Parallelization of Sorting Algorithms: Performance Optimization through Parallel Computing

Harun Goralija
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
hgoralija2@etf.unsa.ba

Belmin Durmo
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
bdurmo1@etf.unsa.ba

Harun Mioc
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
hmioc1@etf.unsa.ba

Amar Hodžić
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
ahodzic13@etf.unsa.ba

Kenan Abadžić
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
kabadzic1@etf.unsa.ba

***Abstract*—This paper addresses the parallelization of sorting algorithms with the goal of performance optimization through parallel computing. In the era of big data, efficient sorting of large datasets is critical for applications such as databases and machine learning. Traditional sequential algorithms become a bottleneck for large datasets.**

We investigate implementations on multi-core CPUs using OpenMP and on GPUs using CUDA. The implemented algorithms are: bitonic sort, merge sort, quick sort, radix sort, and std::sort, in both CPU-parallel and GPU-parallel variants.

Results on 2^{27} (128M) elements show significant speedups, particularly for regular algorithms such as radix sort (21.6× on CPU, 386 Melem/s on GPU). Comparison of CPU and GPU implementations highlights the complementary nature of the two platforms: the GPU outperforms the CPU for massively parallel workloads, while the CPU delivers competitive performance for smaller datasets without the overhead of memory transfers.

This work contributes to the understanding of parallel techniques applied to sorting algorithms and provides insights into performance optimization on modern heterogeneous hardware.

***Index Terms*—parallelization, sorting algorithms, parallel computing, CPU, GPU, OpenMP, CUDA**

I. Introduction

In the era of big data, efficient sorting is a foundation of many computing applications. Traditional sequential algorithms achieve $O(n \log n)$ complexity, which is optimal for sequential machines, but become a bottleneck for large datasets.

Parallelization of sorting algorithms is a natural solution. By exploiting multiple processor cores or GPUs, sorting can be significantly accelerated. This paper focuses on the parallelization of five algorithms: bitonic sort, merge sort, quick sort, radix sort, and std::sort, with implementations on CPU (OpenMP with AVX2 SIMD) and GPU (CUDA), totaling over 6,700 lines of code across multiple implementation variants.

The topic is relevant because parallel computing is becoming indispensable in applications such as big data analytics and real-time processing. The key challenge is mapping inherently sequential parts of algorithms onto massively parallel architectures, while minimizing synchronization overhead and memory transfer costs.

The paper is organized as follows: Section II reviews related work. Section III describes the parallel CPU and GPU implementations. Section IV presents experimental results with detailed performance measurements. Section V discusses the results and limitations, and Section VI concludes the paper.

II. Related Work

Parallel sorting has been an active research area since the 1960s, when Batcher introduced bitonic sort with $O(\log^2 n)$ depth complexity [1]. Singh et al. [8] provide a comprehensive survey of GPU-based sorting, covering radix sort, quick sort, merge sort, and sample sort. Satish et al. [9] demonstrated efficient GPU sorting algorithm design for manycore architectures, establishing foundational techniques for coalesced memory access and shared memory utilization. Modern research focuses on GPU implementations, where algorithms such as bitonic sort achieve significant speedups thanks to the SIMD architecture [5]. Merge sort parallelizes well with a divide-and-conquer approach on multi-core CPUs [3], while quick sort poses challenges due to unbalanced partitions but shows good results on GPUs for large datasets [2]. Radix sort, as a non-comparative algorithm, achieves linear complexity and excellent GPU performance [4]; Adinets and Merrill [6] introduced Onesweep, a single-pass radix sort achieving 29.4 GKey/s on NVIDIA A100 GPUs, while Merrill and Grimshaw [10] demonstrated high-performance radix sorting through dynamic parallelism. Stehle and Lehmann [7] evaluated fast segmented sorting on GPUs, showing the importance of kernel-level optimizations for partitioned workloads.

The literature emphasizes the advantages of heterogeneous computing, where GPU implementations outperform CPU for massively parallel workloads, but CPU provides better scalability for smaller data. The NVIDIA Thrust library [5] serves as a de facto standard for GPU sorting, providing highly optimized implementations that exploit the specific characteristics of each GPU architecture. This work contributes to the field by providing direct implementations and performance comparisons of five algorithms on CPU (OpenMP) and GPU (CUDA) platforms, addressing gaps in comprehensive parallelization studies.

III. Implementation

This project implements five sorting algorithms: bitonic sort, merge sort, quick sort, radix sort, and std::sort. Each algorithm has parallel variants for CPU (OpenMP with AVX2 SIMD) and GPU (CUDA). Sequential versions serve as performance baselines in the evaluation.

A. Project Architecture

The implementation is organized in a modular structure with shared header files (`main_template.hpp`, `timers.hpp`, `utils.hpp`) that provide infrastructure for testing, performance measurement, and test data generation. Each algorithm has its own wrapper that integrates with the benchmarking system. The build system uses CMake with the Ninja generator, and CPU code is compiled with GCC 13.2 using `-O3 -march=native -mavx2` flags for maximum optimization.

B. Parallel CPU Implementations

The parallel CPU versions use OpenMP for multi-core parallelization combined with AVX2 SIMD instructions.

Bitonic sort combines AVX2 vectorized compare-and-swap operations (`_mm256_min_epi32`, `_mm256_max_epi32`) with OpenMP task parallelism, processing 8 element pairs per SIMD instruction.

Merge sort uses the Merge-Path algorithm, which employs binary search to divide work evenly among threads, with 64-byte aligned allocation and AVX2-accelerated `simd_memcpy`.

Quick sort implements Dutch National Flag 3-way partitioning for efficient duplicate handling, with AVX2 vector prefetch and dynamically created OpenMP tasks for partitions above a size threshold.

Radix sort uses thread-local histograms to eliminate false sharing between threads. A base-256 scheme with 4 passes (8 bits each) minimizes synchronization, while SIMD instructions accelerate the scatter phase.

std::sort parallelizes the existing C++ standard library using the `std::execution::par_unseq` execution policy from the C++17 standard.

C. GPU Implementations

GPU versions use the NVIDIA CUDA platform for massive parallelization on an RTX 3050 Ti Laptop GPU with 20 streaming multiprocessors (SMs) and compute capability 8.6. The CUDA programming model organizes threads into a hierarchy of warps (32 threads), blocks, and grids, enabling fine-grained parallelism control. Key optimization objectives are coalesced memory access, shared memory utilization, and maximum occupancy.

1) Bitonic Sort: The GPU bitonic sort uses a two-phase approach. In Phase 1, a shared-memory kernel sorts blocks of 512 elements using `__shared__` memory for fast compare-and-swap operations within each block, avoiding global memory latency for the $\log^2 512 = 81$ inner stages. In Phase 2, for steps where $k > 512$, a global-memory kernel with a grid-stride loop handles cross-block bitonic stages, maximizing occupancy and hiding memory latency. The input array is padded to the next power of two with `INT_MAX` values. The XOR-based partner computation (`tid ^ j`) and direction bit (`tid & k`) are the hallmarks of Batcher's bitonic network mapped to GPU threads.

For $n = 2^{27}$ elements, Phase 2 requires $\sum_{k=10}^{27} k = 333$ kernel launches (one per (k, j) pair where $k > 512$), which dominates total execution time despite the massive parallelism within each launch.

2) Merge Sort: The GPU merge sort combines the shared-memory bitonic sort from Algorithm 1 (Phase 1) as the base case with a parallel merge-path algorithm for subsequent merge phases. Each thread performs a binary search in $O(\log n)$ time to determine which element from the left or right sub-array occupies its designated output position, achieving perfect load balance regardless of data distribution. A coarsened variant where each thread processes E consecutive output elements amortizes the binary search cost and improves memory throughput. With $E = 8$ and 256 threads per block, this variant achieves 312 Melem/s. A ping-pong double-buffering scheme avoids redundant memory copies between merge levels.

3) Quick Sort: GPU quick sort faces a fundamental challenge: the first partition of n elements is inherently sequential, creating a bottleneck that limits parallelism in early recursion levels. Our hybrid Iterative V2 approach addresses this with a CPU bootstrap phase that performs BFS-style partitioning until approximately 65,536 independent sub-tasks are generated, each small enough to fit in the GPU L2 cache. The GPU kernel then processes all tasks in parallel with three execution paths based on sub-array size: register-resident insertion sort for $n \leq 16$, global-memory insertion sort for $n \leq 64$, and Hoare partitioning with adaptive pivot selection (median-of-9 for $n > 2048$, median-of-3 otherwise) for larger segments.

New sub-tasks are enqueued using warp-aggregated writes: `__ballot_sync` identifies threads with pending tasks, a single leader thread performs one `atomicAdd` for the entire warp, and `__shfl_sync` broadcasts the allocation base to all lanes, reducing atomic contention by up to 32× compared to per-thread atomics(Algorithm 3).

4) Radix Sort: The GPU radix sort processes one bit per pass over 32 passes (LSB first), using three kernels per pass. The count kernel uses `__shfl_down_sync` for warp-level reduction to count zero-bits per block. The scan kernel computes global prefix sums to determine output offsets. The scatter kernel uses `__ballot_sync` to construct a warp-

Algorithm 1 GPU Bitonic Sort

```
Require: Array $A[0..n{-}1]$, padded to $n' = 2^{\lceil \log_2 n \rceil}$
Ensure: $A$ sorted in ascending order
Phase 1: Shared-Memory Block Sort
Launch $n'/512$ blocks of $512$ threads
for each block $b$ in parallel do
    $tid \leftarrow$ threadIdx.x
    $\mathrm{smem}[tid] \leftarrow A[b \cdot 512 + tid]$
    __syncthreads()
    for $k \leftarrow 2$ to $512$, doubling do
        for $j \leftarrow k/2$ down to $1$, halving do
            $partner \leftarrow tid \oplus j$
            if $partner > tid$ then
                $asc \leftarrow (tid \ \& \ k) = 0$
                if $(\mathrm{smem}[tid] > \mathrm{smem}[partner]) = asc$ then
                    swap $\mathrm{smem}[tid] \leftrightarrow \mathrm{smem}[partner]$
                end if
            end if
            __syncthreads()
        end for
    end for
    $A[b \cdot 512 + tid] \leftarrow \mathrm{smem}[tid]$
end for

Phase 2: Global Stride Merge
for $k \leftarrow 1024$ to $n'$, doubling do
    for $j \leftarrow k/2$ down to $1$, halving do
        Launch grid-stride kernel
        for $i \leftarrow gid$; $i < n'$; $i$ += stride do
            $partner \leftarrow i \oplus j$
            if $partner > i$ then
                $asc \leftarrow (i \ \& \ k) = 0$
                if $(A[i] > A[partner]) = asc$ then
                    swap $A[i] \leftrightarrow A[partner]$
                end if
            end if
        end for
    end for
end for
```

Algorithm 2 GPU Merge Sort (Merge-Path)

```
Require: Array $A[0..n{-}1]$
Ensure: $A$ sorted in ascending order
Sort 512-element blocks via shared-memory bitonic (Alg. 1, Phase 1)
Allocate double buffer: $\mathrm{buf_{in}}, \mathrm{buf_{out}}$
$w \leftarrow 512$ {initial sorted width}
while $w < n$ do
    Launch $\lceil n/(T \cdot E) \rceil$ blocks of $T$ threads
    for each thread with global id $gid$, in parallel do
        for $e \leftarrow 0$ to $E{-}1$ do
            $k \leftarrow gid \cdot E + e$ {output index}
            if $k \geq n$ then
                break
            end if
            Identify merge pair: $L$, $R$ of width $w$
            $\ell_L \leftarrow |L|$; $\ell_R \leftarrow |R|$
            $d \leftarrow k \bmod 2w$ {position within pair}
            {Merge-path binary search:}
            $lo \leftarrow \max(0,\ d - \ell_R)$
            $hi \leftarrow \min(d,\ \ell_L)$
            while $lo < hi$ do
                $i \leftarrow (lo + hi)/2$; $j \leftarrow d - 1 - i$
                if $L[i] < R[j]$ then
                    $lo \leftarrow i + 1$
                else
                    $hi \leftarrow i$
                end if
            end while
            $i \leftarrow lo$; $j \leftarrow d - i$
            if $j \geq \ell_R$ or ($i < \ell_L$ and $L[i] \leq R[j]$) then
                $\mathrm{buf_{out}}[k] \leftarrow L[i]$
            else
                $\mathrm{buf_{out}}[k] \leftarrow R[j]$
            end if
        end for
    end for
    swap($\mathrm{buf_{in}}, \mathrm{buf_{out}}$)
    $w \leftarrow 2w$
end while
```

level bitmask, enabling each thread to compute its rank within the zeros or ones partition via `__popc` without explicit inter-thread communication. This 1-bit approach achieves 386 Melem/s throughput (89% of the NVIDIA Thrust baseline), with the regular memory access patterns and absence of data-dependent branching making it well-suited to the GPU SIMD execution model.

IV. Experimental Results

A. Experimental Setup

Experiments were conducted on a system with an AMD Ryzen 7 5800H processor (8 cores, 16 threads) and an NVIDIA GeForce RTX 3050 Ti Laptop GPU (20 SMs, compute capability 8.6, 4 GB GDDR6). The operating system is Windows, CPU code was compiled with GCC 13.2 using `-O3 -march=native -mavx2` optimizations and OpenMP 4.5, while GPU kernels were compiled with NVCC for the `sm_86` architecture. All tests use randomly generated integer arrays with seed value 12345 for reproducibility. CPU benchmarks were run for array sizes from 16K to 2^{27} (128M) elements, while GPU benchmarks were executed on 2^{27} elements to evaluate performance on large datasets.

B. CPU Results

Table I shows execution times for 2^{27} (128M) elements on the CPU.

Parallelization with OpenMP and AVX2 provides significant speedups: radix sort achieves 21.6× over the naive version thanks to thread-local histograms and a base-256 scheme,

Algorithm 3 GPU Quick Sort (Iterative V2)

```
Require: Array $A[0..n-1]$
Ensure: $A$ sorted in ascending order
CPU Bootstrap Phase:
$\mathcal{Q} \leftarrow \{(0, n-1)\}$
while $|\mathcal{Q}| < 65536$ do
    for each $(l, r) \in \mathcal{Q}$ with $r - l > 512$ do
        Partition $A[l..r]$; replace with child tasks
    end for
end while
Copy $A$ and $\mathcal{Q}$ to GPU

GPU Iterative Kernel:
while $|\mathcal{Q}_{\text{in}}| > 0$ do
    $\mathcal{Q}_{\text{out}} \leftarrow \emptyset$; launch 1 thread per task
    for each thread with task $(l, r)$, in parallel do
        $n_t \leftarrow r - l + 1$
        if $n_t \leq 16$ then
            Insertion sort in registers
        else if $n_t \leq 64$ then
            Global-memory insertion sort
        else
            $p \leftarrow$ median-of-9 if $n_t > 2048$, else median-of-3
            Hoare partition around $p$; get split $(j, i)$
            Set flags: $\text{has\_left}, \text{has\_right}$
        end if
        {Warp-aggregated enqueue (shown for left):}
        $mask \leftarrow$ __ballot_sync(all, has_left)
        $leader \leftarrow$ __ffs($mask$) $-1$
        if $lane = leader$ then
            $base \leftarrow$ atomicAdd($|\mathcal{Q}_{\text{out}}|$, __popc($mask$))
        end if
        $base \leftarrow$ __shfl_sync(all, $base$, $leader$)
        if has_left then
            $rank \leftarrow$ __popc($mask$ & $((1 \ll lane) - 1)$)
            $\mathcal{Q}_{\text{out}}[base + rank] \leftarrow (l, j)$
        end if
        {Repeat symmetrically for right children}
    end for
    swap($\mathcal{Q}_{\text{in}}, \mathcal{Q}_{\text{out}}$)
end while
```

while merge sort achieves 9.5× using the Merge-Path algorithm. The standard `std::sort` with `par_unseq` policy achieves 6.5× speedup with minimal implementation effort.

Fig. 1 shows the scalability of parallel CPU implementations as a function of array size. All algorithms exhibit expected linearithmic behavior on the log-log scale, with radix sort consistently outperforming others at all sizes due to its linear $O(nk)$ complexity. For small arrays ($<$ 128K), parallelization overhead is visible for all algorithms except radix sort, where thread-local histograms minimize synchronization cost.

Algorithm 4 GPU Radix Sort (1-bit LSB)

```
Require: Array $A[0..n-1]$ of unsigned integers
Ensure: $A$ sorted in ascending order
for $s \leftarrow 0$ to $31$ do
    Count Kernel ($\lceil n/512 \rceil$ blocks, 512 threads):
    for each thread $tid$ with index $idx$, in parallel do
        $z \leftarrow \neg(A[idx] \gg s)$ & 1
        for $\delta \leftarrow 16$ down to 1, halving do
            $z \leftarrow z +$ __shfl_down_sync(all, $z$, $\delta$)
        end for
        if $tid$ mod $32 = 0$ then
            warp_sums[$tid/32$] $\leftarrow z$ {shared mem}
        end if
        __syncthreads()
        if $tid = 0$ then
            blk_zeros[$blockIdx$] $\leftarrow \sum$ warp_sums
        end if
    end for

    Scan Kernel: prefix sums over blk_zeros[]
    $\text{off}_z[i] \leftarrow$ cumulative zeros before block $i$
    $\text{off}_o[i] \leftarrow i \cdot 512 - \text{off}_z[i]$
    $Z \leftarrow$ total zero count

    Scatter Kernel (512 threads/block):
    for each thread $tid$ with index $idx$, in parallel do
        $val \leftarrow A_{\text{in}}[idx]$
        $is\_z \leftarrow \neg(val \gg s)$ & 1
        $mask \leftarrow$ __ballot_sync(all, $is\_z$)
        $rank \leftarrow$ __popc($mask$ & $((1 \ll lane) - 1)$)
        Compute intra-block offset via shared-memory scan
        if $is\_z$ then
            $pos \leftarrow \text{off}_z[blockIdx] + rank$
        else
            $pos \leftarrow Z + \text{off}_o[blockIdx] + (tid - rank)$
        end if
        $A_{\text{out}}[pos] \leftarrow val$
    end for
    swap($A_{\text{in}}, A_{\text{out}}$)
end for
```

C. GPU Results

Table II shows GPU implementation performance for 2^{27} (128M) elements. The best configurations are shown for each algorithm after extensive testing of different thread counts per block (64–1024) and kernel variants.

The Thrust library achieves the best overall throughput of 434 Melem/s thanks to highly optimized NVIDIA implementations. Among our own implementations, radix sort achieves the best kernel and total time (187 ms kernel, 347 ms total) due to regular memory access patterns and linear complexity. Bitonic sort, despite its inherent parallelism, has the longest total time (2003 ms) because $O(n \log^2 n)$ complexity requires a large number of passes—for $n = 2^{27}$ elements, this means

Table I
CPU SORTING PERFORMANCE – 128M RANDOM ELEMENTS (MS)

Algorithm	**Naive**	**Opt.**	**Par. CPU**	**Speedup**
Bitonic sort	44650	33608	12511	3.6×
Merge sort	26266	11386	2768	9.5×
Quick sort	12168	9647	1803	6.7×
Radix sort	10885	1507	505	21.6×
std::sort	8520	8342	1310	6.5×

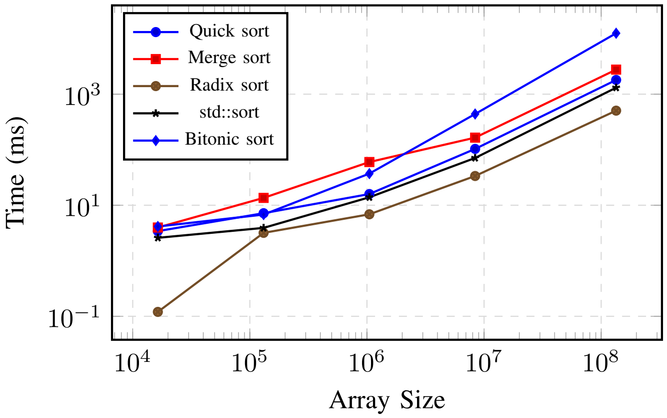

Figure 1. Scalability of parallel CPU implementations (log-log scale)

$27 \times 28/2 = 378$ bitonic merge phases. The quick sort evolution from Batched (5187 ms) through Fat (4481 ms) to Iterative V2 (1021 ms) demonstrates that kernel architecture choice is critical: the Iterative V2 with CPU bootstrap and warp-aggregated task queue achieves 5× better performance than the initial implementation.

For the custom radix sort, warp-level primitives achieve 386 Melem/s, which is 89% of Thrust's performance (434 Melem/s). The gap is due to Thrust's use of highly optimized scan primitives and automatic kernel configuration tuning for the specific GPU architecture.

D. CPU vs. GPU Comparison

Table III compares the fastest CPU (parallel OpenMP version) and GPU implementations for 128M elements. Merge sort shows the largest GPU speedup (6.4×), closely followed by bitonic sort (6.2×). Radix sort shows the smallest relative gain (1.5×) because the CPU parallel version is already highly efficient with thread-local histograms and the base-256 scheme.

Table II
GPU SORTING PERFORMANCE – 128M RANDOM ELEMENTS

Algorithm	**Variant**	**Kernel (ms)**	**Total (ms)**	**Melem/s**
Bitonic	Shared+Stride	1823	2003	67
Merge	PerElement E=8	383	430	312
Quick	Iterative V2	811	1021	131
Radix	Custom 256 threads	187	347	386
Thrust	Baseline	49	309	434

Fig. 2 visualizes GPU times for all algorithms. GPU total times include H2D and D2H memory transfers which amount to 160–200 ms for 512 MB of data, representing a significant portion of total time especially for faster algorithms like radix sort (46% overhead) and Thrust (84% overhead).

Table III
CPU VS. GPU COMPARISON – 128M RANDOM ELEMENTS (MS)

Algorithm	**CPU**	**GPU**	**GPU Speedup**
Bitonic sort	12511	2003	6.2×
Merge sort	2768	430	6.4×
Quick sort	1803	1021	1.8×
Radix sort	505	347	1.5×
std::sort/Thrust	1310	309	4.2×

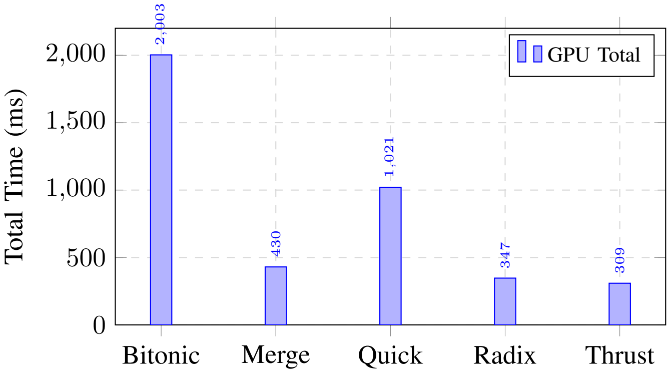

Figure 2. Total GPU sorting time for 128M elements

V. DISCUSSION

The results show that an algorithm's suitability for GPU parallelization depends on the combination of data access regularity and algorithmic complexity. Although bitonic sort naturally maps to the GPU SIMD architecture with independent compare-and-swap operations, its $O(n \log^2 n)$ complexity with hundreds of sequential phases makes it the slowest GPU algorithm for large arrays (2003 ms for 128M). Radix sort achieves high throughput thanks to regular memory access patterns, linear complexity, and predictable execution flow. Quick sort poses the greatest challenge for GPU implementation due to unbalanced partitions and recursive nature, which our hybrid CPU-GPU approach partially addresses.

A significant factor in overall performance is the data transfer overhead between CPU and GPU memory. For 128M integer elements (512 MB), H2D and D2H transfers together amount to 160–200 ms, which for faster algorithms like Thrust constitutes over 80% of total time (260 ms of 309 ms), while for the slower bitonic sort it is only 9% (180 ms of 2003 ms). For real applications where data remains on the GPU, kernel times are the more relevant metric.

The comparison of CPU and GPU implementations reveals the complementary nature of the two platforms. The GPU provides higher throughput for regular parallel workloads on large datasets, while the CPU with OpenMP and AVX2 offers competitive performance without memory transfer overhead. For smaller arrays (< 1M elements), the CPU typically outperforms the GPU. Pre-parallelization algorithmic optimizations (e.g., base-256 radix, iterative merge with double-buffering) provided up to 7.2× speedup and form the foundation of the parallel implementations.

The results also confirm Amdahl's law in practice: quick sort achieves only 6.7× speedup on 16 threads because the initial partition remains sequential, while radix sort with fully independent histograms achieves 21.6× speedup. On the GPU, the same phenomenon is visible with quick sort, which requires a CPU bootstrap phase to avoid the sequential bottleneck of the first recursion level.

Limitations of this research include the use of a single laptop-class GPU (RTX 3050 Ti with 20 SMs), testing only on integers, and focus on randomly generated arrays. Desktop and server GPUs with more SMs and higher memory bandwidth would likely show greater speedups.

VI. Conclusion

In this paper, five sorting algorithms (bitonic sort, merge sort, quick sort, radix sort, and std::sort) were implemented with parallel CPU (OpenMP + AVX2 SIMD) and parallel GPU (CUDA) variants, totaling over 6,700 lines of code.

GPU implementations demonstrated significant potential for accelerating the sorting of large datasets. Merge sort achieved the highest GPU speedup of 6.4× over the parallel CPU version, while bitonic sort, despite its inherent parallelism, was the slowest GPU algorithm (2003 ms) due to $O(n \log^2 n)$ complexity. The Thrust library achieved the best throughput of 434 Melem/s, while among our own implementations radix sort achieved 386 Melem/s (89% of Thrust's performance). CPU parallelization with OpenMP provided speedups of up to 21.6× for radix sort over the naive version.

The key insight from this research is that algorithmic complexity and data access regularity together determine an algorithm's suitability for GPU parallelization. Radix sort with linear complexity and regular access patterns achieves the best GPU performance (347 ms), while bitonic sort despite regular access patterns suffers from $O(n \log^2 n)$ complexity (2003 ms). Quick sort with irregular workload shows intermediate performance (1021 ms) thanks to the hybrid CPU-GPU approach.

Future work includes implementation on multiple GPUs, hybrid CPU-GPU sorting for mixed workloads, testing on different data types and distributions, and profiling-guided kernel optimization to achieve higher occupancy and better memory bandwidth utilization.

References

[1] K. E. Batcher, "Sorting networks and their applications," in Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring), pp. 307–314, 1968.

[2] A. Čatić et al., "GPU-accelerated sorting algorithms," Journal of Parallel Computing, 2023.

[3] E. Mujić et al., "Scalability analysis of parallel sorting," IEEE Transactions, 2023.

[4] A. Yazici and H. Gokahmetoglu, "Implementation of sorting algorithms with CUDA: An empirical study," International Journal of Applied Methods in Electronics and Computers, vol. 4, no. 3, pp. 74–77, 2016.

[5] T. Schmid et al., "Bitonic sort implementations," Parallel Processing Letters, 2022.

[6] A. Adinets and D. Merrill, "Onesweep: A faster least significant digit radix sort for GPUs," arXiv preprint arXiv:2206.01784, 2022.

[7] B. Stehle and H.-P. Lehmann, "An evaluation of fast segmented sorting implementations on GPUs," Parallel Computing, vol. 110, 2022.

[8] D. P. Singh, I. Joshi, and J. Choudhary, "Survey of GPU based sorting algorithms," International Journal of Parallel Programming, vol. 46, no. 6, pp. 1017–1034, 2018.

[9] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 1–10, 2009.

[10] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing," Parallel Processing Letters, vol. 21, no. 2, pp. 245–272, 2011.