King Saud University

CSC453: Parallel Processing

Project Report Template

First Semester 2025

Project Title:

An Experimental Study of Sequential and CUDA Implementations of Bitonic Sort

Rana alhamdan, Nouf alawad, Nouf alsabti, Lojien alabo Omar, yara Alanazi

Computer Science Department

College of Computer and Information Sciences

King Saud University

{444201131, 444200547, 444201021, 444200080, 444200864} @KSU.EDU.SA

Abstract

Sorting is a fundamental operation in computer science, used in databases, searching, scheduling, memory management, and many real-time systems. When dataset sizes grow large, sequential sorting causes serious latency. -9To overcome this problem, parallel sorting algorithms are provided. A main one is the Bitonic Sort, which is naturally suitable for parallel architectures such as GPUs due to its dependency on a fixed, data-independent sorting network. In this paper, we compared the sequential and CUDA Bitonic Sort versions, showing the high speedup that CUDA Bitonic Sort can provide for medium and large datasets, and the enhancement of speedup as dataset size grows. Each version was tested 10 times on arrays of three different sizes. We also detected the preference to use the sequential version over the parallel one for small datasets, where parallel execution shows negative speedup. Although the parallel version provides big improvement, there are still some limitations such as the limit of scalability of Bitonic Sort due to GPU memory, and the instability of the algorithm.

1 Introduction

Sorting algorithms play a fundamental role in computer science because many core tasks rely on having data in a sorted structure. For example, searching becomes significantly faster when the input is sorted. Sorting is also essential in databases, where operations such as removing duplicates. In data analysis, sorted data helps identify minimum or maximum values, generate ranked lists, and prepare data for visualization or aggregation. As data continues to grow, the need for faster and more scalable sorting methods becomes increasingly important[1].

Parallel sorting on GPUs offers a major performance advantage because GPUs are capable of executing thousands of operations simultaneously[1]. Bitonic Sort, in particular, is well-suited for GPU implementation due to its regular and predictable compare–swap pattern, which maps naturally onto parallel hardware. The aim of this project is to implement both sequential and CUDA-based versions of Bitonic Sort and compare their performance.

2 Problem Definition

The problem addressed in this project is the efficient sorting of an integer array using the Bitonic Sort algorithm in both sequential and parallel GPU environments. Sorting is a fundamental operation in computer science, used in databases, searching, scheduling, memory management, and many real-time systems. When dataset sizes grow large, sequential sorting becomes a performance bottleneck, making parallel sorting techniques essential for achieving high throughput and low latency.

Bitonic Sort is particularly suitable for parallel architectures such as GPUs because it is based on a fixed, data-independent *sorting* network [2]. A sorting network is a pre-defined, fixed sequence of comparators whose execution order does not depend on the input data, enabling deterministic and fully parallel compare and swap operations. This means that the sequence of comparisons does not depend on the input values, allowing thousands of GPU threads to perform compare-and-swap operations simultaneously without complex branching or synchronization.

Mathematically, Bitonic Sort relies on the concept of a **bitonic sequence**, a sequence that first increases then decreases, or vice-versa. Formally, a sequence $\langle a_0, a_1, \dots, a_{n-1} \rangle$ is bitonic if there exists an index **K** such that:
$a_0 \leq a_1 \leq \dots \leq a_k$ **and** $a_k \geq a_{k+1} \geq \dots \geq a_{n-1}$.
Any bitonic sequence can be converted into a sorted sequence through a structured process known as bitonic merge.

From a mathematical standpoint, the bitonic merge works by comparing elements that differ by a distance of 2^j at stage j, recursively splitting the sequence into smaller bitonic subsequences at each step. After log(n) comparison stages, the bitonic sequence becomes completely sorted. This regular mathematical structure enables Bitonic Sort to be implemented as a fixed sorting network.

A **bitonic sorting network** consists of log(n) major steps, each composed of log(n) comparison-and-swap stages, giving the algorithm a theoretical complexity of **$O(\log^2 n)$.** Because the network structure is completely fixed, every thread knows exactly which pair of indices to compare. This sequential independence makes the algorithm highly suitable for the SIMT (Single Instruction, Multiple Threads) execution model used by modern GPUs.

On the GPU, each thread is assigned a specific pair of elements to compare based on the XOR pattern defined by the network. With thousands of threads running concurrently, Bitonic Sort achieves substantial speedup compared to its sequential CPU version, especially for large input sizes.

This project focuses on understanding the mathematics of bitonic sequences, implementing the Bitonic Sort algorithm in C (sequential) and CUDA (parallel), and comparing their performance to highlight the benefits of GPU-based parallelism.

3 Sequential Algorithm

3.1 Pseudo Code

```
Algorithm Sequential_BitonicSort(A, N)
    if N is not a power of 2 then
        print "Array size must be a power of 2"
        return

for k = 2 to N, multiply k by 2 each step
 for j = k / 2 down to 1, divide j by 2 each step
for i = 0 to N - 1
  partner = i XOR j
  if partner > i
  if (i AND k) == 0
     direction = ascending
  else
     direction = descending
 if direction is ascending and A[i] > A[partner]
 swap A[i] and A[partner]
 if direction is descending and A[i] < A[partner]
 swap A[i] and A[partner]

Algorithm Random(A, N)
    for i = 0 to N - 1
        A[i] = random integer between 0 and 1,000,000

Algorithm MeasureTime(N, runs)
    create array A of size N
    totalTime = 0
    repeat runs times
        Random(A, N)
        start = current clock time
        Sequential_BitonicSort(A, N)
        end = current clock time
        totalTime = totalTime + (end - start)
    averageTime = totalTime / runs
    return averageTime

Algorithm Run()
    N = 8
    create array A of size N
    print "Enter 8 integers:"
    for i = 0 to N - 1
        read A[i]
    print "Before sorting:"
    print A
    Sequential_BitonicSort(A, N)
    print "After sorting:"
    print A

Algorithm RunExperiment()
    runs = 10
    sizes = [2^10, 2^15, 2^20]
    print "Sequential Bitonic Sort Results (average time)"
    for each N in sizes
        avg = MeasureTime(N, runs)
        print N, avg

Algorithm Main()
    repeat
        print "1) Sort 8 numbers"
        print "2) Run performance test"
        print "0) Exit"
        read choice
        if choice == 1
            Run()
        if choice == 2
            RunExperiment()
    until choice == 0
```

3.2 Performance

The time complexity of Bitonic Sort is $O(n\log^2 n)$.
This comes from the structure of the algorithm: the outer loop **(k)** runs $\log n$times, the middle loop **(j)** also runs $\boldsymbol{\log n}$ times for each k, and the **inner loop (i)** processes all $\boldsymbol{N}$ elements. Multiplying these together gives $\boldsymbol{n \times \log\ n \times \log\ n = O(n\log^2 n)}$**.**

The experiments for the sequential Bitonic Sort implementation were executed on a Google Colab environment. The system specifications were obtained using the commands lscpu, nproc, and free -h.

The CPU is an **Intel(R) Xeon(R) processor running at 2.20GHz**, and the machine provides **2** virtual CPU cores. The available system **memory is approximately 12 GB** of RAM.

These specifications are reported because execution time depends heavily on the underlying hardware and ensures that the results can be interpreted and reproduced accurately.

The Sequential Bitonic Sort Results (average of 10 runs):

Size	Time (s)
1024	0.000441
32768	0.021009
1048576	1.220847

We repeated each experiment 10 times and reported the average execution time because the results can change slightly from one run to another. This happens mainly because a new random array is generated before every run, so the input is never exactly the same. Taking the average makes the final number more representative and reliable than using only a single measurement.

4 CUDA Algorithm

4.1 Pseudo Code

```
Algorithm CUDA_BitonicKernel(A_device, j, k, N)
    idx  = thread_index_in_grid
    pair = idx XOR j
    if pair > idx and idx < N and pair < N
        if (idx AND k) == 0
            direction = ascending
        else
            direction = descending
        if direction is ascending and A_device[idx] > A_device[pair]
            swap A_device[idx] and A_device[pair]
        if direction is descending and A_device[idx] < A_device[pair]
            swap A_device[idx] and A_device[pair]

Algorithm Random(A_host, N)
    for i = 0 to N - 1
        A_host[i] = random integer between 0 and 1,000,000

Algorithm GPU_BitonicSort(A_host, N, runs)
    allocate A_device of size N on GPU
    copy A_host to A_device
    threadsPerBlock = 512
    blocks = ceil(N / threadsPerBlock)
    totalTime = 0
    repeat runs times
        copy A_host to A_device
        start = current GPU time
        for k = 2 to N, multiply k by 2 each step
            for j = k / 2 down to 1, divide j by 2 each step
                launch CUDA_BitonicKernel on (blocks, threadsPerBlock) with (A_device, j, k, N)
                synchronize device
        end = current GPU time
        totalTime = totalTime + (end - start)
    averageTime = totalTime / runs
    copy A_device back to A_host if needed
    free A_device
    return averageTime

Algorithm RunExperiment_GPU()
    runs  = 10
    sizes = [2^10, 2^15, 2^20]
    print "CUDA Bitonic Sort Results (average time)"
    for each N in sizes
        create array A_host of size N
        Random(A_host, N)
        avg = GPU_BitonicSort(A_host, N, runs)
        print N, avg

Algorithm Main_GPU()
    RunExperiment_GPU()
```

4.2 Performance

In the CUDA version of the Bitonic Sort algorithm, the array elements are divided among the GPU threads. Each thread is responsible for comparing one index i with its partner (i XOR j). These comparisons happen at the same time because the GPU can run multiple threads in parallel. We used 512 threads per block, and the number of blocks was computed as ceil(n / 512) so that every element is handled by a thread.

Bitonic Sort contains two nested loops (k and j), which together form a sorting network of depth $O(\log^2 n)$. Although the total amount of work of the algorithm is $O(n \log^2 n)$, the effective parallel runtime on the GPU can be approximated as $O((n / T) \log^2 n)$, where n is the input size and T is the number of active parallel threads. In practice, the actual performance depends not only on the input size, but also on GPU hardware characteristics such as memory bandwidth, warp scheduling, and the number of CUDA cores.

We ran the CUDA program on a NVIDIA Tesla T4 GPU (2560 CUDA cores, 16 GB memory) using Google Colab. For each input size, we executed the program 10 times and reported the average kernel execution time using CUDA events. The results are:

```
Array size: 1024
  Run 1: 42.2603 ms
  Run 2: 0.04768 ms
  Run 3: 0.050976 ms
  Run 4: 0.054112 ms
  Run 5: 0.062208 ms
  Run 6: 0.071136 ms
  Run 7: 0.05952 ms
  Run 8: 0.061056 ms
  Run 9: 0.061536 ms
  Run 10: 0.06496 ms
  Average time: 4.27935 ms
```

```
Array size: 32768
  Run 1: 0.132832 ms
  Run 2: 0.158272 ms
  Run 3: 0.151808 ms
  Run 4: 0.123008 ms
  Run 5: 0.136064 ms
  Run 6: 0.128 ms
  Run 7: 0.14912 ms
  Run 8: 0.145856 ms
  Run 9: 0.136352 ms
  Run 10: 0.150784 ms
  Average time: 0.14121 ms
```

```
Array size: 1048576
  Run 1: 0.16704 ms
  Run 2: 0.157824 ms
  Run 3: 0.157536 ms
  Run 4: 0.156928 ms
  Run 5: 0.156416 ms
  Run 6: 0.156352 ms
  Run 7: 0.156544 ms
  Run 8: 0.1568 ms
  Run 9: 0.15648 ms
  Run 10: 0.164704 ms
  Average time: 0.158662 ms
```

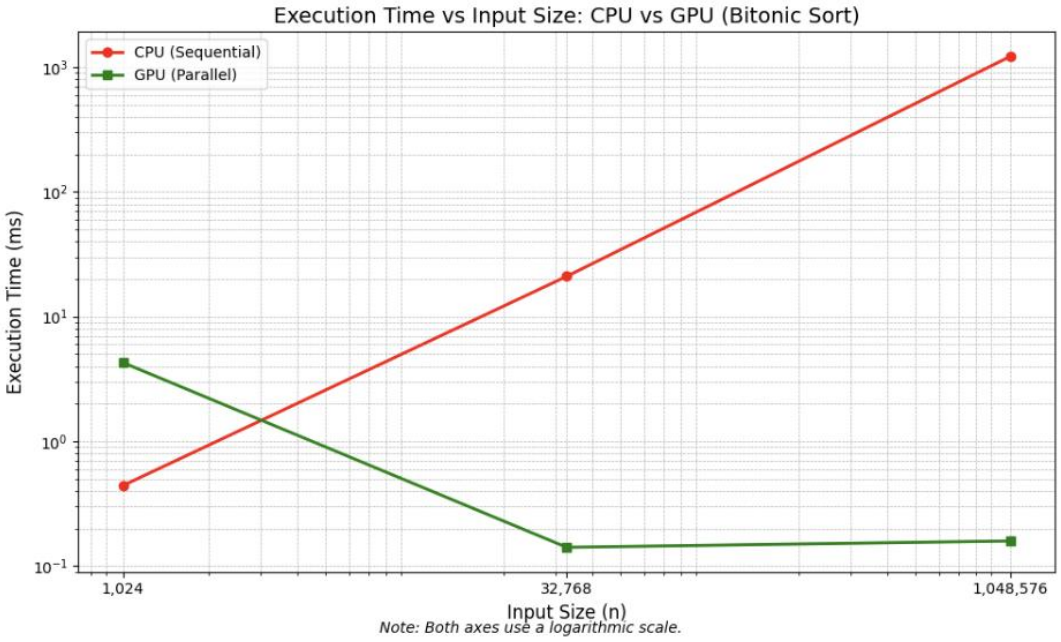
These results show that small inputs (like 1024 elements) are affected by kernel launch overhead, so they take longer compared to midsize inputs. For larger inputs (such as 1 million elements), the GPU parallelism becomes more effective, and the runtime remains very small, demonstrating the benefit of using CUDA for Bitonic Sort.

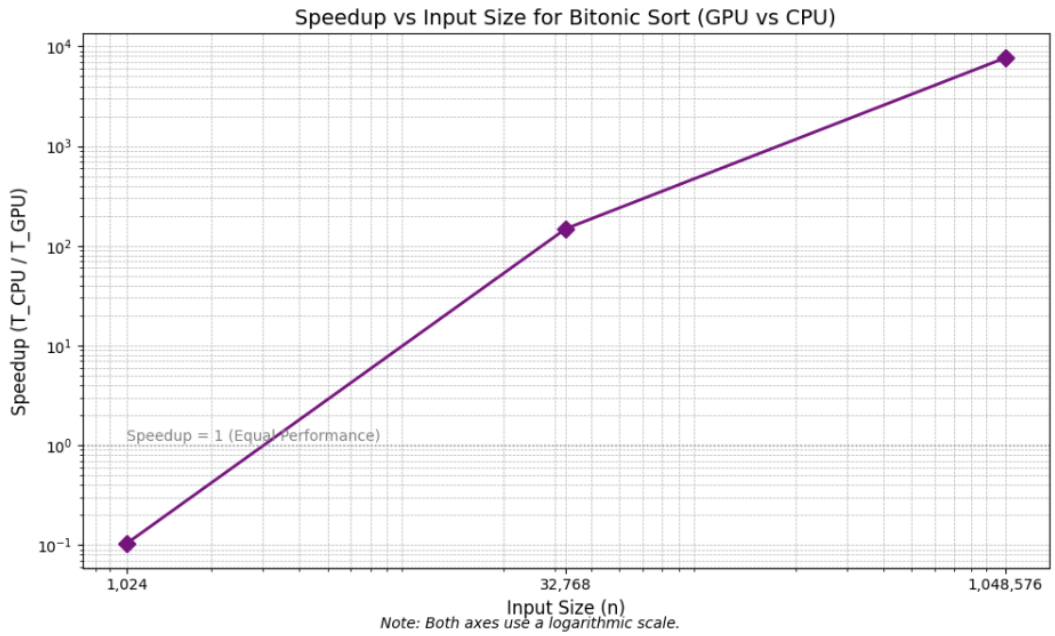
5 Results and Discussion

Execution Time and Speedup for Sequential and CUDA Bitonic Sort

Array size	Sequential time (ms)	Cuda time (ms)	Speed up	Interpretation
1024	0.441	4.27935	≈ 0.103	approximately ten times slower
32768	21.009	0.14121	≈ 148.8	Approximately 149 times faster
1048576	1220.847	0.158662	≈ 7694	Approximately 7700 times faster

Performance Comparison between Sequential and CUDA Bitonic Sort.






```
n =    1,024 → Speedup =      0.1x
n =   32,768 → Speedup =    148.8x
n = 1,048,576 → Speedup =   7694.6x
```

In this section we aim to compare the sequential code and CUDA Bitonic Sort performance, where we compared the average run time of 10 runs for each of the sequential and CUDA codes. As shown in the previous table and figures, we can notice that when the input size was small (e.g., 1024), the CUDA Bitonic Sort code was worse than the sequential one. It was 10 times slower. This can mainly be interpreted by the overhead caused by kernel launch.

But when we increased the input size to medium size (e.g., 32768), the CUDA Bitonic Sort code provided a big difference in run time compared to the sequential version, being 149 times faster. The impressive point is that when the array size increased to a really large number (e.g., 1048576), the speedup was even better than the previous case, where the CUDA Bitonic Sort code was 7700 times faster!

This indicates the following: the sequential Bitonic Sort code is better for small size data because of the overhead caused by kernel launch, but as the data becomes larger, the effectiveness of CUDA Bitonic Sort becomes much better.

Even though the CUDA Bitonic Sort code shows impressive performance and speedup as the data becomes larger, there still exist some limitations of Bitonic Sort in parallel, such as the previously mentioned point about small data size. Also, GPU memory imposes a limit on how large the data can be. Finally, even with the big improvement in run time, the algorithm remains not stable in both sequential and parallel Bitonic Sort.

These results clearly show that GPU parallelism is highly beneficial for Bitonic Sort with large data, while sequential execution may be better for small inputs or educational purposes.

6 Conclusion and Future Work

In this project, we implemented and compared Sequential and CUDA Bitonic Sort. Our experiments show the high effectiveness of CUDA Bitonic Sort for medium and large datasets. The CUDA Bitonic Sort showed a high speedup compared to the sequential version, up to 7700 times faster for 1 million inputs! The experiments also show that as the data size increases, the effectiveness of using the parallel version becomes more obvious, which highlights the advantage of GPU parallelism for Bitonic Sort on large datasets.

Even with this high effectiveness, there are still some limitations of CUDA Bitonic Sort. For example, the speedup on small datasets were negative because of kernel launch overhead, where the sequential version is still better. Another limitation is the size of GPU memory which limit the maximum data size. Also, the algorithm remains unstable in both sequential and CUDA versions.

For future work we suggest:

1. Executing similar experiments on other sort algorithms (e.g. Quik sort) to compare between the sequential and parallel versions.

2. Repeating the experience on larger arrays to check if the speed up still increase as data size increases.

3. Comparing results with CPU-based multi-threaded implementations to study parallel performance on modern multi-core processor

References

1- Tanasic, I., Vilanova, L., Jordà, M., Cabezas, J., Gelado, I., Navarro, N., & Hwu, W.-m. "Comparison based sorting for systems with multiple GPUs," *GPGPU-6: Proc. of the 6th Workshop on General Purpose Processing Using GPUs*, pp. 1–11, 2013. doi:10.1145/2458523.2458524.

2- L. Cui, "Bitonic Sorting Network Optimization for Parallel Architectures," arXiv:1506.01446 [cs.DC], 2015. doi: 10.48550/arXiv.1506.01446.

Appendix A

The Sequential Bitonic Sort Results (average of 10 runs):

```
Size  Time (s)
1024    0.000441
32768   0.021009
1048576 1.220847
```

The Sequential Bitonic Sort results (sorting 8 integer):

```
Sequential Bitonic Sort
1) Sort 8 numbers (manual input)
2) Run performance test (2^10, 2^15, 2^20)
0) Exit
Select an option: 1
Enter 8 integers:
10 8 2 0 8 11 1 3
Before sorting:
Array: 10 8 2 0 8 11 1 3
After sorting:
Array: 0 1 2 3 8 8 10 11

Sequential Bitonic Sort
1) Sort 8 numbers (manual input)
2) Run performance test (2^10, 2^15, 2^20)
0) Exit
Select an option: 0
Exiting program...
```

The CUDA Bitonic Sort results (average of 10 runs):

```
Array size: 1024
  Run 1: 42.2603 ms
  Run 2: 0.04768 ms
  Run 3: 0.050976 ms
  Run 4: 0.054112 ms
  Run 5: 0.062208 ms
  Run 6: 0.071136 ms
  Run 7: 0.05952 ms
  Run 8: 0.061056 ms
  Run 9: 0.061536 ms
  Run 10: 0.06496 ms
  Average time: 4.27935 ms
```

```
Array size: 32768
  Run 1: 0.132832 ms
  Run 2: 0.158272 ms
  Run 3: 0.151808 ms
  Run 4: 0.123008 ms
  Run 5: 0.136064 ms
  Run 6: 0.128 ms
  Run 7: 0.14912 ms
  Run 8: 0.145856 ms
  Run 9: 0.136352 ms
  Run 10: 0.150784 ms
  Average time: 0.14121 ms
```

```
Array size: 1048576
  Run 1: 0.16704 ms
  Run 2: 0.157824 ms
  Run 3: 0.157536 ms
  Run 4: 0.156928 ms
  Run 5: 0.156416 ms
  Run 6: 0.156352 ms
  Run 7: 0.156544 ms
  Run 8: 0.1568 ms
  Run 9: 0.15648 ms
  Run 10: 0.164704 ms
  Average time: 0.158662 ms
```

The CUDA Bitonic Sort results (sorting 8 integer):

```
Demo: Enter 8 integers:
7 4 8 1 0 2 3 10
Sorted array:
0 1 2 3 4 7 8 10
```