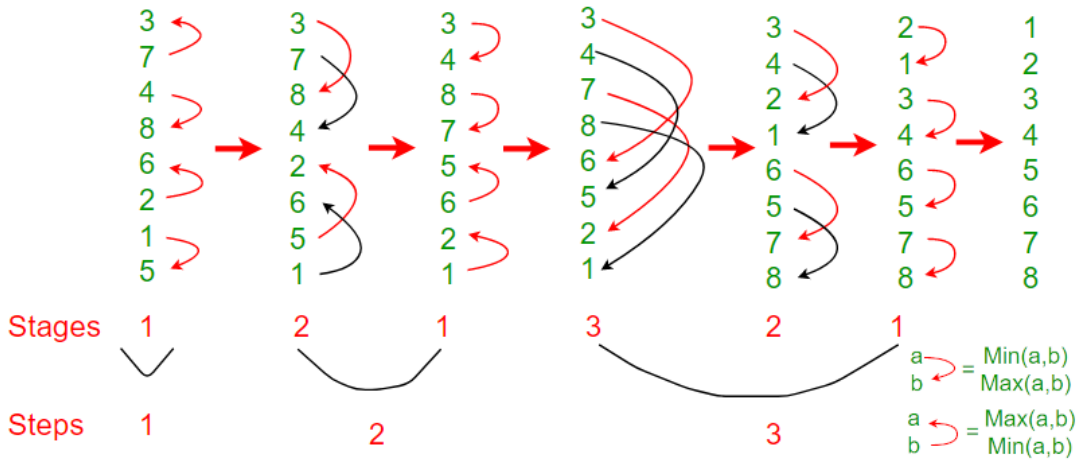


Figure 2.1: Bitonic Sort

2.2 Example

To illustrate the process, consider the following random sequence:

$$S\{3, 7, 4, 8, 6, 2, 1, 5\}$$

In each step each element is compared to its neighbor with distance **stage**.

1. **Step 1, Stage 1 min-max pattern:** *min max max min min max max min*

   $$S\{3, 7, 8, 4, 2, 6, 5, 1\}$$

2. **Step 2, Stage 2 min-max pattern:** *min min max max max max min min*

   $$S\{3, 4, 8, 7, 5, 6, 2, 1\}$$

3. **Step 2, Stage 1 min-max pattern:** *min max min max max min max min*

   $$S\{3, 4, 8, 7, 6, 5, 2, 1\}$$

4. **Step 3, Stage 3 min-max pattern:** *min min min min max max max max*

   $$S\{3, 4, 2, 1, 6, 5, 7, 8\}$$

5. **Step 3, Stage 2 min-max pattern:** *min min max max min min max max*

   $$S\{2, 1, 3, 4, 6, 5, 7, 8\}$$

6. **Step 3, Stage 1 min-max pattern:** *min max min max min max min max*

   $$S\{1, 2, 3, 4, 5, 6, 7, 8\}$$

2.3 Remarks

1. The complexity of this algorithm is $O(nlog^2n)$. While it is higher than other popular sorting algorithms like **Merge Sort** or **Quick Sort**, **Bitonic Sort** is ideal for parallel implementation, because it always compares elements in a predefined sequence and the sequence of comparison does not depend on data.

2. **Bitonic Sort** can only be used if the number of elements to sort is 2^n. The procedure fails if the number of elements is not in the aforementioned quantity precisely.

Chapter 3

Distributed Bitonic

As mentioned above, this algorithm, when implemented in a serial manner, has a time complexity of $\boldsymbol{O(n \log^2(n))}$, which is higher than most sorting algorithms. Despite that, this algorithm is useful, as it is well-suited for parallel implementations, like the one we will be discussing. For a parallel implementation, we can achieve a time complexity of $\boldsymbol{O(\log^2(n))}$, which is significantly faster than our serial implementation. We will be achieving this goal by utilizing the **CUDA** platform for *GPUs*.

This assignment was split across three versions that needed to be implemented, each one being more advanced than the previous. More specifically:

- **V0:** In this first version we were tasked with implementing a kernel where each thread only compares and exchanges. Being easy to write, but requiring too many function calls and global synchronizations.
- **V1:** The second version needed to be faster than V0. More specifically we needed to create an implementation that required fewer calls global synchronizations.
- **V2:** Lastly, we where tasked to modify the kernel of V1 to work with local memory instead of global for even faster results.

3.1 Generating Random Sequences

The sequences we used to test our results were generated using the **rand()** function, which is part of the **C++ standard library**. More specifically, we selected *integers* ranging from **1** to **999** for simplicity.

```
void generateArray(int *values, int N) {
    std::srand(std::time(0));

    for (int i = 0; i < N; i++) {
        values[i] = std::rand() % 1000; // 0 - 999
    }
}
```

3.2 Synchronization

For a parallel implementation of the serial algorithm, we need to implement a method to exchange data between each sub-array for every step of the algorithm.

In **CUDA**, this is achieved by calling a function that will be executed by every thread inside a block, and synchronizing the results after each iteration.

```
for (int stage = 2; stage <= N; stage <<= 1) {
    for (int step = stage >> 1; step > 0; step = step >> 1) {
        bitonicSortStep<<<blocks, threads_per_block>>>
                (dev_values, stage, step);
        cudaDeviceSynchronize();
    }
}
```

3.3 Exchanging Data Between Partners

Data exchanges are done after each thread calculates its *partner*. This is done using the logical operation **XOR** *(tid ŝtep)*. The threads are arranged in a *hypercube-like* structure, where the global **ID** of each thread is treated as a binary number. The **XOR** operation between two binary numbers results in a number where the bits are set to 1 at positions where the two numbers differ. The distance between two nodes in a hypercube is the number of differing bits between their binary representations, known as the **Hamming Distance**.

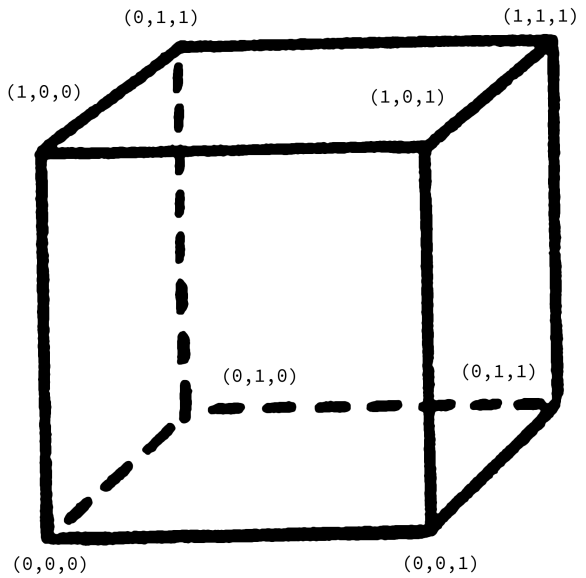


Figure 3.1: Enter Caption

3.4 V0

This first version of our parallel implementation is very simple. Using the simple concepts mentioned above, a significant improvement compared to a serial implementation is observed. But, while its performance is significantly faster, it still has downsides. Launching kernels multiple times can incur additional overhead, as each launch involves setting up the execution environment on the GPU. Also, multiple synchronizations need to be established, and considering that global synchronizations are generally slow operations, there is a big room for improvement in this implementation.

3.5 V1

The V1 version of the bitonic sort implementation introduces several key differences compared to the initial version. One of the most notable changes is the introduction of a second kernel, **localSort**, which is executed alongside the **bitonicSortStep** kernel we see in the first version (V0). The **localSort** function is designed to handle the initial sorting of elements within each block by using shared memory for faster data access and reducing the need for frequent global memory calls.

By performing local sorting within each block, the **localSort** kernel helps to set up the array for the subsequent stages of the bitonic sort.

```
void bitonicSort(int *values, int N) {
    int *dev_values;
    size_t size = N * sizeof(int);
    int threads_per_block = 1024;
    int blocks = ((N/2) - 1) / threads_per_block + 1;

    cudaMalloc((void**)&dev_values, size);
    cudaMemcpy(dev_values, values, size, cudaMemcpyHostToDevice);

    localSort<<<blocks, threads_per_block>>>
            (dev_values, N, 2, 1);

    for (int stage = 2048; stage <= N; stage <<= 1) {
        for (int step = stage >> 1; step > 512; step >>= 1) {
            bitonicSortStep<<<blocks, threads_per_block>>>
                    (dev_values, (N/2), stage, step);
            cudaDeviceSynchronize();
        }
        localSort<<<blocks, threads_per_block>>>
                (dev_values, N, stage, 512);
    }

    cudaMemcpy(values, dev_values, size, cudaMemcpyDeviceToHost);
    cudaFree(dev_values);
}
```

3.6 V2

The last version of our algorithm improves upon the *V1* version by utilizing local memory instead of global. This is done inside the **localSort** kernel that we discussed about in the previous version by copying data into shared memory at the beginning, allowing threads within the same block to access data more quickly and efficiently. After the local sorting, the data is written back to global memory, minimizing the overhead associated with frequent global memory accesses.

```
__global__ void localSort(int *dev_values, int N, int stage, int step) {
    extern __shared__ int shared_values[];
    unsigned int tid = threadIdx.x + blockDim.x * blockIdx.x;
    unsigned int offset = N >> 1;
    unsigned int local_tid = threadIdx.x;

    shared_values[local_tid] = dev_values[tid];
    shared_values[local_tid + min(offset, blockDim.x)] =
                                dev_values[tid + offset];
    __syncthreads();
}
```

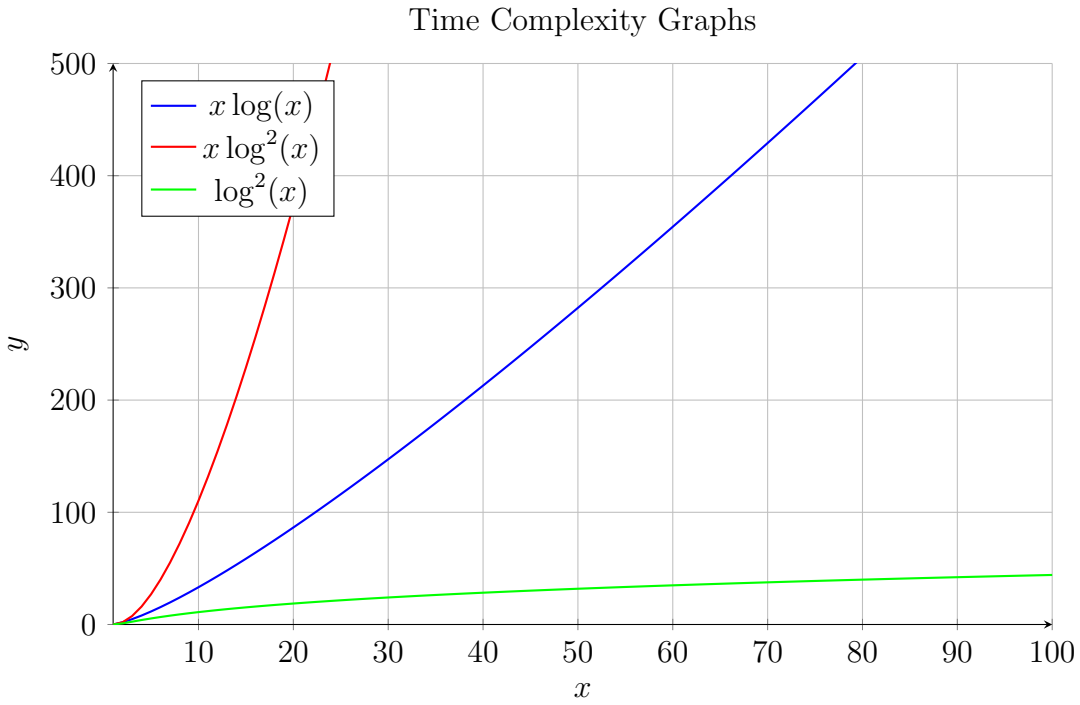
Chapter 4

Results

As previously mentioned, a serial version of this algorithm has a time complexity of $\boldsymbol{O(n \log^2(n))}$, which is not ideal, as there are faster algorithms. However, a parallel or distributed version, significantly lessens this to $\boldsymbol{O(\log^2(n))}$.

Performance Table	**Serial**	**Distributed**	**Quick Sort**
Time Complexity	$\boldsymbol{O(n \log^2(n))}$	$\boldsymbol{O(\log^2(n))}$	$\boldsymbol{O(n \log(n))}$

Table 4.1: Performance Table



This program was executed on a **Windows 11** machine with a **NVIDIA Geforce RTX 3060 12GB** graphics card. We tested our program across a range of n values ([20:29]) where the random array (to be sorted) was populated with integers in the range **1–999**. We then verified the correctness of the results by iterating through the array and comparing each element with the next one.

In the graphic below, we can see the results of each version of our implementation compared to **qsort** for aforementioned N values range. This graphic was created using the *test.py* script that is also available at this projects GitHub repository .

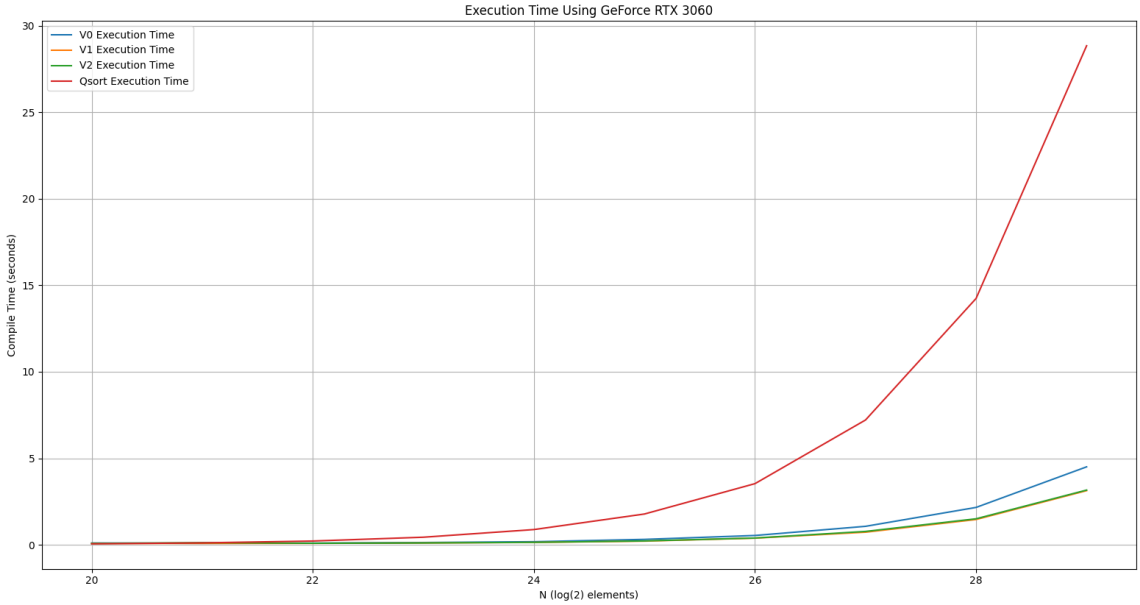


Figure 4.1: Performance results

Chapter 5

Tools and Sources

In this project, the following tools where used:

1. The **C++** programming language.
2. The **CUDA** Library.
3. The **Aristotle Cluster** for testing.
4. **GitHub** for version control.
5. **GitHub Copilot** as an AI assistant.
6. **Python** with **matplotlib** for graphics

The following sources were helpful for understanding the problem presented in the assignment:

- `https://en.wikipedia.org/wiki/Bitonic_sorter`
- `https://www.geeksforgeeks.org/bitonic-sort/`
- Lecture notes from the **Parallel and Distributed Systems** course, written by professor **Nikolaos Pitsianis**.