A GPU Bitonic Sort Implementation using CUDA

Fraidakis Ioannis

January 2025

Abstract

This report presents a detailed implementation and performance analysis of the Bitonic Sort algorithm accelerated using CUDA. Four distinct CUDA kernel implementations are developed and evaluated, each progressively addressing performance bottlenecks. Various optimizations, including shared memory usage and warp-level operations, are explored to improve efficiency. Experimental results, obtained by varying input sizes showcase the effectiveness of the proposed optimization strategies. Performance comparisons with Radix Sort using the CUB library are provided.

1 Introduction

Efficient sorting algorithms are crucial for a wide range of computational tasks. With the advent of Graphics Processing Units (GPUs) and their massively parallel architectures, there has been a growing interest in accelerating sorting algorithms to exploit their computational throughput. This report presents an implementation and performance analysis of Bitonic Sort on NVIDIA GPUs using CUDA (Compute Unified Device Architecture).

Bitonic Sort is a deterministic, comparison-based sorting algorithm characterized by a regular and predictable sequence of operations, making it ideal for parallel execution. Its structured computation pattern aligns naturally with the Single-Instruction, Multiple-Data (SIMD) execution model of modern GPUs, enabling thousands of threads to operate concurrently. However, realizing the full potential of GPUs requires careful optimization, including minimizing memory access latency and reducing synchronization overhead.

This report explores the GPU-based implementation of Bitonic Sort, sorting input arrays of size N (restricted to powers of two) using parallel CUDA kernels. We present three distinct implementations, each progressively optimizing performance by addressing specific computational bottlenecks:

- **Version 0 (V0):** A straightforward implementation. Each thread performs a single compare-and-swap operation. This version serves as a baseline, highlighting the significant overhead of excessive kernel launches and implicit global synchronization.
- **Version 1 (V1):** This version integrates the inner loop of the Bitonic Sort algorithm directly within the kernel, drastically reducing the number of kernel invocations and, consequently, global synchronization points.
- **Version 2 (V2):** Building upon the kernel fusion approach of V1, this version introduces shared memory within the kernel to minimize global memory accesses.

- **Version 3 (V3):** This last version further reduces memory latency by using additional intra-warp exchanges performed directly on registers, thereby minimizing shared memory accesses. It also employs pinned memory to accelerate host–device data transfers.

1.1 Theoretical background

1.1.1 Bitonic sequence

A **bitonic sequence** is a sequence of numbers $a_0, a_1, ..., a_{n-1}$ such that:

1. There exists an index k $(0 \leq k \leq n-1)$ such that $a_0 \leq a_1 \leq ... \leq a_k$ and $a_k \geq a_{k+1} \geq ... \geq a_{n-1}$, or
2. The sequence is a cyclic rotation of a sequence satisfying condition 1.

For example:

- Example 1: {2, 9, 14, 16, 20, 23, **32**, 25, 20, 15, 10, 5} (Peak at 32)
- Example 2 (Cyclic Rotation): {18, 23, **27**, 20, 10, 3, 8, 12, 16} (Peak at 27, rotated)

1.1.2 Bitonic Sort

Bitonic Sort is a comparison-based sorting algorithm that first transforms an input sequence into a bitonic sequence and then merges it into a fully sorted sequence. It consists of two main phases:

Phase 1 — Bitonic Sequence Construction:

Given an input sequence of size $n = 2^k$ (k is a positive integer), this phase recursively builds a bitonic sequence:

- Initially, treat each pair of elements as a 2-element subsequence and sort them in alternating ascending and descending order.
- Repeatedly merge sorted subsequences into larger bitonic sequences by doubling their size. Each merge step sorts the first half of the subsequence in ascending order and the second half in descending order (using the Bitonic Merge process described in Phase 2).

Phase 2 — Bitonic Merge Sorting:

A bitonic sequence of size n is sorted in $\log_2 n$ stages. In each stage i $(0 \leq i < \log_2 n)$, elements at a distance of 2^i are compared and potentially swapped. The comparisons ensure that the resulting subsequences remain bitonic, with smaller elements in the first half and larger elements in the second half. This process continues until adjacent elements are compared, resulting in a fully sorted sequence.

Final ouput:

The output is a sorted sequence (ascending by default; descending can be achieved by reversing the comparison directions during the merge phase).

1.2 Hypercube-Based Partner Selection in Parallel Bitonic Sort

Parallel Bitonic Sort leverages the hypercube topology for efficient partner selection. In a p-dimensional hypercube, each node (representing a thread) has p neighbors, determined by flipping a single bit in its binary representation. This concept can be adapted for parallel Bitonic Sort, where comparisons and swaps occur between elements identified using bitwise operations. A thread with ID *tid* calculates its comparison partner's index:

$$partner = tid \oplus distance \tag{1}$$

The *distance* parameter, representing the separation between compared elements, changes dynamically: Initially, it's $2^{(dimension-1)}$, and it halves in each subsequent iteration, progressing from large-scale exchanges to fine-grained swaps.

Exchange Pattern Visualization

The following figures illustrate 2-dimensional and 3-dimensional hypercubes, showing the structure and the exchange distances between elements:

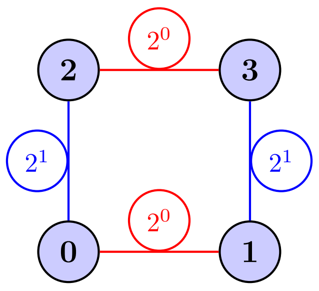

Figure 1: 2-Dimensional Hypercube

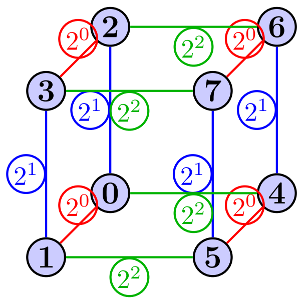

Figure 2: 3-Dimensional Hypercube

1.3 Comparative Algorithm Analysis

Bitonic Sort's serial time complexity is $O(n \log^2 n)$, which is higher than the $O(n \log n)$ complexity of algorithms like Merge Sort. However, its inherent parallelism allows for a significant reduction in time complexity on parallel architectures like GPUs. Ideally, with $O(n)$ threads, the parallel time complexity of Bitonic Sort is $O(\log^2 n)$[1]. This parallel execution has the potential to overcome the $O(n \log n)$ lower bound that applies to sequential sorting algorithms. While Bitonic Sort offers the possibility of near-linear speedup, practical factors, including memory bandwidth limitations, thread synchronization overhead, and kernel launch latency, can influence the actual performance achieved.

2 Version 0 - Basic Implementation

This initial CUDA implementation (V0) provides a straightforward parallelization of the standard Bitonic Sort algorithm described in §1.1.2.

2.1 Pseudocode for Bitonic Sort (V0)

Algorithm 1 GPU Bitonic Sort Version 0

```
1:  procedure BitonicSortV0($data, size$)
2:      Allocate device memory $devData$                          ▷ cudaMalloc
3:      $devData \leftarrow$ Copy $data$ from CPU                 ▷ cudaMemcpyHostToDevice
4:      threadsPerBlock $\leftarrow \min(size, maxThreadsPerBlock)$
5:      blocksPerGrid $\leftarrow \lceil size/$threadsPerBlock$\rceil$
6:      max_hypercube_dimension $\leftarrow \log_2($size$)$
7:      for $dimension \leftarrow 1$ to $max\_hypercube\_dimension$ do
8:          for $distance \leftarrow 2^{dimension-1}$ downto 1 do   ▷ Distance halving each iteration
9:              Launch compareSwapV0 kernel with:
10:                 Grid: blocksPerGrid,
11:                 Block: threadsPerBlock,
12:                 Parameters: $devData$, $size$, $dimension$, $distance$
13:         end for
14:     end for
15:     Copy $devData$ back to $data$                             ▷ cudaMemcpyDeviceToHost
16:     Free device memory $devData$                              ▷ cudaFree
17: end procedure
```

Algorithmic Breakdown of `BitonicSortV0`

1. **Outer Loop (Bitonic Subsequence Growth):**
   - Iterates over dimensions $d = 1, 2, ..., \log_2(N)$ of the hypercube.
   - Processes subsequences of size 2^d at each stage.
   - Goal: Build bitonic sequences of doubling size.
   - Example: $d = 3 \Rightarrow$ handles 8-element subsequences.

2. **Inner Loop (Comparison Distance):**
   - For each stage d, distance starts at 2^{d-1} (maximum distance for this dimension)
   - Halves distance each step: $2^{d-1} \to 2^{d-2} \to ... \to 1$.
   - At each *distance*, performs compare-and-swap operations on elements separated by that distance.
   - The final step ($distance = 1$) sorts adjacent elements, completing the merge.

3. **Parallel Execution Pattern:**
   - Separate kernel launch for each (dimension, distance) pair.
   - Each thread handles one element pair independently.

2.2 Compare-Swap Kernel (V0)

Algorithm 2 Compare-Swap Kernel (V0)

```
procedure compareSwapV0($data, size, dimension, distance$)
    $tid \leftarrow$ Global thread ID                    ▷ blockIdx.x * blockDim.x + threadIdx.x
    if $tid \geq size$ then return                      ▷ Early exit for threads beyond array size
    end if
    $partner \leftarrow tid \oplus distance$             ▷ Bitwise XOR for partner index
    if $partner > tid$ and $partner < size$ then
        $isAscending \leftarrow$ getSortingDirection($tid, distance, dimension$)
        if ($data[tid] > data[partner]$) == $isAscending$ then      ▷ Out of order
            swap($data[tid], data[partner]$)
        end if
    end if
end procedure
```

Explanation of `compareSwapV0`

1. **Thread ID and Bounds Check:** Each thread calculates its global ID (*tid*) based on its block and thread indices. Threads with IDs exceeding the array size exit early to avoid out-of-bounds memory access.

2. **Partner Calculation:** The index of the comparison partner (*partner*) is calculated using a bitwise XOR operation ($tid \oplus distance$). This operation efficiently determines the partner element based on the current *distance* parameter (§1.2).

3. **No redundant swaps:** This is achieved by performing a swap if $partner > tid$. This ensures that each pair of elements is compared and swapped only once.

4. **Sorting Direction:** The *getSortingDirection* function determines whether the comparison should be in ascending or descending order based on the *tid* and the current *dimension*. This is essential for correctly merging the bitonic sequences.

2.3 Performance Considerations

While this initial implementation (`V0`) correctly implements the bitonic compare-and-swap operation, it suffers from several performance limitations that hinder its efficiency:

- **Excessive Kernel Launch Overhead:** Each comparison step requires a separate kernel launch, leading to excessive kernel invocation overhead.

- **Implicit Global Synchronization:** CUDA kernels cannot directly synchronize globally. The implicit synchronization between consecutive kernel launches introduces additional latency. Each kernel launch acts as a synchronization point, forcing the GPU to wait for all threads to complete before the next kernel can begin. This frequent synchronization further contributes to the overall execution time.

Subsequent versions of the Bitonic Sort implementation address these performance bottlenecks by reducing kernel launch overhead and global synchronization.

3 Version 1 - Kernel Fusion

Version 1 implements kernel fusion by integrating the inner loop of the Bitonic Sort algorithm directly within each kernel whenever possible. This allows each kernel to perform multiple compare-and-swap operations, reducing the number of kernel invocations.

3.1 Pseudocode for Bitonic Sort (V1)

Algorithm 3 GPU Bitonic Sort Version 1

```
1: procedure BitonicSortV1($data, size$)
2:     Allocate device memory $d\_data$ and copy $data$ to GPU
3:     Compute threadsPerBlock and blocksPerGrid    ▷ Based on device capabilities
4:     $max\_hypercube\_dimension \leftarrow \log_2(size)$
5:     $max\_intra\_block\_dimension \leftarrow \log_2(threadsPerBlock)$
6:     $max\_intra\_block\_distance \leftarrow \frac{max\_intra\_block\_dimension}{2}$
7:
8:     Launch intraBlockSort kernel    ▷ Initial block-level sort
9:
10:    for $dimension \leftarrow max\_intra\_block\_dimension + 1$ to $max\_hypercube\_dim$ do
11:        for $distance \leftarrow 2^{dimension-1}$ downto $max\_intra\_block\_distance$ do
12:            Launch compareSwapV0 kernel    ▷ Inter block merge
13:        end for
14:        Launch intraBlockMerge kernel    ▷ Block-level merge
15:    end for
16:    Copy $d\_data$ back to $data$ and free device memory
17: end procedure
```

Enhancements in BitonicSortV1

The only difference in `Version 1` over `Version 0` is the introduction of two new kernels:

1. **Initial Sort Kernel (`intraBlockSort`)** This kernel performs an initial Bitonic Sort within each thread block, sorting subsequences up to the size of a block (`threadsPerBlock`) in a single kernel call. Unlike `V0`, which launched a separate kernel for each `(dimension, distance)` pair, this approach significantly reduces kernel launch overhead. In addition, all comparisons happen within a block, so synchronization is handled using `__syncthreads()`, avoiding global synchronization.

2. **Intra-Block Merge Kernel (`intraBlockMerge`)** After the initial sorting step, subsequent merging is split into inter-block and intra-block phases. This kernel finalizes the bitonic merge within each block once the `distance` in the inner loop is smaller than or equal to `max_intra_block_distance`. Instead of launching new kernels, this phase leverages `__syncthreads()` for block-level synchronization, allowing all threads within a block to efficiently perform remaining compare-and-swaps.

3.2 Pseucode for Initial Intra Block Sorting Kernel

Algorithm 4 Initial Intra Block Sorting Kernel

```
procedure intraBlockSort($data, size, max\_intra\_block\_dimension$)
    $tid \leftarrow$ Global thread ID
    if $tid \geq size$ then return
    end if
    for $dimension \leftarrow 1$ to $max\_intra\_block\_dimension$ do
        $isAscending \leftarrow$ getSortingDirection($tid, dimension$)
        for $distance \leftarrow 2^{dimension-1}$ downto 1 do
            $partner \leftarrow tid \oplus distance$
            if $partner > tid$ and $partner < size$ then
                if $(data[tid] > data[partner]) == isAscending$ then
                    swap($data[tid], data[partner]$)
                end if
            end if
            __syncthreads()                    ▷ Block-level synchronization
        end for
    end for
end procedure
```

Explanation of `intraBlockSort`

1. **Structure:** Its structure is similar to the `compareSwapV0` kernel, but it incorporates the two nested loops of the Bitonic Sort algorithm within the kernel itself.

2. **Synchronization:** The `__syncthreads()` call ensures that all threads within the block are synchronized after each distance-based comparison step. This is essential for the correct execution of the intra-block Bitonic Sort.

The `intraBlockMerge` kernel also performs operations within each block, similar in structure to `intraBlockSort` and `compareSwapV0`. However, it integrates only the inner distance-based loop of the algorithm. This kernel is responsible for completing the bitonic merge after the inter-block merge has taken place.

3.3 Performance Considerations

- Fewer kernel launches, reducing overhead.

- Global synchronization is now only required for inter-block merging, while intra-block operations rely on `__syncthreads()` for efficient thread coordination.

These optimizations lay the foundation for Version 2, which introduces shared memory to further reduce global memory bottleneck and enhance performance.

4 Version 2 - Shared Memory

Version 2 builds upon V1 but replaces `intraBlockSort` and `intraBlockMerge` with `intraBlockSortShared` and `intraBlockMergeShared`, which leverage shared memory to reduce global memory accesses, a key performance bottleneck.

Pseudocode for Shared Memory Intra Block Sorting Kernel

Algorithm 5 Initial Intra Block Sorting Kernel utilizing shared memory

```
procedure INTRABLOCKSORTSHARED($data, size, max\_intra\_block\_dimension$)
    $local\_tid \leftarrow$ Local thread ID within the block

    Allocate shared memory $sharedData$
    $sharedData \leftarrow$ Load data from global memory
    __syncthreads()                              ▷ Sync threads before start exchanging

    for $dimension \leftarrow 1$ to $max\_intra\_block\_dimension$ do
        $isAscending \leftarrow$ getSortingDirection($global\_tid, dimension$)
        for $distance \leftarrow 2^{dimension-1}$ downto 1 do
            $partner \leftarrow local\_tid \oplus distance$
            if $partner > local\_tid$ then
                if $(sharedData[local\_tid] > sharedData[partner]) == isAscend$ then
                    swap($sharedData[local\_tid], sharedData[partner]$)
                end if
            end if
            __syncthreads()
        end for
    end for
    Copy $sharedData$ back to $data$             ▷ Write sorted data back to global memory
end procedure
```

Enhancements in `BitonicSortV2`

1. **Shared Memory Sort Kernel (`intraBlockSortShared`):** This kernel optimizes intra-block Bitonic Sorting by utilizing shared memory. Instead of repeatedly accessing global memory, data is loaded from global memory to shared memory at the beginning of the kernel, and the sorting is performed entirely within shared memory. The sorted data is then written back to global memory at the end of the kernel. This reduces global memory accesses, significantly improving performance by mitigating one of the major bottlenecks in GPU computation.

2. **Shared Memory Merge Kernel (`intraBlockMergeShared`):** Similarly to the previous kernel, this one also leverages shared memory to perform intra-block merging efficiently. By keeping data in shared memory during the merge process, it significantly reduces global memory accesses and improves performance. However, inter-block merging still requires global memory operations and synchronization across blocks, which is handled separately.

5 Version 3 - Register-Based Intra-Warp Exchange

Version 3 introduces an optimization strategy by performing intra-warp exchanges directly on registers, minimizing shared memory accesses and reducing memory latency. Furthermore, unlike previous versions where n threads were spawned to handle n elements, Version 3 spawns only n/2 threads, each responsible for a single swap between two elements. It also employs pinned memory to accelerate host–device data transfers.

1. **Register-Based Intra-Warp Exchange:** This kernel performs exchanges within a warp when the exchange distance is less than `warpSize/2`, utilizing registers for data storage and exchange. By minimizing shared memory accesses and leveraging the high-speed registers available to each thread, this approach further reduces memory latency and improves performance. Since all threads within a warp execute in lockstep, intra-warp exchanges are performed without explicit synchronization, as there is no risk of thread divergence or concurrent modifications by other warps.

2. **Pinned Memory for Host-Device Transfers:** Version 3 also employs pinned memory (also known as page-locked memory) for host-device data transfers. Pinned memory enables direct memory access (DMA) between the host (CPU) and the device (GPU), bypassing the overhead associated with pageable memory. This optimization reduces the time spent on memory transfers, further enhancing the overall performance of the Bitonic Sort algorithm.

6 Radix Sort Comparison

To provide a comprehensive performance analysis and establish a baseline for our Bitonic Sort implementations, we compare them against `cub::DeviceRadixSort` algorithm from the Device Wide Primitives[3]. Radix Sort serves as an excellent benchmark because it is a non-comparison-based sorting algorithm known for its efficiency on GPUs. Like V3, it also utilizes pinned memory to accelerate host-device data transfers. By comparing against `cub::DeviceRadixSort`, we can assess how close our implementations come to the performance of a highly tuned, industry-standard radix sort.

6.1 Radix Sort Characteristics

- **Efficient Sorting via Bitwise Operations:** Radix Sort exploits the bit-level representation of data to achieve an average time complexity of $O(nk)$, where n is the number of elements and k is the number of bits required to represent the largest number. It processes elements based on their bit representation, typically starting from the least significant bit (LSB) and progressing to the most significant bit (MSB). This bitwise approach avoids costly comparison operations, which are the fundamental operations in comparison-based sorts like Bitonic Sort.

- **Memory Access Efficiency:** While Radix Sort requires additional memory for temporary buffers (often proportional to the input size), it benefits from coalesced memory accesses and efficient data movement patterns. However, the additional memory footprint can be a disadvantage in memory-constrained environments.

7 Performance Analysis

7.1 Experimental Setup

We conduct our experiments on three NVIDIA GPUs: the Turing-based T4, the Pascal-based Tesla P100, and the Ampere-based A100. These GPUs offer varying levels of compute resources, memory configurations, and performance characteristics, allowing us to analyze the impact of hardware differences on the sorting algorithms' performance.

GPU Model	**Memory Config.**	**Compute Resources**	**Performance Characteristics**
NVIDIA T4 (Turing)	16GB GDDR6 256-bit Bus	2560 CUDA Cores 320 Tensor Cores Compute Capability 7.5	320 GB/s Bandwidth 8.1 TFLOPS FP32
NVIDIA Tesla P100 (Pascal)	16GB HBM2 4096-bit Bus	3584 CUDA Cores 224 Tensor Cores Compute Capability 6.0	732 GB/s Bandwidth 9.3 TFLOPS FP32
NVIDIA A100 (Ampere)	40GB HBM2 5120-bit Bus	6912 CUDA Cores 432 Tensor Cores Compute Capability 8.0	1555 GB/s Bandwidth 19.5 TFLOPS FP32

Table 1: GPU Specifications and Performance Characteristics

7.2 Performance Metrics

We evaluate the performance of the different Bitonic Sort implementations using the following metrics:

- **Total Execution Time:** This metric measures the wall-clock time for the entire sorting process, providing an end-to-end performance evaluation. It starts from the initiation of data transfer from the host (CPU) to the device (GPU) and ends when all sorted data has been transferred back to the host. It encompasses the time spent on data transfers, kernel execution, CUDA runtime overhead, and any other system-level delays. This is the primary metric for comparing the overall performance of the different versions, reflecting real-world sorting time.
- **Kernel Execution Time:** This metric quantifies the duration of CUDA kernel execution on the GPU. It provides a breakdown analysis of the time spent in each kernel, allowing us to isolate the performance of the core sorting algorithms. Kernel execution times are measured for each individual kernel (`intraBlockSort`, `compareSwapV0`, `intraBlockMerge`, and their shared memory counterparts) to determine their respective contributions to the overall execution time.

7.3 Experimental Results

7.3.1 Comparative Performance Analysis

We evaluate the performance of the Bitonic Sort implementations (V0, V1, V2 and V3) accelerated with CUDA, comparing them against the highly optimized `cub::DeviceRadixSort`.

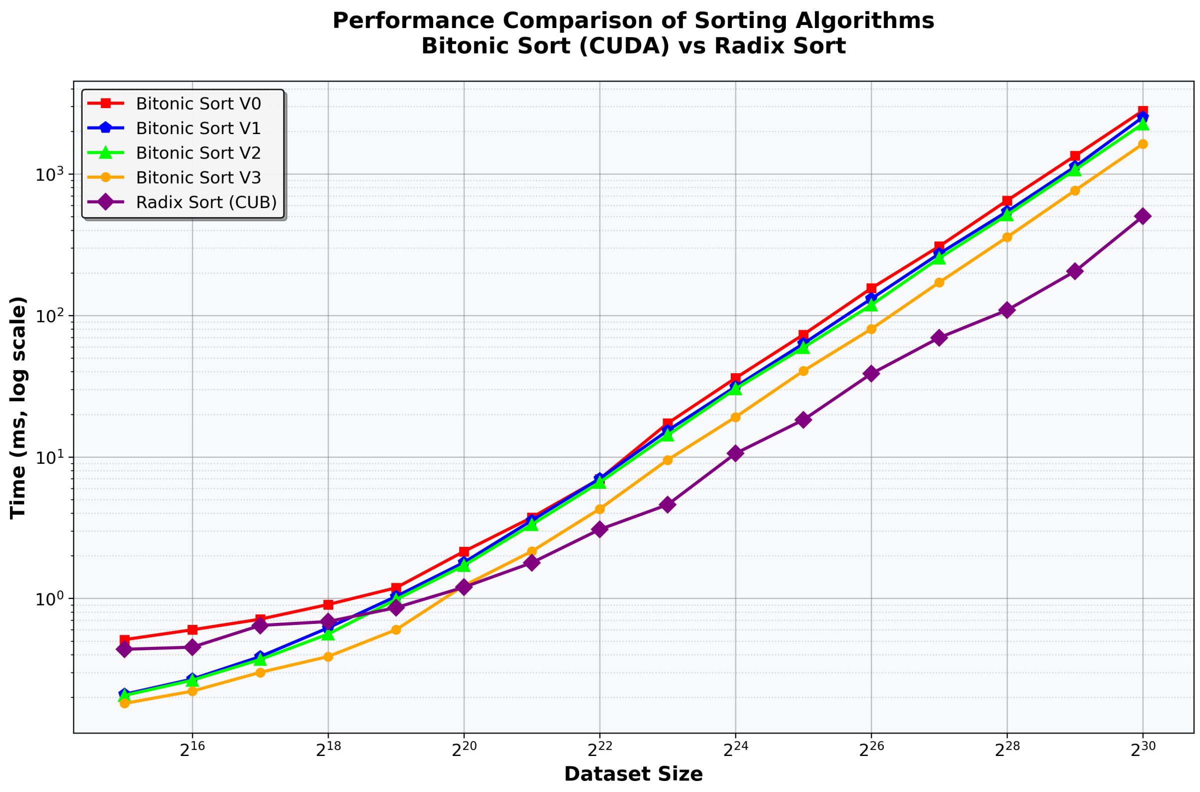

Figure 3: Total Sorting Time vs. Dataset Size on NVIDIA A100 GPU

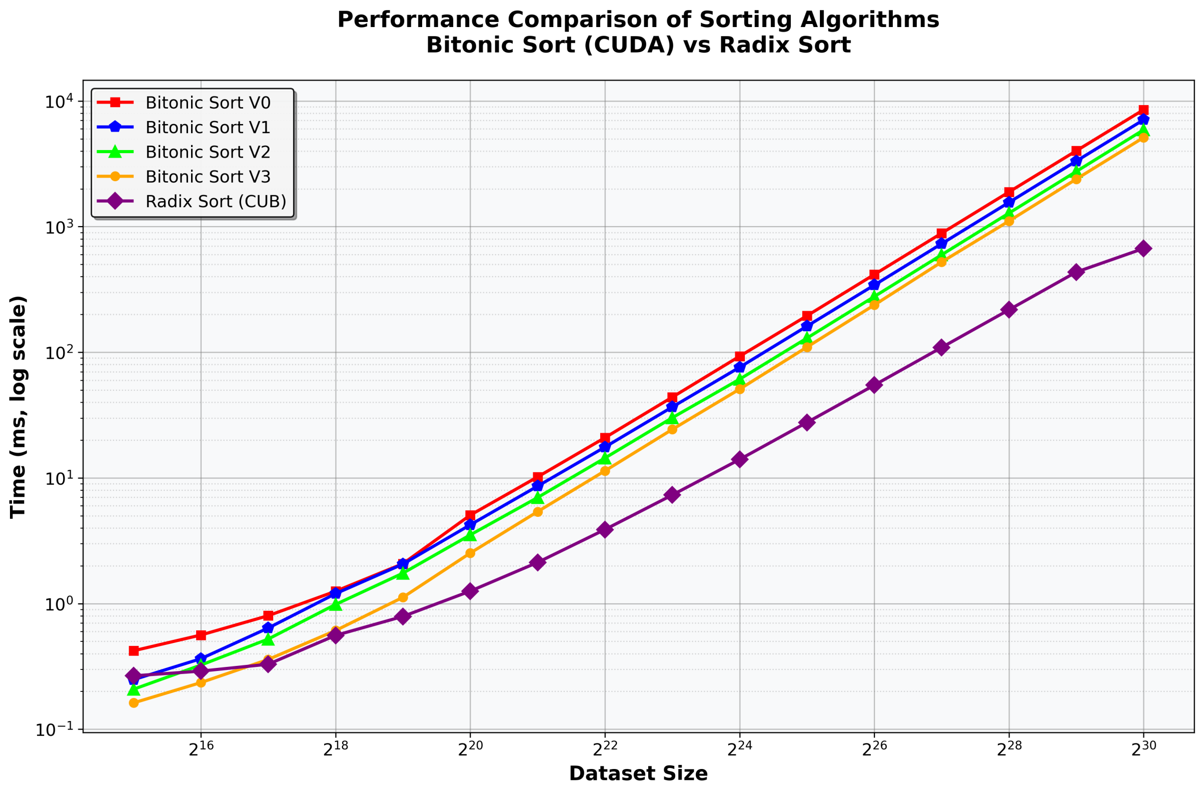

Figure 4: Total Sorting Time vs. Dataset Size on NVIDIA P100 GPU

Discussion of Results The figures present the total sorting time as a function of dataset size for all implementations, with both axes using a logarithmic scale for better visualization of performance trends across a wide range of input sizes.

1. **Version Comparison**
   - **Version 0 (Baseline):** As expected, V0 exhibits the worst performance across all dataset sizes and both GPUs. The high kernel launch overhead, coupled with implicit global synchronization after each kernel invocation and the lack of shared memory utilization, significantly limits its scalability and efficiency.
   - **Optimized Versions:** Versions 1, 2, and 3 demonstrate substantial performance improvements over the baseline (V0). Version 1 employs kernel fusion and intra-block sorting to reduce overhead. Version 2 incorporates shared memory, resulting in slightly better performance than V1, particularly on the P100 GPU. Version 3 leverages register-based intra-warp exchanges and pinned memory, leading to the best performance, with a more pronounced effect on the A100 GPU.
   - **CUB Radix Sort:** The `cub::DeviceRadixSort` consistently outperforms all Bitonic Sort implementations for larger datasets. This highlights the inherent advantages of radix sort for general-purpose GPU sorting and underscores the importance of choosing the appropriate sorting algorithm for specific use cases.

2. **Scaling Characteristics**
   - **Crossover Point:** Interestingly, for smaller datasets (up to approximately 2^{20} elements on the A100 and a slightly smaller size on the P100), Bitonic Sort V3 outperforms `cub::DeviceRadixSort`. This seemingly counterintuitive result arises because, for small datasets, the overhead associated with CUDA's Radix Sort (including memory allocations and kernel setup) outweighs its algorithmic advantages. Furthermore, the profiling data for V3 (Table 2) reveals that the `compareSwapV0` kernel, while still significant, does not yet dominate the execution time for these sizes, allowing the benefits of register-based intra-warp exchanges to shine through.
   - **Radix Sort Scaling:** Beyond this crossover point, the `cub::DeviceRadixSort` exhibits superior scaling behavior. Its execution time increases slower with dataset size compared to the Bitonic Sort implementations. The well tuned kernels and efficient memory access patterns of the CUB library's radix sort enable it to leverage the GPU's parallel processing capabilities very effectively as the data size grows.
   - **Bitonic Sort Scaling:** While Version 3 of the Bitonic Sort shows significant performance improvements over previous implementations, its scaling behavior is not as efficient as the Radix Sort. As the dataset size increases, the performance gap between the Bitonic Sort implementation and the Radix Sort widens considerably.

Impact of GPU Architecture: The general performance trends observed across the different Bitonic Sort versions and the Radix Sort are consistent across both the A100 and P100 GPUs. While the A100 generally exhibits higher absolute performance due to its more advanced architecture, the relative performance gains from each optimization are similar across both GPUs. This indicates that the implemented optimizations are broadly effective and not reliant on specific architectural features of either GPU.

7.3.2 Profiling Results

Detailed GPU profiling reveals the distribution of execution time across different kernels, providing valuable insights into the effectiveness of our optimization strategies. The results, summarized in following Tables, demonstrate the impact of kernel fusion and shared memory utilization on overall performance.

Table 2: Kernel Time Breakdown - Version 0

Input Size	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}	2^{29}
Total Time (ms)	101.96	211.51	448.89	954.98	2032.17	4321.68
Kernel Time Distribution (%)						
compareSwapV0	85.34	86.43	87.49	88.34	89.07	89.70
Memory Transfer (H2D)	8.81	8.28	7.66	7.17	6.72	6.36
Memory Transfer (D2H)	5.86	5.29	4.84	4.48	4.21	3.94

Analysis of Version 0: As seen in Table 2, the `compareSwapV0` kernel dominates the execution time, accounting for over 85% of the total time. This highlights the overhead associated with launching a separate kernel for each comparison step. The high percentage of time spent in `compareSwapV0` indicates that this kernel is a significant bottleneck.

Table 3: Kernel Time Breakdown - Version 1

Input Size	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}	2^{29}
Total Time (ms)	78.40	166.61	350.04	749.66	1600.94	3418.19
Kernel Time Distribution (%)						
compareSwapV0	34.74	37.39	39.85	42.10	44.34	46.45
intraBlockMerge	30.07	30.24	29.64	29.47	29.07	28.78
intraBlockSort	16.24	15.23	14.43	13.54	12.60	11.80
Memory Transfer (H2D)	11.40	10.45	9.88	9.17	8.65	7.98
Memory Transfer (D2H)	7.55	6.70	6.20	5.72	5.34	4.98

Analysis of Version 1: Table 3 shows a significant reduction in the percentage of time spent in the `compareSwapV0` kernel, with the introduction of kernel fusion achieving approximately **30% speedup over Version 0**. The overall sorting process now takes about 5% less of the total execution time compared to V0, indicating more efficient computational operations. For example, for a data size of 2^{28}, the number of all sorting kernel launches was reduced from 406 to 190. This shift demonstrates that our kernel fusion optimization successfully reduced the sorting overhead.

Table 4: Kernel Time Breakdown - Version 2

Input Size	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}	2^{29}
Total Time (ms)	65.03	133.54	287.24	619.07	1334.06	2865.97
Kernel Time Distribution (%)						
compareSwapV0	41.95	45.87	48.55	50.97	53.22	55.41
intraBlockMergeShared	26.54	25.04	24.76	24.38	23.93	23.50
intraBlockSortShared	8.71	7.65	7.09	6.57	6.09	5.67
Memory Transfer (H2D)	13.74	13.05	12.02	11.15	10.36	9.49
Memory Transfer (D2H)	9.06	8.39	7.58	6.93	6.39	5.93

Analysis of Version 2: Table 4 shows the impact of shared memory utilization. The percentage of time spent in `intraBlockSortShared` and `intraBlockMergeShared` is significantly lower than in `intraBlockSort` and `intraBlockMerge` of the previous version, indicating that shared memory reduces memory latency, achieving around **20% speedup compared to Version 1**. However, the percentage of time spent in `compareSwapV0` increases, suggesting that this kernel is becoming a bottleneck (in reality the absolute time for compareSwap kernel remains roughly the same[2] , but represents a larger portion of the reduced total execution time).

Table 5: Kernel Time Breakdown - Version 3

Input Size	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}	2^{29}
Total Time (ms)	53.47	115.18	241.94	522.43	1126.30	2427.59
Kernel Time Distribution (%)						
compareSwapV0	47.59	50.10	53.77	55.93	57.95	59.76
intraBlockMergeShared	24.53	24.12	22.40	22.01	21.59	21.04
intraBlockSortShared	8.22	7.56	6.50	6.02	5.58	5.40
Memory Transfer (H2D)	10.11	9.37	8.91	8.25	7.65	7.10
Memory Transfer (D2H)	9.55	8.85	8.41	7.79	7.22	6.70

Analysis of Version 3: Table 5 shows the impact of 1) reduced kernel launches, 2) register-based intra-warp exchange, and 3) pinned memory. Although the number of `compareSwapV0` kernel launches is halved, leading to a reduction in its absolute execution time, its relative contribution to the total execution time continues to increase, indicating that this kernel has become the primary performance bottleneck. Meanwhile, the overall execution time is further reduced compared to Version 2, achieving an **additional 20% speedup**. Furthermore, the memory transfer overhead is significantly reduced, with the use of pinned memory nearly halving the host-to-device (H2D) absolute transfer time.

[2]See raw profiling data at https://github.com/fraidakis/PDS_BitonicSortCUDA/tree/main/results

8 Conclusion

This report presented a detailed analysis of a GPU-accelerated Bitonic Sort implementation using CUDA. We demonstrated that kernel fusion (Version 1), shared memory utilization (Version 2), and register-based intra-warp exchange progressively improve performance compared to a baseline implementation (Version 0). Version 3 further optimized performance by leveraging register-based intra-warp exchanges and pinned memory for host-device data transfers. While the optimized Bitonic Sort implementation (V3) outperformed `cub::DeviceRadixSort` for small datasets, the radix sort exhibited superior scaling behavior for larger inputs. Profiling results revealed that the `compareSwapV0` kernel consistently emerges as a significant bottleneck across all optimized versions, suggesting that further optimizations should focus on minimizing global memory accesses and improving the efficiency of this kernel.

9 Acknowledgments

Computational resources were provided by the Aristotle University of Thessaloniki (AUTH) through access to the Aristotelis High Performance Computing (HPC) cluster.

References

[1] NVIDIA. (2023). CUDA C++ Programming Guide. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

[2] NVIDIA. (2023). CUDA Runtime API. [Online]. Available: `https://docs.nvidia.com/cuda/cuda-runtime-api/index.html`

[3] NVIDIA. (2021). CUB: A C++ library for CUDA. [Online]. Available: `https://nvlabs.github.io/cub/`

[4] Aristotelis High Performance Computing Cluster. *Documentation.* [Online]. Available: `https://hpc.it.auth.gr/`

[5] The complete source code for the CUDA Bitonic Sort implementation is available on GitHub at: `https://github.com/fraidakis/PDS_BitonicSortCuda`.