



UNIVERSITÀ DI PISA

Master Degree in Computer Engineering

Computer Architecture Project Report

Parallel Bitonic Sort

Team Members:

Francesco Martoccia

Salvatore Lombardi

Academic Year 2023/2024

Contents

1	Introduction	2
1.1	Bitonic Sort	2
2	CPU implementation	4
2.1	CPU Specifications and Environment	4
2.2	First Version	5
2.2.1	Implementation	5
2.2.2	Tests and Results	6
2.2.3	Performance Analysis	9
2.3	Optimized Version	11
2.3.1	Implementation	12
2.3.2	Improvements Achieved	13
2.3.3	Performance Analysis	14
2.3.4	Tests and Results	14
3	GPU implementation	19
3.1	GPU Specifications and Environment	19
3.2	First version	20
3.2.1	Implementation	20
3.2.2	Tests and Results	21
3.2.3	Performance Analysis	23
3.3	Optimized version	26
3.3.1	Implementation	26
3.3.2	Performance Analysis	28
3.3.3	Tests and Results	30

1 - Introduction

Sorting is a fundamental operation in computer science, with applications ranging from data processing to algorithm optimization.

Among the many sorting algorithms, **Bitonic Sort** is particularly interesting due to its structured, parallelizable nature, which makes it well suited for implementation in modern hardware architectures such as CPUs and GPUs.

Bitonic Sort is a comparison-based sorting algorithm that operates on elements arranged into **bitonic sequences**. The algorithm works by recursively merging these sequences in sorted order, using a divide-and-conquer approach.

Its highly regular data access patterns and inherent parallelism make it an ideal candidate for efficient execution on multi-threaded CPUs and massively parallel GPUs.

1.1 Bitonic Sort

A bitonic sequence is a sequence of numbers that is monotonically increasing and then decreasing or can be transformed into such a sequence through rotation.

For example, [2, 3, 4, 8, 6, 5] is a bitonic sequence.

The algorithm requires the input array to have a length that is a **power of 2** and comprises **two** main phases:

- **Constructing a bitonic sequence**
- **Sorting of bitonic sequence**

1. Constructing a Bitonic Sequence

Initially, the input sequence is not necessarily bitonic. The algorithm transforms it into a bitonic sequence by repeatedly merging smaller sequences into larger bitonic sequences. This process involves splitting the sequence into two parts, sorting one part in ascending order and the other in descending order. These smaller parts are recursively merged into larger bitonic sequences until the entire array is transformed.

2. Sorting of Bitonic Sequence

Once a bitonic sequence is constructed, it can be sorted in ascending or descending order using the **Bitonic merge** operation. The merge works as follows:

- **Step 1: Compare and swap** elements across the bitonic sequence at specific distances (called the "stride"). For example, compare the first half of the sequence with the second half.

- **Step 2:** Repeat the process within each half, recursively narrowing the comparisons to smaller subsequences. This operation progressively refines the order of the sequence, eventually resulting in a fully sorted array.

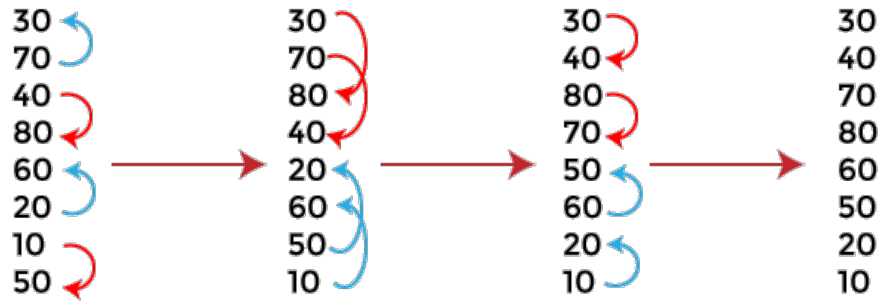


Figure 1.1: Bitonic Sequence Construction Phase

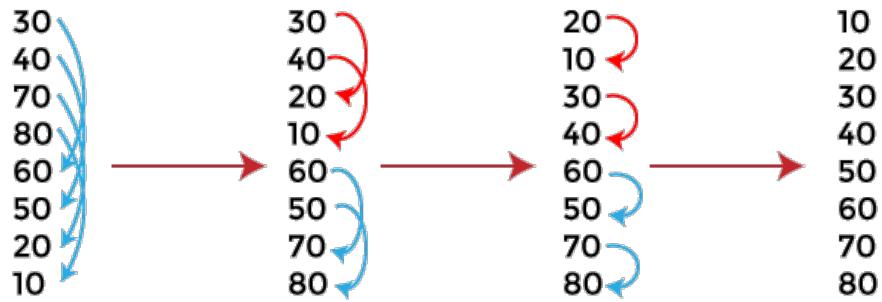


Figure 1.2: Bitonic Sequence Sorting Phase

2 - CPU implementation

2.1 CPU Specifications and Environment

This section outlines the hardware and software environment used for the implementation of the CPU-based Bitonic Sort algorithm, providing an overview of the computational resources and tools leveraged for efficient execution and parallel processing.

Hardware Specifications

The following table presents the specifications of the **CPU** used to implement and evaluate the Bitonic Sort algorithm, highlighting its topology, clock speeds, and cache configurations.

Feature	Details
Name	AMD Ryzen 9 7940HS with Radeon 780M Graphics
Topology	1 physical processor; 8 cores; 16 threads
Logical CPU Config	16x @ 5263 MHz
Clock Speed	400 MHz - 5263 MHz (16x)
Cache Levels	Level 1 (Data): 8x 32 KB (256 KB total), 8-way set-associative, 64 sets
	Level 1 (Instruction): 8x 32 KB (256 KB total), 8-way set-associative, 64 sets
	Level 2 (Unified): 8x 1024 KB (8192 KB total), 8-way set-associative, 2048 sets
	Level 3 (Unified): 1x 16384 KB, 16-way set-associative, 16384 sets

Table 2.1: CPU Specifications

Software Specifications

The software environment utilized for the implementation and testing of the CPU-based Bitonic Sort algorithm is outlined below:

- **Programming Language: C++20.**

The CPU implementation of the Bitonic Sort algorithm is written in C++20, utilizing modern C++ features such as threading and memory management.

- **Compiler: GCC** (GNU Compiler Collection) or **Clang** for C++ compilation.

The code is compatible with any modern C++ compiler that supports C++20 standards.

- **Parallelism: OpenMP.**

The second implementation utilizes OpenMP (Open Multi-Processing) for multi-threading on the CPU, allowing the program to efficiently utilize multiple CPU cores during execution.

- **Profilers:**

- **AMD μ Prof.**

It provides detailed performance analysis, including function execution time, memory utilization, and CPU core usage, specifically designed for applications running on AMD processors.

- **Visual Studio CPU Usage Tool.**

It is a performance profiling utility that helps to analyze how the application utilizes CPU resources. By providing detailed insights into CPU consumption, the tool enables identification of bottlenecks, inefficiencies, and potential areas for optimization.

2.2 First Version

In this section, the implementation of the first version of the algorithm will be presented.

2.2.1 Implementation

The code implements the algorithm on CPU, allowing the use of a multi-threaded approach. Here's a detailed explanation of the functions:

- ***sortCPUv1***: it is the entry point for the CPU-based bitonic sort
- ***bitonicSort***: it orchestrates the overall bitonic sorting process on the CPU
- ***compareAndSwap***: it performs the core comparison and swapping operations for the bitonic sort

The ***sortCPUv1*** function accepts as parameters the **input array**, its **length**, the desired **sort order** (ascending or descending), and the **number of threads** to use.

It initializes a timer to measure the execution time, calls the ***bitonicSort*** function to perform the sorting, and then outputs the time taken to sort the array.

The ***bitonicSort*** function performs the following steps:

1. **Padding the Input Array:** The array is padded to the nearest power of 2, as the bitonic sort algorithm requires the length of the array to be a power of 2.
2. **Chunk Size Calculation:** The input array is divided into equal-sized chunks based on the number of threads specified (**numThreads**). The size of each chunk is calculated as **paddedLength / numThreads**.
3. **Building Bitonic Sequences:** The outer loop iterates over increasing sizes of bitonic sequences, starting from 2 and doubling until it covers the full array length (**paddedLength**). This step ensures that smaller sorted sequences are combined iteratively to form larger bitonic sequences.

4. **Merging Sub-Sequences:** The middle loop (`mergeStep`) iteratively merges smaller sub-sequences within each bitonic sequence. The `mergeStep` starts from half the size of the current bitonic sequence and decreases until it reaches 1, progressively merging sorted parts.
5. **Parallel Processing:** For each merge step, threads are created to execute the `compareAndSwap` function in parallel. Each thread processes a chunk of the array, comparing and swapping elements to align them with the bitonic sequence order.
6. **Copying the Sorted Values:** Once the sorting is complete, the sorted values are copied back from the padded array (`paddedValues`) to the original array (`values`), discarding any padded elements.
7. **Handling Descending Order:** If the `sortOrder` parameter indicates descending order (0), the array is reversed after sorting to produce the desired order.

The ***compareAndSwap*** function identifies the pair of elements to compare based on the current merge step and bitonic sequence size.

Depending on whether the current subarray is sorted in ascending or descending order, the function determines if a swap is required. This ensures that each bitonic sequence is correctly merged into the desired order.

It is designed to be executed by multiple threads, each processing a chunk of the array.

The **input values** are generated as random **32-bit unsigned integers** using a **uniform distribution**. A high-resolution clock is used to generate a seed, ensuring variability for each run. This seed initializes the **Mersenne Twister** random number generator, which is known for its high-quality randomness and efficiency.

Moreover, the ***TimerCPU*** class was used to measure the time taken to execute the Bitonic Sort algorithm. The use of the `std::chrono` library and `high_resolution_clock` as clock type ensures precise time measurement.

2.2.2 Tests and Results

This section presents the results obtained from testing the Bitonic Sort algorithm under various configurations, including different numbers of threads and input array sizes, to evaluate its performance and efficiency across diverse scenarios.

The **performance metrics** used in this study were the **mean execution time (in milliseconds)** and the **speedup**.

Two distinct scenarios were considered: one involving **small input array sizes** and the other with **larger array sizes**. For each scenario, **three** different array sizes were selected, and, for each size, the number of threads was varied from **1** to **32**.

To evaluate the mean execution time in both scenarios, for each array size and a specific number of threads, the algorithm was executed **30 times**, and the corresponding mean execution time along with a **95% confidence interval** was calculated.

Mean Execution Time - Small Arrays Test

To conduct this type of test, arrays with a size of **512 KB**, **2 MB** and **4 MB** were used.

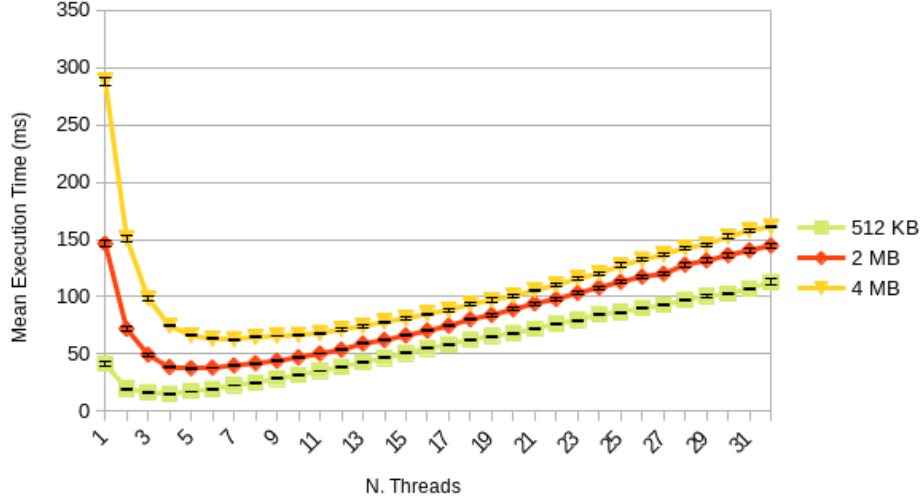


Figure 2.1: Mean Execution Time with small Array Sizes

The figure 2.1 demonstrates that as the number of threads increases, the mean execution time initially decreases, highlighting the benefits of parallelism in dividing the workload across multiple threads.

However, beyond a certain approximately "small" number of threads (e.g., 4 threads in case of array size of 512 KB), the mean execution time begins to rise. This increase can be attributed to **thread overhead**, such as context switching and synchronization costs, which surpass the performance gains provided by additional parallelism.

Generally, this effect is more pronounced for smaller array sizes, where the computational workload is insufficient to fully leverage the advantages of parallelism, resulting in a diminishing benefit compared to larger arrays.

Mean Execution Time - Large Arrays Test

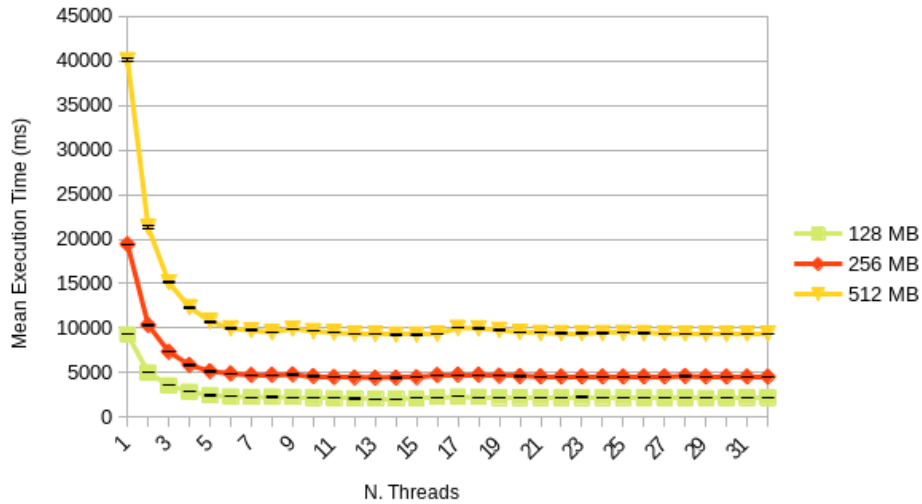


Figure 2.2: Mean Execution Time with big Array Sizes

The figure 2.2 illustrates the mean execution time for larger input sizes (**128 MB**, **256 MB**, and **512 MB** arrays) as the number of threads increases.

The benefits of parallelism are more pronounced in this scenario. Larger arrays exhibit a greater workload, which allows the system to better utilize the computational resources provided by additional threads.

The overhead is amortized, resulting in more sustained performance improvements and a flatter curve at higher thread counts.

This highlights that the effectiveness of parallelism is strongly dependent on the size of the input array, with larger inputs benefiting more from multi-threaded execution.

Speedup

Figure 2.3 shows the speedup obtained by considering all the various array sizes analyzed in the scenarios described above and varying the number of threads.

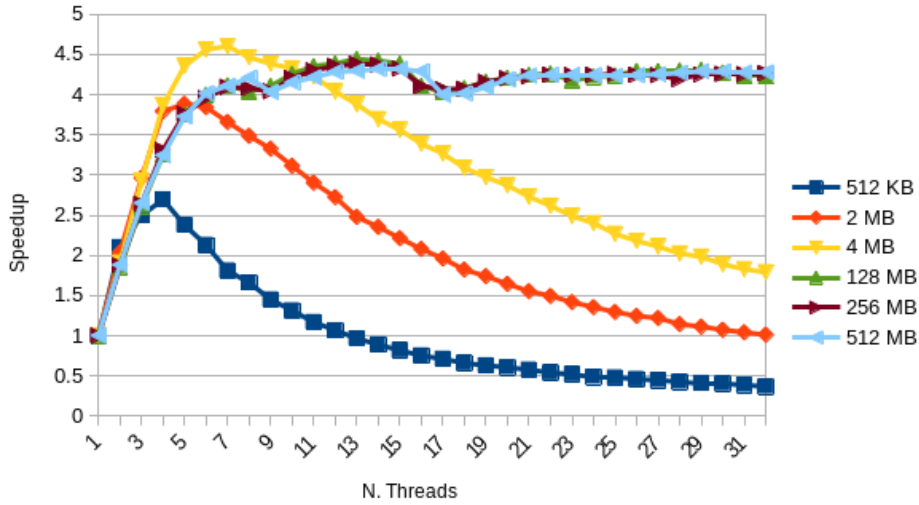


Figure 2.3: Speedup obtained for all the array sizes considered

It is possible to notice that, with regard to cases where the input size is smaller, the peak speedup is observed at a low number of threads.

Speedup drops off rapidly as the thread count increases, especially for the configuration with 512 KB as array size. In this particular case, starting with 13 threads, a negative speedup was obtained, indicating that the overhead from thread management and synchronization outweighs the benefits of parallel execution.

Unlike smaller arrays, larger inputs maintain relatively stable speedups across a higher range of thread counts.

So, in general, for the second scenario, as the input array size increases, the peak speedup shifts to configurations with a higher number of threads, reflecting the increased workload that can better leverage the parallelism provided by additional threads before overhead becomes dominant.

Moreover, for larger input sizes, the peak speedup is achieved when the number of threads is near the number of physical threads in the CPU. This demonstrates that increasing the number of threads beyond the physical core count generally results in reduced gains, as the overhead from context switching and resource contention outweighs the advantages of additional parallelism in scenarios without disk or inter-thread synchronization

requirements.

2.2.3 Performance Analysis

From the analysis conducted on speedup, although (as the number of threads increases) the latter remains more stable in the scenario with larger array sizes than the other considered, the greatest peak observed among all the configuration types was that obtained by the one with 4MB and 7 threads.

In order to provide an explanation for this result and, in general, to justify why the speedup is not higher in the scenario involving larger array sizes, the analysis focused on the potential influence of inter-thread synchronization and cache behavior.

The evaluation started by considering two configurations belonging to each scenario:

- Array size = 4 MB, N. of Threads = 7;
- Array size = 4 MB, N. of Threads = 13;
- Array size = 128 MB, N. of Threads = 7;
- Array size = 128 MB, N. of Threads = 13;

Next, Visual Studio's **CPU Usage Tool** was used to identify the function in the code with the highest CPU utilization.

For all configurations considered, the function that consumed the most CPU time during the execution of the algorithm was *CompareAndSwap*, as can be seen for example from the figure 2.4.

Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module	Category
ConsoleApplication1 (PID: 21128)	18339 (100,00%)	0 (0,00%)	Multiple modules	
[External Call] RtlUserThreadStart	18096 (98,67%)	8 (0,04%)	ntdll	Kernel Runtime
std::thread::Invoke<std::tuple<void (&__cdecl*)(std::vector<unsigned int, std::allocator<unsigned int>, 0, 0>>&...>	17650 (96,24%)	0 (0,00%)	consoleapplication1	Runtime
compareAndSwap	17629 (96,13%)	17528 (95,58%)	consoleapplication1	
[External Call] ucrtbase.dll!0x00007ffc0e6e2ad0	15 (0,08%)	15 (0,08%)	ucrtbase	Runtime
[External Call] msvcrt140.dll!0x00007ffc0e6e2ad0	2 (0,01%)	2 (0,01%)	msvcrt140	
[External Call] msvcrt140.dll!0x00007ffc0e6e2b1e	2 (0,01%)	2 (0,01%)	msvcrt140	
[External Call] msvcrt140.dll!0x00007ffc0e6e2ad5	1 (0,01%)	1 (0,01%)	msvcrt140	
[External Call] msvcrt140.dll!0x00007ffc0e6e2af0	1 (0,01%)	1 (0,01%)	msvcrt140	
_scrt_common_main_seh	438 (2,39%)	0 (0,00%)	consoleapplication1	Runtime
[External Call] LdrInitializeThunk	142 (0,77%)	142 (0,77%)	ntdll	Kernel
[System Code] 0xfffff8042362a408	78 (0,43%)	78 (0,43%)	[Unknown Code]	
[Unwalkable]	8 (0,04%)	0 (0,00%)		Kernel
[System Code] 0xfffff804236170c1	3 (0,02%)	3 (0,02%)	[Unknown Code]	
[System Code] 0xfffff804236178c7	3 (0,02%)	3 (0,02%)	[Unknown Code]	
[System Code] 0xfffff8042361da15	3 (0,02%)	3 (0,02%)	[Unknown Code]	
[System Code] 0xfffff8042361e148	3 (0,02%)	3 (0,02%)	[Unknown Code]	
[System Code] 0xfffff80423616677	1 (0,01%)	1 (0,01%)	[Unknown Code]	
[System Code] 0xfffff80423617047	1 (0,01%)	1 (0,01%)	[Unknown Code]	
[System Code] 0xfffff80423617941	1 (0,01%)	1 (0,01%)	[Unknown Code]	

Figure 2.4: Call Tree for the CPU Usage of the configuration with 128MB and 7 Threads

Figure 2.4 also shows that in the configurations belonging to the second scenario, the CPU devotes most of the execution time of the entire algorithm to the running of the considered function (about 96% CPU usage).

This allowed to conclude that the **low speedup was not due to inter-thread synchronization mechanisms**.

At this point, the AMD μ Prof profiler was leveraged, which provides detailed information on various aspects of system performance, including metrics related to cache usage. These metrics were analyzed to evaluate the efficiency of memory access patterns during the compareAndSwap function execution.

Among the many metrics available, the focus was on **IBS_LD_L1_DC_MISS_LAT (Instruction-Based Sampling for Load Level 1 Data Cache Miss Latency)**, as the results obtained for different configurations allowed significant conclusions to be drawn about the impact of caches.

The metric quantifies **the latency (in number of clock cycles) incurred due to misses in the level 1 data cache.**

Array Size	N. of Threads	IBS_LD_L1_DC_MISS_LAT
4 MB	7	17
4 MB	13	961
128 MB	7	633866
128 MB	13	957471

Table 2.2: L1 Cache Miss Latency Analysis for Different Configurations

From the table 2.2 it can be seen that the dramatic increase that occurs between the two 4MB configurations is **disproportionate** to that present in the other two considered and indicates a serious degradation in performance.

Indeed, since the array size is very small (4MB), when using 7 threads the workload is distributed across 4 physical processors (cores), and the threads can fully load the entire chunk into their respective cache memory. This means that each thread works mostly within its cache, **avoiding frequent cache misses.**

When increasing to 13 threads, more physical processors (7 cores) are involved to handle the extra threads and, therefore, the array needs to be shared across more caches.

This sharing leads to **cache contention**, where multiple processors compete for access to the same memory lines. As a result, **cache misses increase**, and threads are forced to fetch data from the slower main memory (DRAM) instead of cache, which adds considerable latency.

In configurations that include an array size of 128 MB, the increase of about 50% can be reasonably justified by the increase in the number of threads (7 to 13), as more threads naturally lead to higher resource utilization and, then, to higher number of cache misses. Moreover, the number of capacity misses in this scenario is so high that **the previously described phenomenon for smaller arrays, while present, is less pronounced.**

Shifting the comparison to the two scenarios, it is evident that there is a significant difference in L1 cache latency, which results in a lower speedup when considering larger array sizes compared to smaller ones. This highlights the impact of increased cache misses on performance as the workload scales.

2.3 Optimized Version

The goal of this part was to try to optimize the code to reduce the L1 cache latency and L1 cache misses produced by executing the first version of the algorithm on large size arrays and, consequently, improve speedup.

After analyzing the code, it was found that the main problem were **repetitive thread access** to the shared array, resulting in **poor data locality**.

Each thread performs comparisons and exchanges across the entire array, without localized processing, at each stage of the bitonic sort, resulting in scattered and unpredictable memory accesses. Since the bitonic sequence size doubles in each iteration, these become more widespread, leading to frequent replacements and increased misses, as the entire array cannot fit within the cache.

```
{
    unsigned int startIndex = threadId * chunkSize;
    unsigned int endIndex = (threadId + 1) * chunkSize;
    // Process the chunk assigned to this thread
    for (unsigned int currentIndex = startIndex; currentIndex <
        endIndex; currentIndex++) {
        // Find the element to compare with
        unsigned int compareIndex = currentIndex ^ mergeStep;
        // Only compare if the compareIndex is greater (to avoid
        // duplicate swaps)
        if (compareIndex > currentIndex) {
            bool shouldSwap = false;
            // Determine if we should swap based on the current
            // subarray's sorting direction
            if ((currentIndex & bitonicSequenceSize) == 0) {
                // First half of subarray (ascending)
                shouldSwap = (paddedValues[currentIndex] >
                    paddedValues[compareIndex]);
            } else {
                // Second half of subarray (descending)
                shouldSwap = (paddedValues[currentIndex] <
                    paddedValues[compareIndex]);
            }
            // Perform the swap if necessary
            if (shouldSwap) {
                std::swap(paddedValues[currentIndex],
                    paddedValues[compareIndex]);
            }
        }
    }
}
```

Listing 2.1: Code snippet of the *compareAndSwap* function

In the above function, `compareAndSwap`, each thread accesses elements in the array by computing the `currentIndex` and the corresponding `compareIndex`. This can lead to widespread access across the array because, depending on the `mergeStep`, the array elements accessed are far apart, thus preventing effective use of the CPU cache.

The `chunkSize`, although calculated by dividing the input size by the number of threads to be used, does not segment the array into well-localized sections and threads operate over wide intervals, increasing cache inefficiency.

2.3.1 Implementation

To address the issues and try to solve them, a second version of the algorithm was implemented, which provides better data locality, minimizes the synchronization overhead and balances the workload among threads. Within this, the **OpenMP** API was utilized to make better use of parallel processing.

Here's an explanation of the functions used:

- ***sortCPUv2***: is the entry point for the sorting process and its primary responsibility is to choose the appropriate sorting algorithm based on the size of the input array
- ***bitonicSortV2***: implements an optimized bitonic sort for small arrays
- ***bucketSort***: sorts large arrays by distributing elements into buckets based on the most significant byte

For arrays smaller than a predefined `BUCKET_THRESHOLD`, the ***sortCPUv2*** function uses ***bitonicSortV2***. This threshold ensures that the simpler, SIMD-optimized (Single Instruction Multiple Data) bitonic sort is used when it is more efficient.

bucketSort, instead, is selected for larger arrays.

Moreover, the number of OpenMP threads used is explicitly set based on the `numThreads` parameter.

In the ***bitonicSortV2*** the array is padded to the nearest power of two using `std::vector`, ensuring compatibility with the bitonic sort structure.

The OpenMP's `#pragma omp parallel for simd` directive is used for vectorization, enabling efficient execution of comparison and swap operations across the array.

Each element is compared with another at a distance determined by the current stage and a comparison distance. Depending on the k-th bit of the current index, ascending (when the bit is 0) or descending (when the bit is 1) order is enforced.

After sorting, padding elements are discarded.

The ***bucketSort*** function implements a **parallel bucket sort** for large arrays.

The operations performed can be grouped into the following 3 phases:

1. **Bucket Distribution**: elements are distributed into **256 buckets** based on their **most significant byte** (bits 24-31). This localizes elements into **smaller groups**, reducing the need for global array scans. The `#pragma omp for nowait` directive is used to allow threads to proceed without synchronization.
2. **Bucket Merging**: after parallel bucket distribution, thread-local buckets are merged into global buckets. A `#pragma omp critical` section prevents concurrent access to the shared buckets.
3. **Individual Bucket Sorting**: each bucket is sorted independently using ***bitonicSortV2***. Dynamic scheduling is applied to balance the load among threads, as bucket sizes can vary significantly.
4. **Sorted Results Collections**: sorted elements are collected back into the original array. Depending on the `sortOrder` (ascending or descending), the buckets are

concatenated in forward or reverse order, with elements within buckets reversed as needed.

```
// Initialize main buckets array - one bucket for each possible byte value (256 buckets)
std::vector<std::vector<uint32_t>> buckets(NUM_BUCKETS);
// First phase: Distribute elements into buckets
#pragma omp parallel
{
    // Create thread-local buckets to avoid synchronization overhead
    std::vector<std::vector<uint32_t>> localBuckets(NUM_BUCKETS);
    // Distribute elements to thread-local buckets
    // nowait allows threads to proceed without synchronization at loop end
    #pragma omp for nowait
    for (int i = 0; i < length; i++) {
        // Extract most significant byte (bits 24-31) for bucket index
        int bucket = (values[i] >> 24) & 0xFF;
        localBuckets[bucket].push_back(values[i]);
    }
}
```

Listing 2.2: Code snippet of the Bucket Distribution Phase

```
// Merge thread-local buckets into global buckets
// Critical section prevents concurrent access to shared buckets
#pragma omp critical
{
    for (int i = 0; i < NUM_BUCKETS; i++) {
        buckets[i].insert(buckets[i].end(),
                        localBuckets[i].begin(),
                        localBuckets[i].end());
    }
}
```

Listing 2.3: Code snippet of the Bucket Merging Phase

```
// Second phase: Sort individual buckets
// Use dynamic scheduling for better load balancing as bucket sizes may vary
#pragma omp parallel for schedule(dynamic)
for (int i = 0; i < NUM_BUCKETS; i++) {
    if (buckets[i].size() > 1) {
        // Sort each non-empty bucket using bitonic sort
        bitonicSortV2(buckets[i].data(), buckets[i].size());
    }
}
```

Listing 2.4: Code snippet of the Individual Bucket Sorting Phase

2.3.2 Improvements Achieved

The optimized version of the algorithm solved cache inefficiencies by making the following improvements:

- **Data Locality via Bucket Sort:** by dividing the array into buckets based on the most significant byte, *sortCPUv2* groups elements that share similar values into

smaller, more localized buckets. This bucket-based division ensures that each thread only processes a subset of the array at a time.

Since each bucket is smaller and can potentially fit into the CPU cache, **data locality improves**, reducing cache misses as the threads do not need to reload distant memory locations constantly.

- **Reduced Cache Contention with Thread-Local Buckets:** in the first phase of bucket sort, each thread operates on **its own local copy of the buckets**. This strategy reduces the number of accesses to shared memory locations, **minimizing cache contention** and allowing each thread to use its cache more efficiently. Moreover, by merging thread-local buckets into global buckets only once per stage, frequent shared memory updates are minimized and better data locality in the cache is maintained.
- **SIMD Optimization:** for smaller arrays or within each bucket in the bitonic sort, the SIMD optimizations allow for contiguous memory accesses. SIMD instructions enable each thread to work with multiple elements at once, reducing the number of individual loads and stores, thus **minimizing cache replacements**.

2.3.3 Performance Analysis

After making the previous changes to the algorithm code, Visual Studio's CPU Usage Tool was used again to find out which function used the most CPU time. This was done in order to compare the cache behavior of the two versions of Bitonic Sort and see if better results were achieved.

The tool showed that the function that occupied the most CPU time during the execution of the algorithm was **bitonicSortV2**.

The latter was then used to launch AMD's μ Prof profiler and compare the metric values obtained with those produced with the compareAndSwap function.

The comparison was made considering for both versions an input array size of **256 MB** and **16 threads**.

Version	IBS_LD_L1_DC_MISS_LAT	IBS_LD_LOCAL_DRAM_HIT
V1 (compareAndSwap)	2,871,344	3,802
V2 (bitonicSortV2)	80,980	215

Table 2.3: Cache Metrics for V1 and V2 Implementations (256 MB - 16 Threads)

As can be seen from the table 2.3, both the L1 cache miss latency and the number of DRAM accesses have been significantly reduced with the optimized version of the bitonic sort, also suggesting a possible improvement in mean execution time and speedup.

2.3.4 Tests and Results

To assess whether there had been any improvement in mean execution time and speedup, the same tests were conducted as for the first implementation and **a throughput of 1GB/sec was set as a target**.

Mean Execution - Time Small Arrays Test

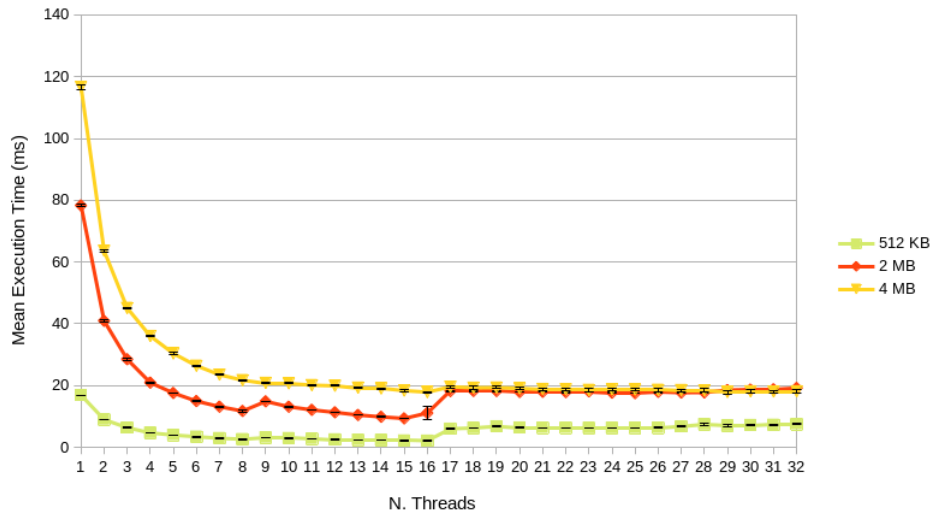


Figure 2.5: Mean Execution Time with small Array Sizes

The figure 2.5 shows the results obtained for the scenario with smaller arrays. It can be seen that for each of the configurations considered, a lower mean execution time was obtained than for the first version.

Furthermore, unlike the first version, after a certain number of threads, the mean execution time does not start to rise quickly, but remains constant or tends to increase very slowly. This is probably due to better management of thread synchronization.

Mean Execution Time - Large Arrays Test

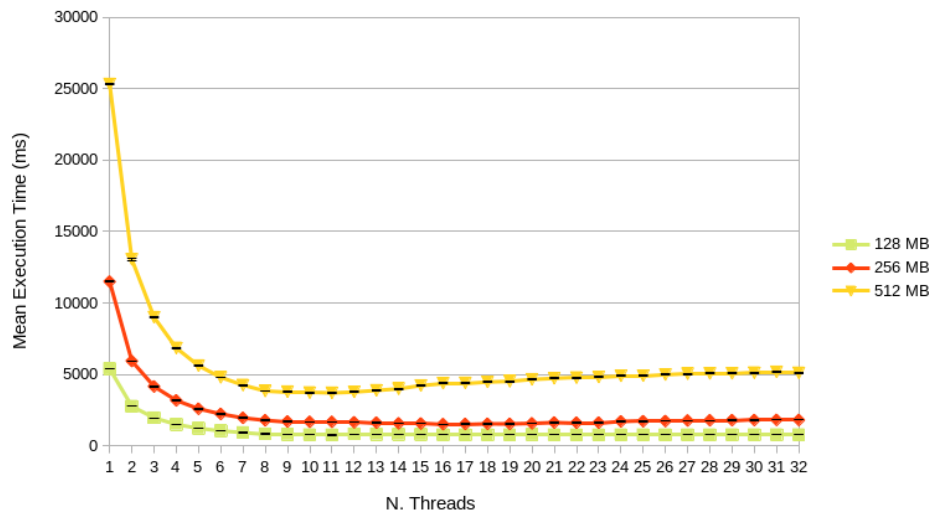


Figure 2.6: Mean Execution Time with big Array Sizes

Even in the scenario with larger array sizes, the mean execution time was much less than in the first version for all configurations tested.

Speedup

Besides evaluating the impact on cache behaviour and mean execution time, the new version was also evaluated in terms of speedup.

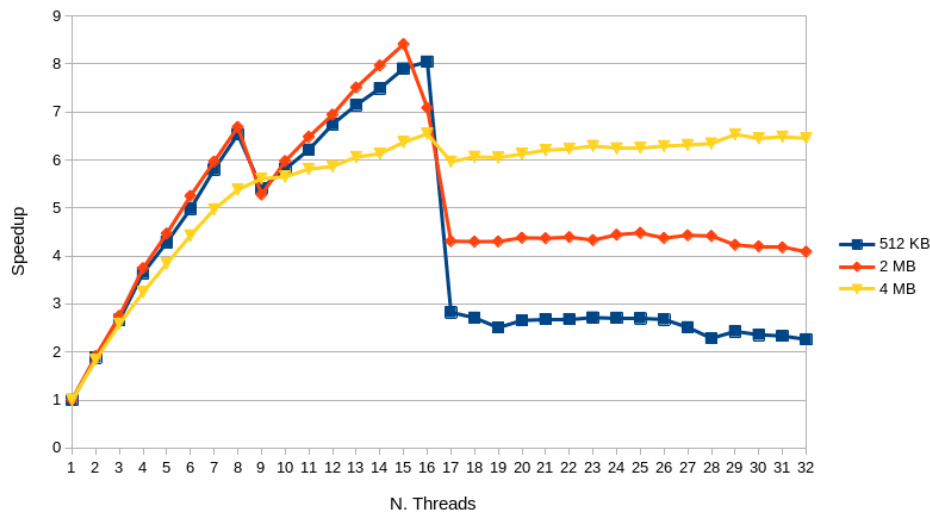


Figure 2.7: Speedup with small Array Sizes

For very small arrays (512 KB and 2 MB), the speedup decreases significantly after 16 threads (the configuration where the maximum peak was observed of all those considered). This indicates that the workload per thread becomes too small to justify the overhead of thread management and synchronization. It also confirms the fact that increasing the number of threads beyond the number of physical cores reduces the benefits of parallelization.

For the 4MB array, the speed stabilizes at around 6x after 16 threads, although it tends to drop off there too.

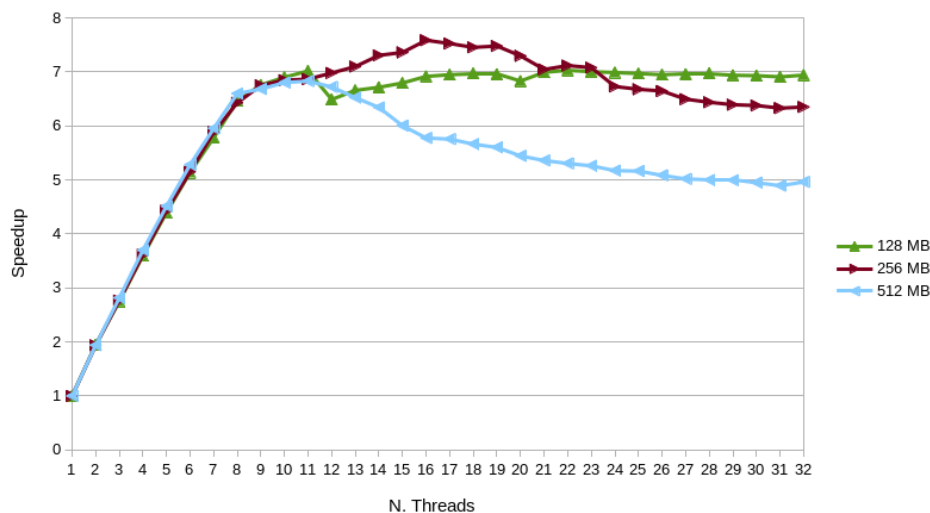


Figure 2.8: Speedup with big Array Sizes

In the second scenario, the speedup initially scales well with the number of threads, reaching a maximum at around 16 threads for the array sizes of 128 MB and 256 MB. Beyond

16 threads, the speedup stabilizes or decreases slightly but tends to be more constant than in the scenario considered above.

Larger arrays (512MB) begin to see a drop in speedup as soon as they exceed 11 threads. This may be due to a reduction in cache efficiency, although this has improved compared to the previous version of the algorithm.

Finally, a comparison was made between the two multi-threaded versions of the algorithm in terms of mean execution time and speedup. An array with a size of 256 MB was used as input, and the two metrics were evaluated by varying the thread count.

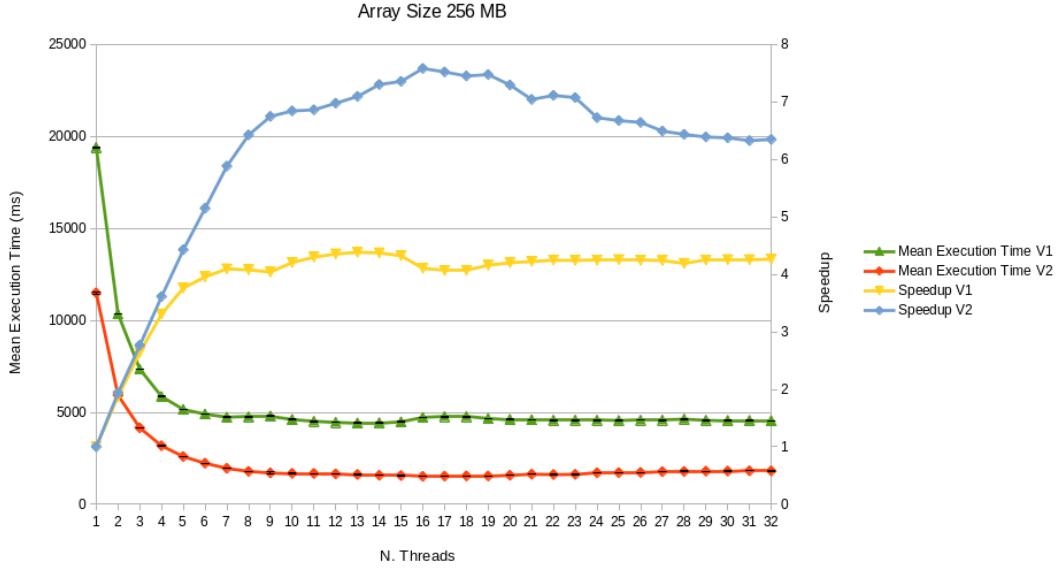


Figure 2.9: Comparison between the two versions of the Bitonic Sort algorithm

As can be seen from the figure 2.9, Version 2 has much lower execution times compared to Version 1 across all thread counts. This reflects an improvement in the efficiency of the optimized version's implementation.

Speedup was evaluated for each version by calculating the improvement achieved with configurations having between 2 and 32 threads compared to a single-thread configuration.

The second version outperforms the first in terms of speedup at all thread counts.

In the second version, the peak speedup occurs when the number of threads is equal to 16 (approximately 7.6x). In the first version, however, the benefits of parallelization tend to wear off before a number of threads equal to the number of physical processor cores is reached. In this case, the maximum speedup was indeed achieved in the 13-thread configuration (approximately 4.4x).

Throughput

Based on the results of the speedup evaluation, the algorithm was also analyzed in terms of throughput to verify if the second version was able to reach the established target of 1GB/s. The optimized implementation was tested by considering different input array sizes, with the 16-thread configuration (i.e. the one that allowed Bitonic Sort to achieve the highest peak speedup among all those used).

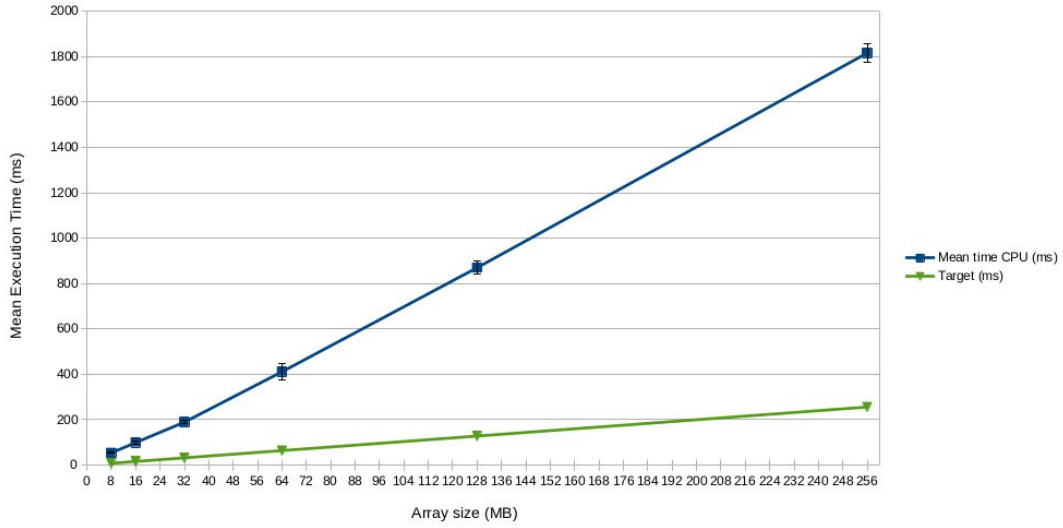


Figure 2.10: Mean Execution Time with different array sizes

The figure 2.10 shows that, despite the improvements achieved, the second version was not able to reach the target set in terms of throughput. In fact, the average value achieved was around **140MB/s**. For this reason, the algorithm was implemented on the GPU.

3 - GPU implementation

3.1 GPU Specifications and Environment

This section details the hardware and software environment utilized for the implementation of the Bitonic Sort algorithm. It provides a comprehensive view of the computational resources and tools employed.

Hardware Specifications

The experiments were conducted on an NVIDIA GeForce RTX 4070 Laptop GPU. The specifications of this GPU are listed in Table 3.1.

CUDA Capability	8.9
Number of Streaming Multiprocessors (SMs)	36
Global Memory	8 GB
L2 Cache Size	32 MB
Shared Memory per Multiprocessor	100 KB
Shared Memory per Block	48 KB
Constant Memory	64 KB
Registers per Block	65,536
Warp Size	32
Maximum Clock Rate	1.98 GHz
Memory Clock Rate	8001 MHz
Memory Bus Width	128-bit
Theoretical Bandwidth	256 GB/s
Maximum Threads per Multiprocessor	1536
Maximum Threads per Block	1024
Maximum Thread Block Dimensions (x, y, z)	(1024, 1024, 64)
Maximum Grid Dimensions (x, y, z)	(2147483647, 65535, 65535)

Table 3.1: NVIDIA GeForce RTX 4070 Laptop GPU Specifications

Software Environment

The software environment utilized for the implementation and testing of the Bitonic Sort algorithm is outlined below:

- **CUDA Version:** 12.60
- **Driver Version:** 560.35.03

- **Profiling Tools:**

- Nsight Compute: For profiling kernel execution and analyzing memory throughput and occupancy.

- **Programming Language:** CUDA-C++

- **Compiler:** NVCC (NVIDIA CUDA Compiler)

The specifications and tools described in this section establish the computational environment for the implementation. Further sections will detail the algorithm design and optimizations leveraging this setup.

3.2 First version

This section provides an overview of the unoptimized GPU implementation of the Bitonic Sort algorithm.

3.2.1 Implementation

Both the first and optimized versions share several common components and files, the main ones and their purposes are as follows:

- **bitonicSortGPU.cu/cuh:** Implements GPU kernels for the Bitonic Sort algorithm, including sorting and merging operations across blocks using CUDA.
- **constants.h:** Defines configuration parameters such as the number of threads, blocks, and sorting order for both GPU and CPU implementations.
- **utils.cu/cuh:** Provides utility functions for initializing arrays, logging results, handling file paths, and calculating block and thread offsets.
- **Sort.cu/cuh:** Encapsulates GPU sorting logic into a class interface, offering functions for memory allocation, kernel launches, and result retrieval.
- **TimerGPU.cu/cuh:** Measures kernel execution times on the GPU using CUDA events.
- **main.cu:** Acts as the program's entry point, orchestrating data initialization, sorting (on CPU and GPU), performance benchmarking, and result validation.

The GPU implementation of Bitonic Sort consists of several key components (defined in **bitonicSortGPU.cu**), each handling different aspects of the parallel sorting process.

The ***bitonicMergeStep*** device function implements the fundamental comparison-exchange operation of bitonic sort:

- Each thread processes multiple pairs of elements in a strided pattern, where in each iteration it compares and potentially swaps two elements based on the desired sort order. The number of pairs handled by each thread depends on the block size and number of threads.

- The function handles both regular merge steps and the specialized first step of each phase in bitonic sort.
- For the first step of each phase, it implements index mirroring within stride groups to maintain the bitonic sequence property.
- The stride parameter determines the distance between elements being compared, which doubles with each phase of the algorithm.

The ***bitonicSort*** kernel:

- Implements the bitonic sort algorithm, which requires special handling of the first step in each phase.
- The sorting process consists of multiple phases, with each phase containing multiple steps of comparison-exchange operations.
- Synchronization barriers (***syncthreads***) ensure proper ordering of operations.

The ***bitonicMergeGlobal*** handles the merging of sorted blocks:

- Implements the merge phase of bitonic sort across multiple thread blocks.
- Uses the same underlying comparison-exchange operation and operates on global memory.
- Handles both regular merge steps and the specialized first steps of each phase.

The main sorting process in ***bitonicSortParallel*** coordinates the overall execution:

- First, it calculates the next power of 2 for the input array length, as bitonic sort requires sequence lengths that are powers of 2.
- The algorithm then proceeds in two main stages:
 - Initial sorting of sub-blocks using global memory (***runBitonicSort***).
 - Global merging of sorted sub-blocks (***runBitonicMergeGlobal***).
- The number of phases is determined by the logarithm (base 2) of the array length.
- Each phase consists of multiple steps, with each step handling different stride lengths for comparison-exchange operations.

3.2.2 Tests and Results

This section summarizes the tests conducted to evaluate the performance of the **unoptimized GPU implementation** of the **Bitonic Sort** algorithm. Various experiments were performed to measure **execution time** and **scalability** under different input sizes and configurations.

Speedup with Different Grid Sizes

The objective of this test is to measure the **speedup** across varying **grid configurations** for a fixed array size of **128 MB**, starting with a **warp** (a single block of 32 threads) as a reference. Increasing the **number of blocks** leads to better performance up to a certain point, after which it becomes worse, as can be seen in **Figure 3.1**. The **optimal number of threads per block** in this case is **384**, while the **optimal number of blocks** is **4096**. This is the configuration that has been used to perform the next tests with a **fixed grid size**.

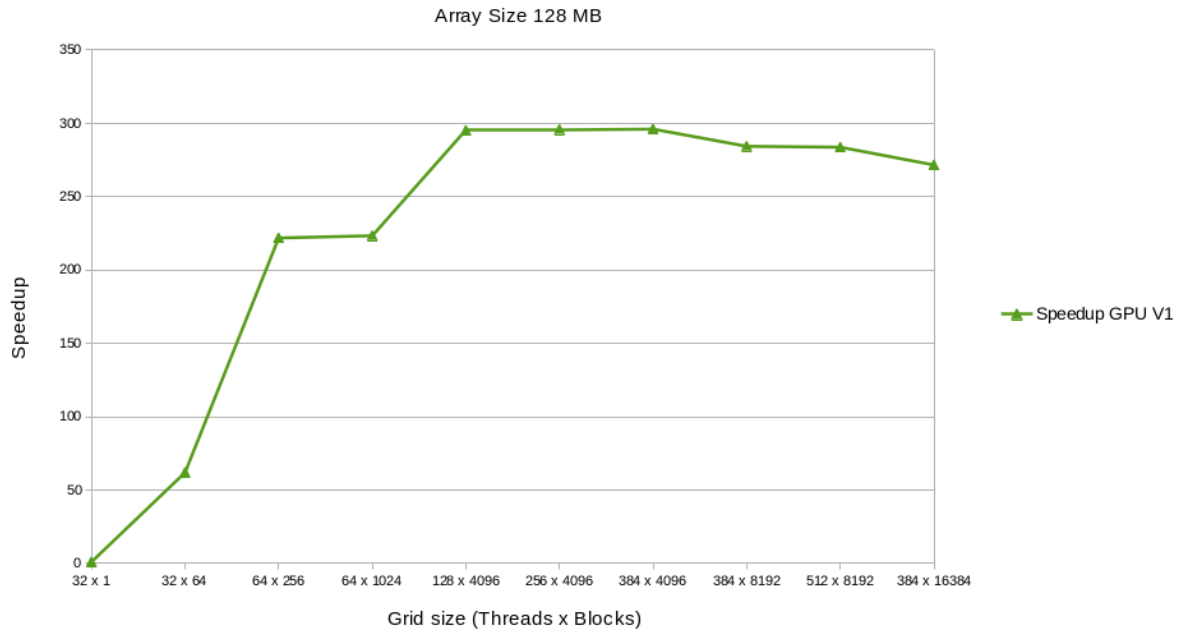


Figure 3.1: Speedup with Different Grid Sizes

Small Arrays Test

The aim of this test is to compare **execution times** for **small arrays** (1–8 MB) to evaluate the effectiveness of GPU parallelization in this range. As can be noted from figure 3.2, for very small arrays the GPU does not offer a significant speed advantage over the CPU. As expected, in this case, the **overhead** is prevalent, and the **benefits of parallelization** on the GPU are limited compared to the **multithreaded CPU version**. For this reason, we decided to evaluate the behavior of the implementation in more detail with **bigger array sizes** (starting from 8 MB).

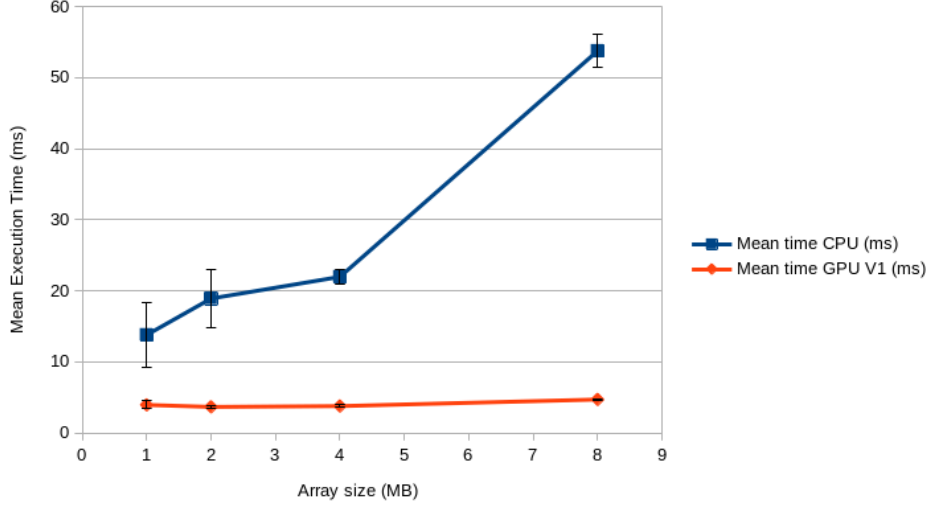


Figure 3.2: Mean Execution Time with Small Array Sizes

Scalability Test with Increasing Array Sizes

This test allows us to determine the **scalability** of the implementation as **array sizes** increase from **8 MB** to **256 MB**. The execution time **increases linearly** with the array size for both **CPU** and **GPU**. The GPU implementation exhibits better **scaling behavior** than the CPU, as shown in figure 3.3. Despite better scaling, the GPU implementation falls short of the desired **throughput target** of **1 GB/s** (for arrays larger than 32 MB). The mean throughput reached is **950 MB/s**. This indicates the need for further optimization.

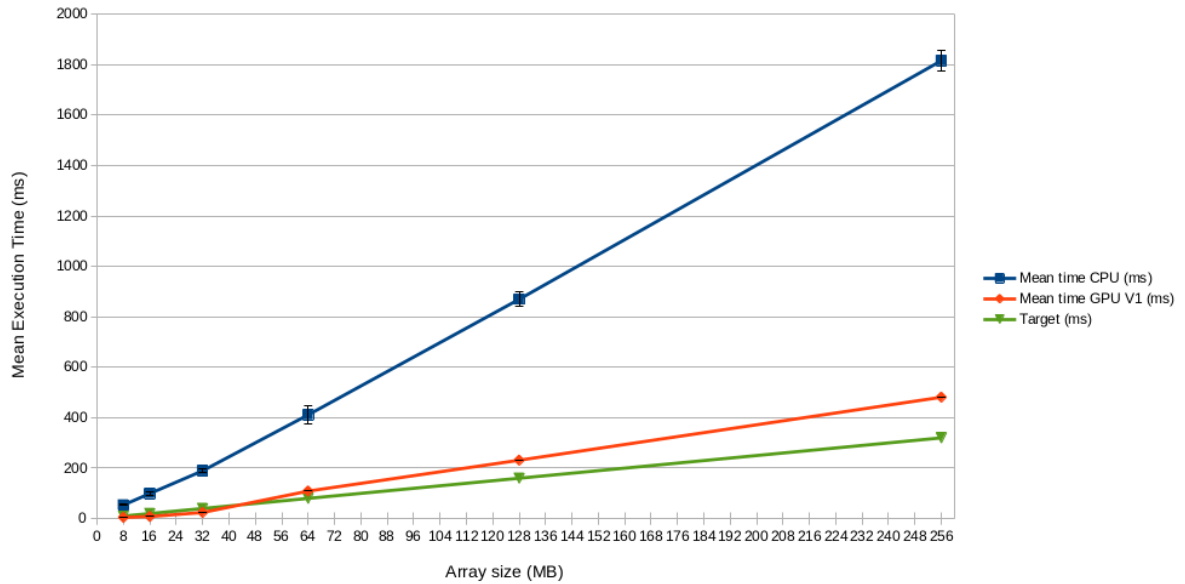


Figure 3.3: Mean Execution Time with different Array Sizes

3.2.3 Performance Analysis

In order to evaluate the behavior of the algorithm and identify potential performance issues, a **profiling analysis** was conducted using the **NVIDIA Nsight Compute** tool.

The profiling results provided valuable information on the execution characteristics of the implemented kernels. This section discusses the key findings, focusing on two kernels: *bitonicSort* and *bitonicMergeGlobal*. The analysis revealed several **performance-related challenges**, which are detailed in the following.

Bitonic Sort

The *bitonicSort* kernel was analyzed, and the main issue was identified in the **Warp State Statistics** section of the **Nsight Compute** profiler. This section provides an **analysis of the states** in which all **warps** spent cycles during the kernel execution. Warp states describe a warp's **readiness** or **inability** to issue its next instruction. Figure 3.4 shows the results obtained in terms of **warp cycles per issued instruction**, indicating the **latency** between two consecutive instructions.

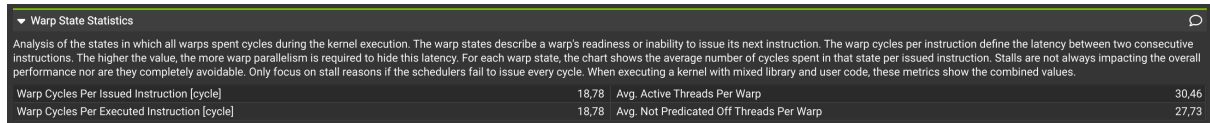


Figure 3.4: Warp Cycles per Issued Instruction for *bitonicSort* Kernel.

The tool highlighted a specific issue with **Long Scoreboard Stalls**, which represent approximately **33.3% of the total average** of **18.8 cycles** between issuing two instructions. The analysis estimates a **local speedup** of **33.33%** if this issue is mitigated (figure 3.5). As can be seen from figure 3.6, on average, each warp spends **6.3 cycles** stalled waiting for a **scoreboard dependency** related to **L1TEX operations** (local, global, surface, texture).

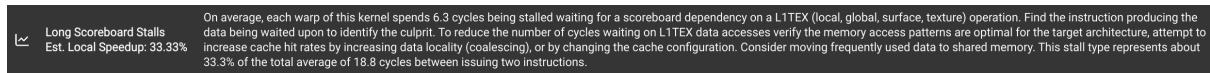


Figure 3.5: Estimated speedup and possible optimizations

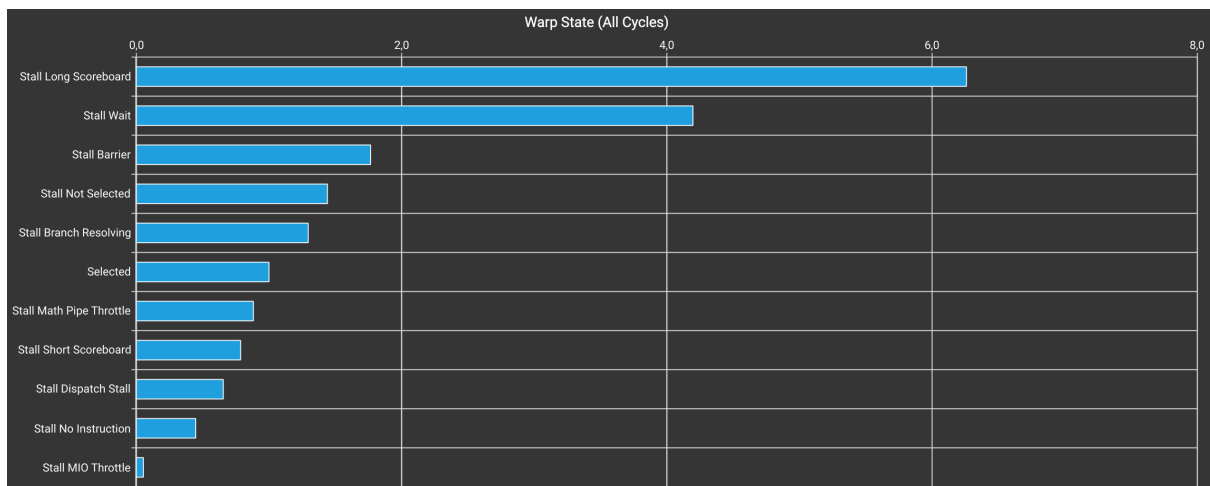


Figure 3.6: Warp Cycles per Issued Instruction for *bitonicSort* Kernel.

To address this issue and potentially achieve the estimated speedup, the suggested optimizations focus primarily on **increasing data locality** and leveraging **shared memory** for frequently accessed data.

Bitonic Merge

The impact of different block sizes on **warp occupancy** was evaluated. This analysis confirmed that, consistent with previous tests, a block size of **384 threads** is one of the configurations that maximizes **occupancy**, as shown in figure 3.7.

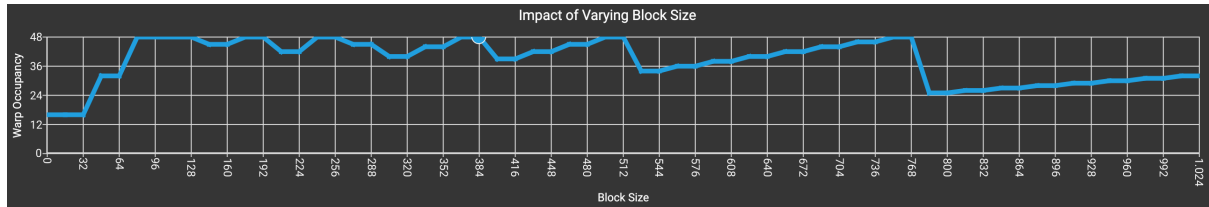


Figure 3.7: Warp occupancy with different block sizes

The **Source Counters** section of the **Nsight Compute** profiler provides metrics related to branch efficiency and warp stall reasons. Figure 3.8 summarizes the key statistics for the *bitonicMergeGlobal*.

The profiler identified an issue related to **Uncoalesced Global Accesses**, which contributes to a significant performance bottleneck. The tool estimates a potential **speedup of 58.49%** if this issue is addressed, as shown in Figure 3.8. The **branch efficiency** in this case is **84.04%**.

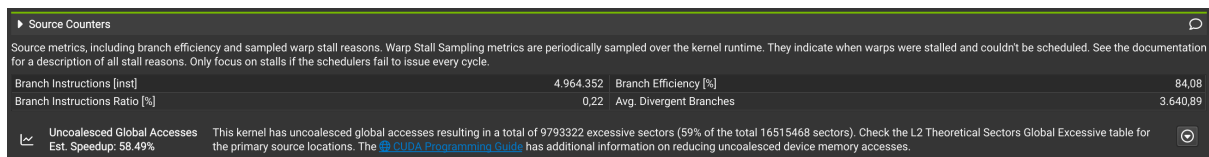


Figure 3.8: Source Counters for *bitonicMergeGlobal*.

The results from the **Warp State Statistics** section are shown in figure 3.9 and figure 3.10. The analysis indicates that, on average, each warp spends **100.3 cycles stalled** waiting for a **scoreboard dependency** on **L1TEX operations**. This accounts for approximately **86.0%** of the total average **116.6 cycles** between issued instructions. The stalls are predominantly due to unoptimized memory access patterns, as highlighted by the profiler.

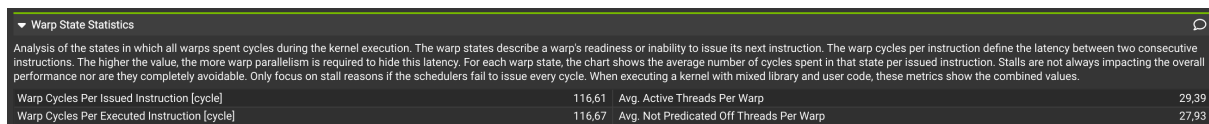


Figure 3.9: Warp State Statistics Header for *bitonicMergeGlobal*.

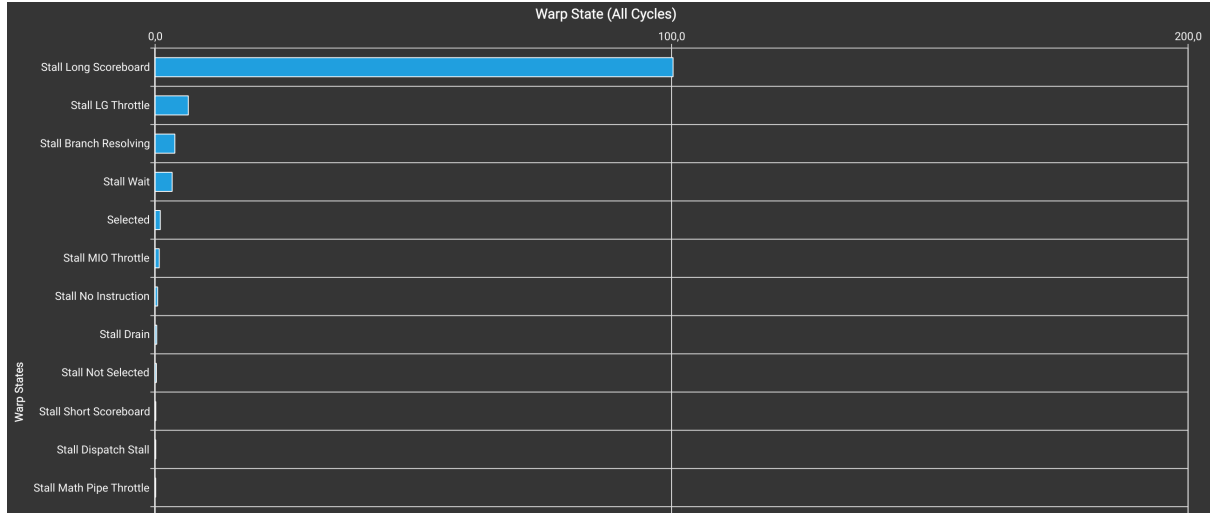


Figure 3.10: Warp State Statistics for *bitonicMergeGlobal*.

Both problems can be observed in the *compareExchange* function, that is the one that performs the most costly comparisons and LOAD/STORE operations. This can be observed in figure 3.11.

File	Line/Address	Marker	Source
compareExchange (5)			
Uncoalesced Global Accesses	3	50.00% of this line's global accesses are excessive.	LD.E R3, [R6, #64]
Uncoalesced Global Accesses	4	50.00% of this line's global accesses are excessive.	LD.E R0, [R4, #64]
Warp Stall	6	This line is responsible for 81.3% of all warp stalls. 99.9% of the stalls for this line are of type long_scoreboard.	ISETP.GE.U32.AND P2, PT, R0, R3, PT
Uncoalesced Global Accesses	14	68.90% of this line's global accesses are excessive.	ST.E [R4, #64], R3
Uncoalesced Global Accesses	15	68.90% of this line's global accesses are excessive.	ST.E [R6, #64], R0

Figure 3.11: *compareExchange* function problems overview

In summary, the profiling data pointed to significant stalls caused by unoptimized **memory access patterns** and insufficient **data locality**. The bitonic merge operations are performed directly on **global memory**, leading to potential **uncoalesced memory accesses**. This inefficiency arises because threads may access global memory in a non-contiguous manner, resulting in multiple memory transactions and increased latency. This first version does not leverage **shared memory** for intermediate computations, which is significantly faster than global memory. By not using shared memory, it missed opportunities for reducing global memory traffic and improving data locality. These issues will be addressed by the optimized version, explained in section 3.3.

3.3 Optimized version

This section provides an overview of the optimized GPU implementation of the Bitonic Sort algorithm.

3.3.1 Implementation

This version introduces several enhancements over Version 1 to improve performance, particularly by using shared memory and optimizing merging phases.

Use of Shared Memory

In Version 1, all operations, including sorting sub-blocks and performing merge phases, are executed directly on global memory. This leads to higher latency and less efficient memory access. In contrast, Version 2 leverages shared memory to store and sort data locally within each thread block. For instance, in the *bitonicSort* kernel, shared memory (`extern __shared__ uint32_t bitonicSortTile[]`) is allocated to hold sub-blocks of data. By copying data into shared memory before sorting, the number of slow global memory accesses is significantly reduced. This optimization is enabled by passing the size of shared memory as a parameter during kernel launch.

Global and Local Merging Phases

In Version 1, all merge phases are handled globally using the *bitonicMergeGlobal*, which operates exclusively on global memory. While this simplifies the implementation, it is inefficient for smaller data blocks. Version 2 introduces a distinction between **global merges** (for larger blocks) and **local merges** (for smaller blocks that fit in shared memory). A new kernel, *bitonicMergeLocal*, performs local merges in shared memory, allowing threads within a block to collaborate more efficiently. The local merge results are then written back to global memory. As can be seen from the code, in Version 1, each phase of the merging process iterates through all steps in a nested loop:

```
// Sort sub-blocks of input data using bitonic sort
runBitonicSort(d_values, arrayLength, sortOrder, false);
// Perform global bitonic merge
for (unsigned int phase = phasesBitonicSort + 1; phase <= phasesAll;
    phase++) {
    for (unsigned int step = phase; step >= 1; step--) {
        runBitonicMergeGlobal(d_values, arrayLength, phase, step,
                               sortOrder);
    }
}
```

Listing 3.1: Code snippet of the *bitonicSortV1* function

This approach redundantly accesses global memory during smaller merging steps. Version 2 optimizes this process by handling higher steps in each phase using the *bitonicMergeGlobal* and the final steps using the *bitonicMergeLocal*:

```
runBitonicSort(d_values, arrayLength, sortOrder, true);
for (unsigned int phase = phasesBitonicSort + 1; phase <= phasesAll;
    phase++) {
    unsigned int step = phase;
    while (step > phasesBitonicSort) {
        runBitonicMergeGlobal(d_values, arrayLength, phase, step,
                               sortOrder);
        step--;
    }
    runBitonicMergeLocal(d_values, arrayLength, phase, step,
                          sortOrder);
}
```

Listing 3.2: Code snippet of the *bitonicSortV2* function

This reduces global memory accesses in the final steps of each phase.

Dynamic Version Selection

Version 2 is only used when the required shared memory per thread block does not exceed the device's **MAX_SHARED_MEMORY_SIZE**. Otherwise, Version 1 serves as a fallback. This selection is implemented in the *bitonicSortParallel* function:

```
if (sharedMemoryUsage > MAX_SHARED_MEMORY_SIZE) {
    bitonicSortV1(d_values, arrayLength, sortOrder, phasesBitonicSort,
        phasesAll);
} else {
    bitonicSortV2(d_values, arrayLength, sortOrder, phasesBitonicSort,
        phasesAll);
}
```

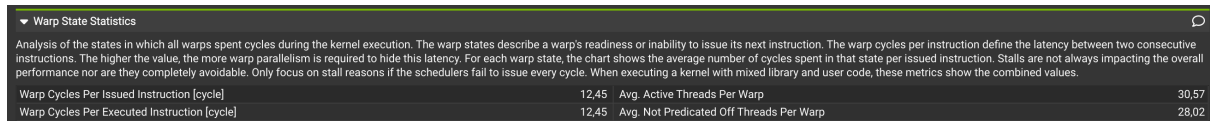
Listing 3.3: Code snippet of the *bitonicSortParallel* function

3.3.2 Performance Analysis

The optimized version of the code has been analyzed to evaluate whether the changes applied had an effect in solving the previously identified issues. The results for the *bitonicSort* and *bitonicMergeGlobal* kernels are discussed below.

Bitonic Sort

As shown in **Figure 3.12**, the number of **warp cycles per issued instruction** decreased significantly, from **18.78** in the first version to **12.45** in the optimized version. This improvement is primarily due to the absence of **Long Scoreboard Stalls**, which were present in the first version of the kernel.



▼ Warp State Statistics				
Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.				
Warp Cycles Per Issued Instruction [cycle]	12.45	Avg. Active Threads Per Warp		30.57
Warp Cycles Per Executed Instruction [cycle]	12.45	Avg. Not Predicated Off Threads Per Warp		28.02

Figure 3.12: Warp State Statistics Header for *bitonicSort*.

In the first version, **6.3 cycles per warp** were spent stalled on scoreboard dependencies related to **L1TEX operations**. These stalls have been completely eliminated in the optimized version, as can be seen in **Figure 3.13**, due to the use of **shared memory** in the optimized version of the kernel.

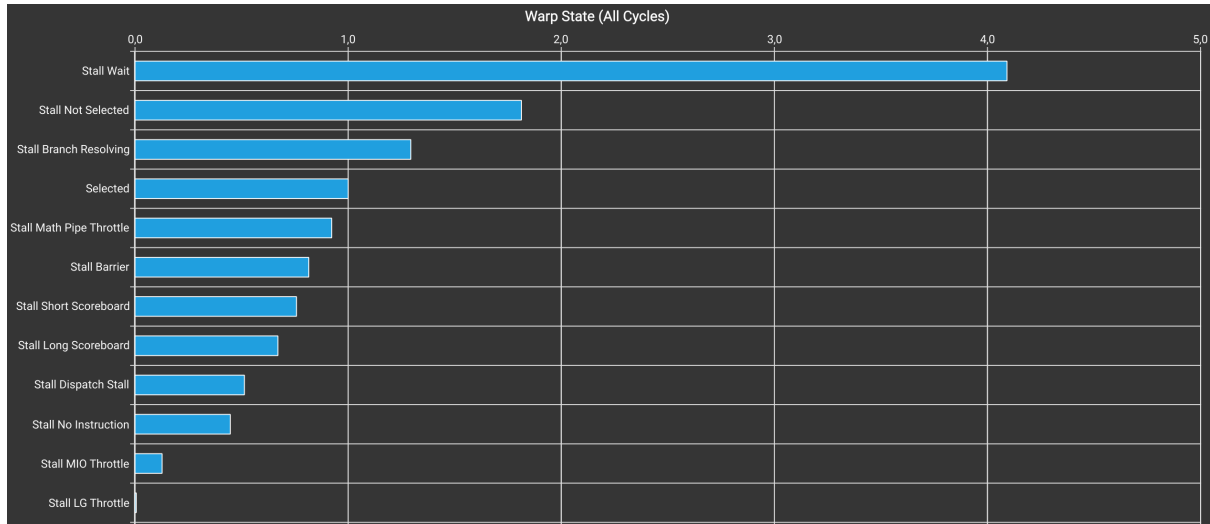


Figure 3.13: Warp State Statistics for *bitonicSort*.

Bitonic Merge

The performance of the *bitonicMergeGlobal* and *bitonicMergeLocal* was analyzed to evaluate the impact of the changes introduced in the optimized version. Figure 3.14 shows the **Source Counters** for the *bitonicMergeGlobal*. The branch efficiency increased from **84.08%** in the first version to **99.93%** in the optimized version. In the first version, there were cases where branch efficiency dropped significantly, indicating divergence issues. These have been effectively resolved in the optimized version.

Source Counters			
Source metrics, including branch efficiency and sampled warp stall reasons. Warp Stall Sampling metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.			
Branch Instructions [inst]	3,919,738	Branch Efficiency [%]	99.93
Branch Instructions Ratio [%]	0.16	Avg. Divergent Branches	13.76

Figure 3.14: Source Counters for *bitonicMergeGlobal*.

Figure 3.15 shows the **Source Counters** for the *bitonicMergeLocal*. The branch efficiency is slightly lower at **92.31%**, which is still an acceptable value, especially given the more complex branching logic required in the local merge phase. This kernel successfully leverages **shared memory**, which reduces global memory traffic and minimizes latency.

Source Counters			
Source metrics, including branch efficiency and sampled warp stall reasons. Warp Stall Sampling metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.			
Branch Instructions [inst]	58,647,162	Branch Efficiency [%]	92.31
Branch Instructions Ratio [%]	0.15	Avg. Divergent Branches	25,232.42

Figure 3.15: Source Counters for *bitonicMergeLocal*.

The **Warp State Statistics Header** for the *bitonicMergeLocal* (Figure 3.16) reports an average of **12.85 warp cycles per issued instruction**, significantly lower than the **116.6 cycles** in the first version of the global merge kernel. Furthermore, the kernel achieves **30.97 average active threads per warp**, an improvement over the **29.39 threads per warp** reported in the first version.

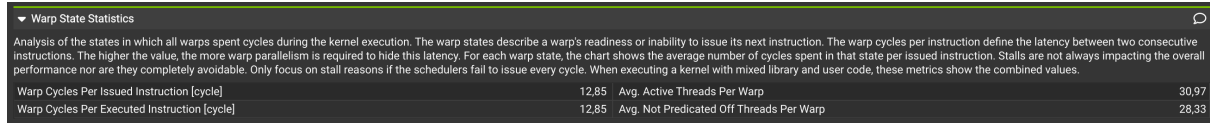


Figure 3.16: Warp State Statistics Header for *bitonicMergeLocal*.

As shown in Figure 3.17, the *bitonicMergeLocal* no longer exhibits **Long Scoreboard Stalls**. This is a significant improvement over the first version, where these stalls represented approximately **86.0% of the total cycles** in the global merge kernel. The absence of these stalls is attributed to the improved **memory access patterns** and the use of **shared memory** for local merge operations.

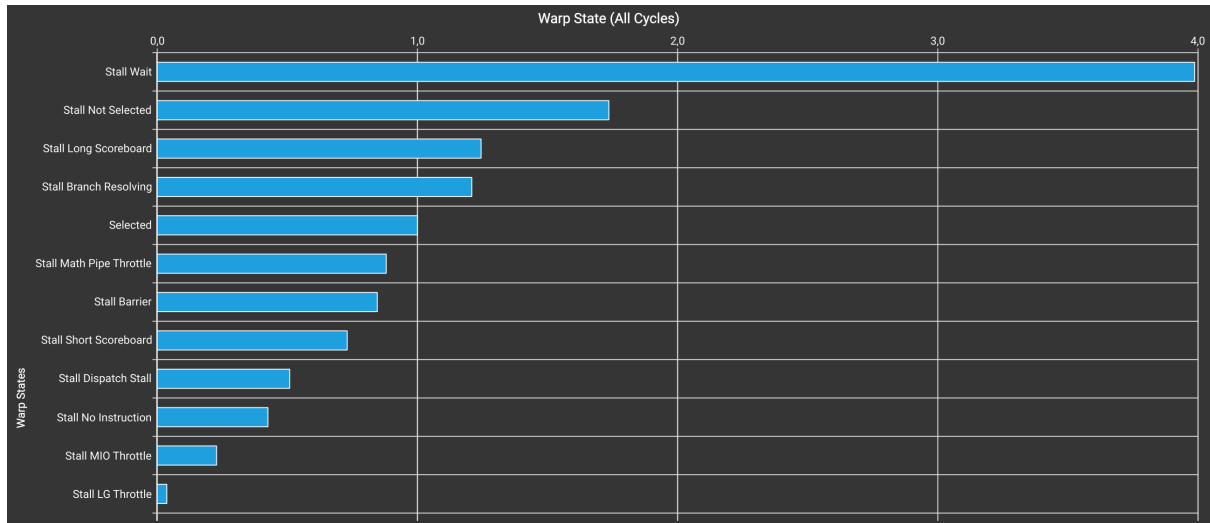


Figure 3.17: Warp State Statistics for *bitonicMergeLocal*.

In summary, the optimized version of the code effectively addresses the main issues identified in the first version by leveraging **shared memory** for local merge operations in the *bitonicMergeLocal* and for sorting in *bitonicSort*. This optimization significantly reduces reliance on **global memory**, thereby improving **data locality** and addressing the bottleneck caused by **uncoalesced memory accesses**. As a result, the **Long Scoreboard Stalls**, which accounted for a significant portion of the execution time in the first version, are eliminated in the optimized kernel.

Metrics such as the reduction in **warp cycles per issued instruction** and the increase in **average active threads per warp** directly highlight the impact of faster memory access and more efficient thread utilization. Furthermore, the improvements in **branch efficiency** reflect a reduction in thread divergence due to more predictable memory access patterns.

Overall, the use of **shared memory** not only reduces global memory latency but also ensures better warp efficiency and a more balanced workload, demonstrating a significant advancement in addressing the performance bottlenecks of the first version.

3.3.3 Tests and Results

This section focuses on the evaluation of the **optimized GPU implementation** and its improvements over the previous implementation and the CPU.

Speedup with Different Grid Sizes

The results for **speedup** with different grid configurations, now including GPU Version 2, are presented in Figure 3.18.

- **GPU Version 2** achieves significantly higher speedup, peaking at **700x** compared to the baseline configuration. This is more than twice the maximum speedup of **GPU Version 1**, which reaches approximately **300x**.
- The GPU Version 2 begins to outperform the first version starting from the **5th configuration**, which is the first that allows the optimization to be applied, since in this case there are enough blocks in the grid (4096) to exploit the shared memory.
- The optimal configuration for GPU Version 2 occurs at **384 threads per block** and **4096 blocks**, as it was in GPU Version 1.



Figure 3.18: Speedup with Different Grid Sizes

Throughput

Building on the earlier results, we include the performance of **GPU Version 2** for various array sizes, as shown in Figure 3.19. While GPU Version 1 already demonstrated better execution times compared to the CPU, GPU Version 2 shows significant improvements, particularly for **larger array sizes**.

In particular, the optimized version was able to reach the target throughput also for array sizes larger than **32 MB**, unlike the GPU Version 1.

These results demonstrate that the optimizations applied reduce execution time, ensure better **scaling** behavior compared to the first version and allow to meet the target **throughput** of **1 GB/s**. The mean throughput obtained by the optimized version is **1.4 GB/s**.

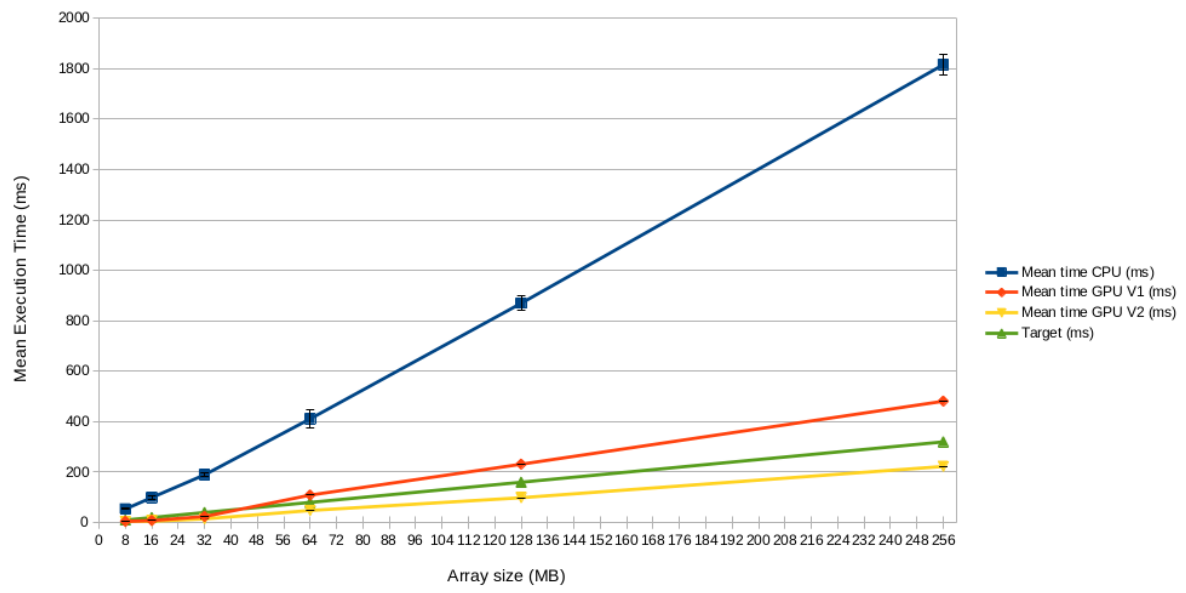


Figure 3.19: Mean Execution Time with Different Array Sizes