Bitonic Sort Using CUDA

Georgios Rousomanis (10703)
Aristotle University of Thessaloniki
Department of Electrical and Computer Engineering
email: rousoman@ece.auth.gr

August 2025

Introduction

Bitonic Sort is a comparison-based sorting algorithm with a regular structure that makes it well-suited for parallel execution on GPU architectures. Although its $O(n \log^2 n)$ time complexity is higher than that of other sorting algorithms, its predictable control flow and memory access patterns enable efficient implementation using CUDA. In this report, we present six progressively optimized versions of Bitonic Sort on the GPU. Each version introduces targeted optimizations, such as kernel fusion to reduce launch overhead, shared memory usage to accelerate intra-block sorting, warp-level shuffling to eliminate intra-warp synchronization, thread reduction to avoid idle execution, and memory access reordering to eliminate bank conflicts in shared memory. Table 1 provides a concise overview of each version's primary features.

Version	**Description**
V0	Naive baseline: one comparison per thread; global memory only; one kernel per sorting step.
V1	Loops over multiple comparison steps in a single kernel; reduces kernel launch overhead.
V2	Uses shared memory for intra-block sorting; improves memory access latency.
V3	Applies warp-level shuffles for intra-warp comparisons; reduces use of shared memory and barriers.
V4	Reduces thread count by half; removes in-warp branching and non-participating threads.
V5	Eliminates shared memory bank conflicts via in-warp index shuffling.

Table 1: Summary of GPU Bitonic Sort versions and their descriptions.

V0 Naive Bitonic Sort

This version represents the most basic GPU implementation of the Bitonic Sort algorithm. The key goal of this version is correctness and a functional mapping of the bitonic sorting network onto the GPU. Each compare-and-swap operation in the sorting network is implemented as a separate kernel launch. All data resides in global memory, and no attempt is made to share or reuse data between threads. While simple and easy to debug, this approach is inefficient due to frequent kernel invocations and the lack of memory locality. This version serves as the baseline for evaluating future optimizations.

The GPU kernel performs the basic compare-and-swap step of the bitonic sorting network. Each thread operates on single index and computes its partner j using the bitonic sort logic ($j = i \oplus step$). Only one of the two indices (the smaller one) performs the swap to avoid duplicate work. The direction of the comparison (ascending or descending) depends on the current size of the bitonic sequence and the index of the thread.

Algorithm 1 Compare-And-Swap Kernel (v0)

procedure CompareAndSwapKernelV0(*data*, n, *size*, *step*)
  $i \leftarrow$ global thread index
  if $i \geq n$ **then return**
  end if
  $j \leftarrow i \oplus step$
  if $j > i$ **then**
    $ascending \leftarrow (i \;\&\; size) == 0$
    CompareAndSwap(*data*, i, j, *ascending*)
  end if
end procedure

The host code runs two nested loops: the outer loop iterates over the sizes of the bitonic sequences, and the inner loop performs the necessary merge steps by repeatedly launching the device kernel. Reversing at the end allows the core logic to always sort in ascending order, simplifying the kernel logic. Because every operation interacts with global memory, performance is limited – especially due to the overhead of many kernel launches and the lack of shared memory usage.

Algorithm 2 Bitonic Sort Host (v0)

procedure BitonicSortV0(*host_data*, n, *descending*)
  Allocate *device_data* on GPU
  Copy *host_data* to *device_data*
  for $size = 2$ to n by $\times 2$ **do**
    for $step = size/2$ down to 1 by $\div 2$ **do**
      Launch CompareAndSwapKernelV0(*device_data*, n, *size*, *step*)
      Synchronize and check for errors
    end for
  end for
  if *descending* **then**
    Launch ReverseKernel(*device_data*, n)
    Synchronize and check for errors
  end if
  Copy *device_data* back to *host_data*
  Free GPU memory
end procedure

V1 Fused Intra-Block Bitonic Sort

While V0 provided a correct but naive GPU implementation of Bitonic Sort, its primary bottleneck was the excessive number of kernel launches and exclusive use of global memory. Each compare-and-swap operation required a separate kernel call, leading to high overhead due to frequent synchronization and poor memory locality.

V1 introduces a key optimization: it *reduces the number of kernel launches during the initial sorting phases* by fusing all intra-block compare-and-swap operations into two persistent kernels. The first kernel performs a complete bitonic sort for each block independently, using global memory and relying on intra-block synchronization (`__syncthreads()`) to coordinate comparisons and swaps between thread pairs. The second kernel is used in later stages to refine the order within each block during global merging phases. These kernels eliminate the need to launch a separate kernel for each individual sorting step as done in V0. Inter-block merging still relies on V0's global kernel, but only for steps that cannot be handled within a single block, striking a balance between flexibility and performance.

Algorithm 3 Intra-Block Sort Kernel (v1)

```
1: procedure IntraBlockSortKernelV1(data, n, chunk_size)
2:     i ← global thread index
3:     if i ≥ n then return
4:     end if
5:     for size = 2 to chunk_size by ×2 do
6:         ascending ← (i & size) == 0
7:         for step = size/2 down to 1 by ÷2 do
8:             j ← i ⊕ step
9:             if j > i then
10:                CompareAndSwap(data, i, j, ascending)
11:            end if
12:            ThreadSync
13:        end for
14:    end for
15: end procedure
```

The `IntraBlockSortKernelV1` performs a complete bitonic sort independently within each block, leveraging global memory and intra-block synchronization. The outer loop iterates over increasing bitonic sequence sizes, doubling from 2 up to the given `chunk_size`. For each size, the thread determines its local sort direction based on whether the i-th index lies in the ascending or descending half. The inner loop implements the bitonic merge steps by computing the partner index $j = i \oplus step$ and conditionally calling `CompareAndSwap` if $j > i$. After each merge step, `__syncthreads()` ensures all threads are synchronized before proceeding to the next stage. This kernel eliminates the need for launching a new kernel at every compare-and-swap step by using a nested loop structure with synchronization at each level.

Algorithm 4 Intra-Block Refinement Kernel (v1)

```
1: procedure IntraBlockRefineKernelV1(data, n, size, chunk_size)
2:     i ← global thread index
3:     if i ≥ n then return
4:     end if
5:     ascending ← (i & size) == 0
6:     for step = chunk_size/2 down to 1 by ÷2 do
7:         j ← i ⊕ step
8:         if j > i then
9:             CompareAndSwap(data, i, j, ascending)
10:        end if
11:        ThreadSync
12:    end for
13: end procedure
```

The `IntraBlockRefineKernelV1` is used during the global merging phases to refine the order within each block. This kernel assumes that the merging logic for large sequences has already been applied across blocks, and now the residual intra-block portion needs to be sorted correctly. The inner loop follows the bitonic merge pattern but is restricted to steps down to `chunk_size`/2, ensuring that refinement remains within block boundaries. As in the initial sort kernel, the thread computes its partner index $j = i \oplus step$, conditionally performs a `CompareAndSwap`, and synchronizes all threads after each step using `__syncthreads()`. This kernel avoids launching multiple V0-style global kernels for each refinement step and provides efficient intra-block correction during the merge phase.

Algorithm 5 Bitonic Sort Host (v1)

procedure BitonicSortV1(*host_data*, n, *descending*)
  Allocate *device_data* on GPU
  Copy *host_data* to *device_data*
  $chunk\_size \leftarrow \min(n, \text{BLOCK\_SIZE})$
  $max\_step \leftarrow chunk\_size/2$
  Launch IntraBlockSortKernelV1(*device_data*, n, *chunk_size*) ▷ Initial sort inside blocks
  Synchronize and check for errors
  for $size = 2 \cdot chunk\_size$ to n by $\times 2$ **do**
    for $step = size/2$ down to $max\_step + 1$ by $\div 2$ **do**
      Launch CompareAndSwapKernelV0(*device_data*, n, *size*, *step*) ▷ Merge across blocks
      Synchronize and check for errors
    end for
    Launch IntraBlockRefineKernelV1(*device_data*, n, *size*, *chunk_size*) ▷ Refine within each block
    Synchronize and check for errors
  end for
  if *descending* **then**
    Launch ReverseKernel(*device_data*, n)
    Synchronize and check for errors
  end if
  Copy *device_data* back to *host_data*
  Free GPU memory
end procedure

The `BitonicSortV1` host function orchestrates the overall sorting procedure by combining efficient intra-block sorting with legacy inter-block merging. The `chunk_size`, typically set to the block size, determines how much work each block handles independently. The first kernel launch performs a full bitonic sort within each block using `IntraBlockSortKernelV1`, leveraging intra-block synchronization to sort local data. For larger merge sizes beyond the block scope, it performs the global merging stages by launching the original `CompareAndSwapKernelV0` for each step beyond the block's range. After global merges, each block locally refines its portion by launching `IntraBlockRefineKernelV1`. This process is repeated until the full array is sorted. If the desired result is descending, a final kernel reverses the data.

V2 Using Shared Memory for Intra-Block Sorting

Version 2 improves upon the previous version by leveraging shared memory for intra-block sorting. While version 1 performed intra-block sorting entirely in global memory, causing many slow global memory accesses, this version loads each block's data chunk into shared memory before sorting. This optimization significantly reduces global memory traffic and exploits the faster shared memory, leading to better performance within each block. The inter-block merging remains the same as in v1.

The updated intra-block sort kernel, which now utilizes shared memory for better performance, is shown below. The same logic applies to the intra-block refinement kernel. The host-side code remains unchanged, and inter-block merging is still handled using the same global memory-based kernel from the previous version.

Algorithm 6 Intra-Block Sort Kernel using shared memory (v2)

procedure IntraBlockSortKernelV2($data$, n, $chunk\_size$)
  $idx \leftarrow$ global thread index
  if $idx \geq n$ **then return**
  end if
  Declare shared memory array s_data of size $chunk\_size$
  $offset \leftarrow$ block index $\times$ block size
  $tid \leftarrow$ thread index within block
  $s\_data[tid] \leftarrow data[offset + tid]$ ▷ Load data chunk from global memory into shared memory
  Synchronize threads
  for $size \leftarrow 2$ **to** $chunk\_size$, doubling each iteration **do**
    $is\_asc \leftarrow (offset + tid) \& size == 0$
    for $step \leftarrow size/2$ **down to** 1, by $\div 2$ **do**
      $j \leftarrow tid \oplus step$
      if $j > tid$ **then**
        Compare and swap $s\_data[tid]$ and $s\_data[j]$ in order is_asc
      end if
      Synchronize threads
    end for
  end for
  $data[offset + tid] \leftarrow s\_data[tid]$ ▷ Copy sorted chunk back to global memory
end procedure

This kernel improves memory access efficiency by first loading a block-aligned chunk of data from global memory into faster shared memory before performing the bitonic sort. Each thread handles a single element, with its location computed using the offset `offset = blockIdx.x * blockDim.x`. As established in V1, the block size is equal to the chunk size, ensuring that each block operates on a distinct segment of the input array.

The global thread ID, derived from the offset, determines the comparison direction based on the standard bitonic sorting rule. The full bitonic sorting network is then executed entirely in shared memory using nested loops and the same bitwise logic introduced in V1. Synchronization after each step guarantees correctness and prevents race conditions before the sorted data is written back to global memory.

It is also important to note that we did not apply shared memory optimization to the inter-block merging phase. Since each kernel launch in this stage performs a single compare-and-swap between non-reused elements, there is no locality benefit to justify shared memory usage. As such, inter-block merging remains unchanged and continues to operate directly on global memory.

V3 Warp-Level Intra-Block Bitonic Sort

In Version 2, intra-block sorting was performed entirely using shared memory and explicit synchronization. However, this approach becomes inefficient for small strides (`step` < 32), where thread divergence and memory access latency are more noticeable. In V3, we optimize the fine-grained stages of the intra-block sort by leveraging warp shuffle instructions (`__shfl_xor_sync`) for intra-warp exchanges. This warp-level intrinsic enables direct register-to-register data exchange without shared memory or explicit synchronization, resulting in faster and more efficient sorting within warps.

Algorithm 7 Intra-Block Sort with Warp-Level Bitonic Sorting (v3)

```
1: procedure IntraBlockSortKernelV3(data, n, max_size)
2:     idx ← global thread index
3:     if idx ≥ n then return
4:     end if
5:     tid ← thread index in block
6:     offset ← block index × block size
7:     shared s_data[max_size]
8:     s_data[tid] ← data[offset + tid]
9:     SyncThreads
10:    global_id ← offset + tid
11:    for size = 2 to max_size by ×2 do
12:        ascending ← (global_id & size) == 0
13:        step ← size/2
14:        while step ≥ 32 do
15:            j ← tid ⊕ step
16:            if j > tid then
17:                CompareAndSwap(s_data, tid, j, ascending)
18:            end if
19:            SyncThreads
20:            step ← step/2
21:        end while
22:        val ← s_data[tid]
23:        while step > 0 do
24:            j ← tid ⊕ step
25:            partner ← ShflXor(val, step)
26:            correct_order ← (tid < j) == (val ≤ partner)
27:            swap ← ascending ≠ correct_order
28:            val ← if swap then partner else val
29:            step ← step/2
30:        end while
31:        s_data[tid] ← val
32:        SyncThreads
33:    end for
34:    data[offset + tid] ← s_data[tid]
35: end procedure
```

This kernel performs full intra-block bitonic sorting using both shared memory and warp-level primitives. For each block, threads first load their corresponding data into shared memory. The sorting process is divided into two phases:

1. For strides larger than or equal to 32, threads use shared memory and synchronize at each step.
2. For smaller strides, shuffle instructions (`__shfl_xor_sync`) are used to compare and exchange values directly in registers within a warp.

This avoids unnecessary memory access and synchronization. The final sorted result is written back to global memory by each thread.

Notice that in the in-warp sorting phase, the conditional branching of the compare-and-swap logic is removed. Instead, both partners involved in an exchange evaluate whether their pair is in the correct order and perform the swap if necessary. This means the same operation is effectively executed twice – once by each thread in the pair – but without introducing performance overhead. Since threads within a warp execute in SIMD fashion, both threads see consistent values and perform the same sequence of operations simultaneously. This approach avoids branch divergence, which can serialize thread execution within a warp and harm performance. Furthermore, race conditions on register values are avoided because all threads within the warp execute the same instruction synchronously; when a thread reads the partner's value via `__shfl_xor_sync`, its partner is performing the exact same operation in reverse, ensuring correctness.

V4 Reducing Thread Overhead and Eliminating Branching

In previous versions, each element in the input array was assigned a thread, but only half of those threads actively participated in each compare-and-swap step. The other half would simply branch out early. This caused overhead due to unnecessary thread launches and control flow divergence (in-warp serialization), which harmed warp efficiency. In Version 4, we launch only the necessary number of threads – i.e., half the data size. All launched threads are guaranteed to perform useful work, thereby eliminating branching and improving performance. This makes the implementation more efficient in both thread utilization and warp execution.

Algorithm 8 Intra-Block Bitonic Sort Kernel (V4)

```
1:  procedure IntraBlockSortKernelV4(data, n, maxSize)
2:      tid ← thread's local index
3:      globalIdx ← block index × blockDim.x + tid
4:      if globalIdx ≥ n then return
5:      end if
6:      offset ← block index × blockDim.x ×2
7:      Allocate shared array sdata[2 × blockDim.x]
8:      i ← 2 × tid
9:      sdata[i] ← data[offset + i]
10:     sdata[i + 1] ← data[offset + i + 1]
11:     Syncthreads
12:     for size = 2 to maxSize step ×2 do
13:         for step = size/2 to 1 step /2 do
14:             log2step ← log2(step)
15:             i ← GetLowerPartner(tid, step, log2step)
16:             j ← i + step
17:             asc ← ((i & size) == 0)
18:             if CompareNeeded(sdata[i], sdata[j], asc) then
19:                 Swap(sdata[i], sdata[j])
20:             end if
21:             Syncthreads
22:         end for
23:     end for
24:     data[offset + 2 × tid] ← sdata[2 × tid]
25:     data[offset + 2 × tid + 1] ← sdata[2 × tid + 1]
26: end procedure
```

This kernel processes twice as much data per thread by loading two elements into shared memory. The key improvement is that only half as many threads are launched – all of which perform meaningful compare-and-swap work. This avoids the in-warp divergence seen in earlier versions where half the threads would skip execution due to conditional logic. The nested loops inside shared memory implement the full intra-block bitonic sort pattern using local synchronization after each step.

Notice that in the above logic, the partner of element i (computed by `get_lower_partner`) is derived using the expression $j = i + step$ and not via the XOR operator, because i is already computed as the lower index of a compare-and-swap pair. This formulation avoids ambiguity and ensures correct pairing within bitonic segments.

By using `get_lower_partner`, we compute the lower index of each bitonic pair, ensuring that all threads coordinate comparisons without overlap or branching. After sorting in shared memory, the threads write back their two elements to global memory.

Understanding `get_lower_partner`

The `get_lower_partner` function computes the lower index in a pair of elements involved in a bitonic comparison during sorting. It ensures consistent partner indexing across both shared and global memory versions of the algorithm, while preserving the structure of the bitonic sorting network.

Inputs:

- `tid`: The thread's local index.
- `step`: The stride of the current compare-and-swap phase (i.e., distance between paired elements).

- `log2step`: The base-2 logarithm of `step`, computed as `__ffs(step) - 1`.

Computation Breakdown:

- `blockPair = tid >> log2step`: Computes which bitonic block pair the thread belongs to by shifting right by `log2step` bits. This groups threads into regions of size `2 × step`.
- `offset = tid & (step - 1)`: Extracts the relative offset of the thread within its block of `step` elements. This is the thread's local position in its group.
- `base = blockPair << (log2step + 1)`: Calculates the starting index of the 2×`step` bitonic group in shared memory.
- `return base + offset`: Adds the offset to the base, yielding the lower partner index.

Example: If `tid = 11`, `step = 4`, then `log2step = 2`.

- `blockPair = 11 >> 2 = 2`
- `offset = 11 & 3 = 3`
- `base = 2 << 3 = 16`
- `return = 16 + 3 = 19`

Thus, thread 11 is mapped to the lower index 19 within the comparison pair (19, 23) for that stage of sorting. This formulation ensures that every pair is handled exactly once without duplication or divergence.

V5 Eliminating 2-Way Shared Memory Bank Conflicts

In Version 4, the intra-block sorting kernels improve thread utilization by assigning each thread to handle a pair of consecutive elements. Specifically, thread t loads elements at indices $2t$ and $2t + 1$ from global memory into shared memory. While this strategy reduces the number of threads by half, it unintentionally introduces a *2-way shared memory bank conflict*.

CUDA shared memory is divided into multiple banks (typically 32). When multiple threads in the *same warp* access different addresses that map to the same memory bank, their accesses are serialized, resulting in a performance penalty. This phenomenon is known as a *bank conflict*.

In V4, for example, thread 0 loads elements 0 and 1, thread 1 loads elements 2 and 3, and so on. When thread 16 loads elements 32 and 33, it accesses the same banks as thread 0 (due to the modulo-based bank mapping), resulting in two simultaneous accesses to banks 0 and 1. Since both threads are part of the same warp and execute in SIMD fashion, these accesses serialize, degrading performance.

Solution: Index Shuffling Trick

Version 5 eliminates this conflict by reordering how data is loaded into and stored from shared memory using a lightweight bitwise trick. The goal is to stagger the access pattern so that consecutive threads no longer access adjacent banks in the same order.

- First, each thread computes whether it belongs to the **lower** or **upper half** of its warp:

```
int t = threadIdx.x;
int t2 = 2 * t;
constexpr int HALF_WARP_SIZE = WARP_SIZE >> 1;
int is_in_lower_half = (t & HALF_WARP_SIZE) == 0;
int is_in_upper_half = !is_in_lower_half;
```

- When loading data from global to shared memory, the two values handled by each thread are accessed in a swapped order depending on its warp-half status:

```
// Load data into shared memory
s_data[t2 + is_in_upper_half] = data[offset + t2 + is_in_upper_half];
s_data[t2 + is_in_lower_half] = data[offset + t2 + is_in_lower_half];
```

- The same reordering is applied during the write-back to global memory:

```
// Copy data back to global memory
data[offset + t2 + is_in_upper_half] = s_data[t2 + is_in_upper_half];
data[offset + t2 + is_in_lower_half] = s_data[t2 + is_in_lower_half];
```

This indexing trick ensures that if thread 0 accesses element 0 (mapped to bank 0), then thread 16 accesses element 1 (mapped to bank 1), and vice versa. This rearrangement prevents two threads in the same warp from simultaneously accessing addresses that fall into the same memory bank, effectively eliminating the 2-way bank conflict.

Note: Bank conflicts are relevant only among threads of the same warp, since warps execute instructions in lockstep. Threads from different warps are not executed in SIMD Fashion and hence they do not interfere with each other's shared memory accesses.

Performance Comparison

To evaluate the effectiveness of our six Bitonic Sort implementations, we benchmarked them on the Aristotelis High-Performance Computing (HPC) Cluster using two distinct GPU partitions: the `gpu` partition with a Pascal-based Tesla P100, and the `ampere` partition featuring the more advanced Ampere-based NVIDIA A100. Table 2 summarizes the key hardware specifications of each GPU.

Partition	GPU Model	Architecture	Memory	Bandwidth	CUDA Capability
`gpu`	Tesla P100-PCIE-12GB	Pascal	12 GB	549 GB/s	6.0
`ampere`	NVIDIA A100-SXM4-40GB	Ampere	40 GB	1,555 GB/s	8.0

Table 2: GPU specifications of the Aristotelis HPC Cluster.

Tables 3 and 4 present the execution times and relative speedups for sorting an input of 2^{29} integers on both GPUs. The baseline corresponds to a serial CPU implementation.

On both GPUs, Version V0 provides the largest initial gain by offloading computation to the GPU, achieving a 24.74× speedup on the P100 and a 44.05× speedup on the A100. Each subsequent version builds upon the previous one with progressively more refined optimizations. Version V5 yields the highest overall speedup: 42.63× on the P100 and 60.69× on the A100, relative to the serial baseline.

Despite the A100's higher memory bandwidth and more modern architecture, the relative speedup progression is more pronounced on the P100. This is because the A100's high absolute performance reduces the margin for gains from each optimization. Notably, the speedup from V4 to V5 is marginal on both GPUs, as the scheduler effectively hides the latency introduced by shared memory bank conflicts, making our optimization less impactful in practice.

Version	Time (ms)	Step Speedup	Cumulative Speedup
SERIAL	110711.370	–	1.00
V0	4475.873	24.74	24.74
V1	3967.719	1.13	27.90
V2	3220.962	1.23	34.37
V3	3028.741	1.06	36.55
V4	2627.175	1.15	42.14
V5	2596.559	1.01	42.63

Table 3: Execution time and speedup per version for sorting 2^{29} integers on the `gpu` partition (Tesla P100).

Version	Time (ms)	Step Speedup	Cumulative Speedup
SERIAL	93705.605	–	1.00
V0	2127.299	44.05	44.05
V1	1769.154	1.20	52.97
V2	1672.394	1.06	56.03
V3	1644.309	1.02	56.99
V4	1562.459	1.05	59.97
V5	1543.992	1.01	60.69

Table 4: Execution time and speedup per version for sorting 2^{29} integers on the `ampere` partition (NVIDIA A100).

Figures 1 and 2 show the execution times for all Bitonic Sort versions across input sizes ranging from 2^{15} to 2^{29}. For small arrays ($n < 2^{20}$), GPU execution times fluctuate due to constant host-side overhead, which dominates the total runtime. As the input size grows, this overhead becomes negligible and the optimized device-side computation leads to stable and improved performance across versions.

On the A100, these early fluctuations are even more pronounced because the kernels complete faster, making the host-side launch overhead a relatively larger component of the total runtime.

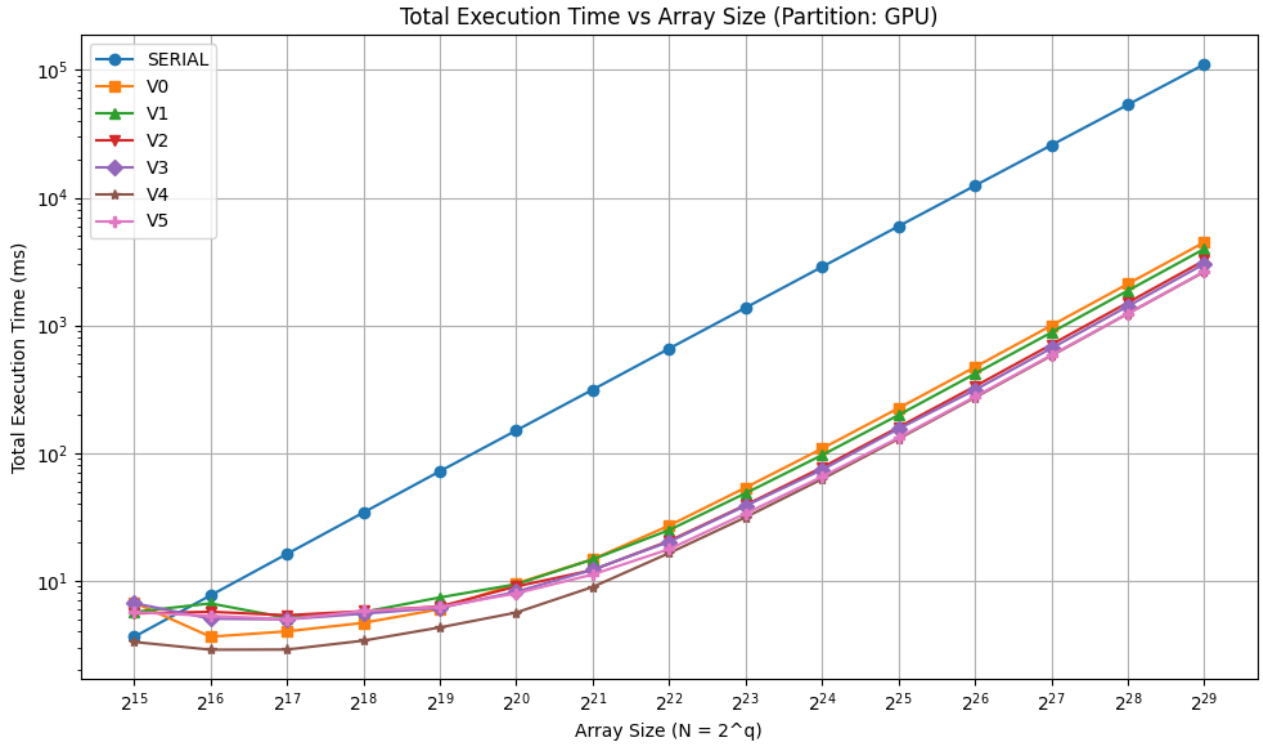

Figure 1: Execution time of each version for input sizes 2^{15} to 2^{29} on the `gpu` partition (Tesla P100).

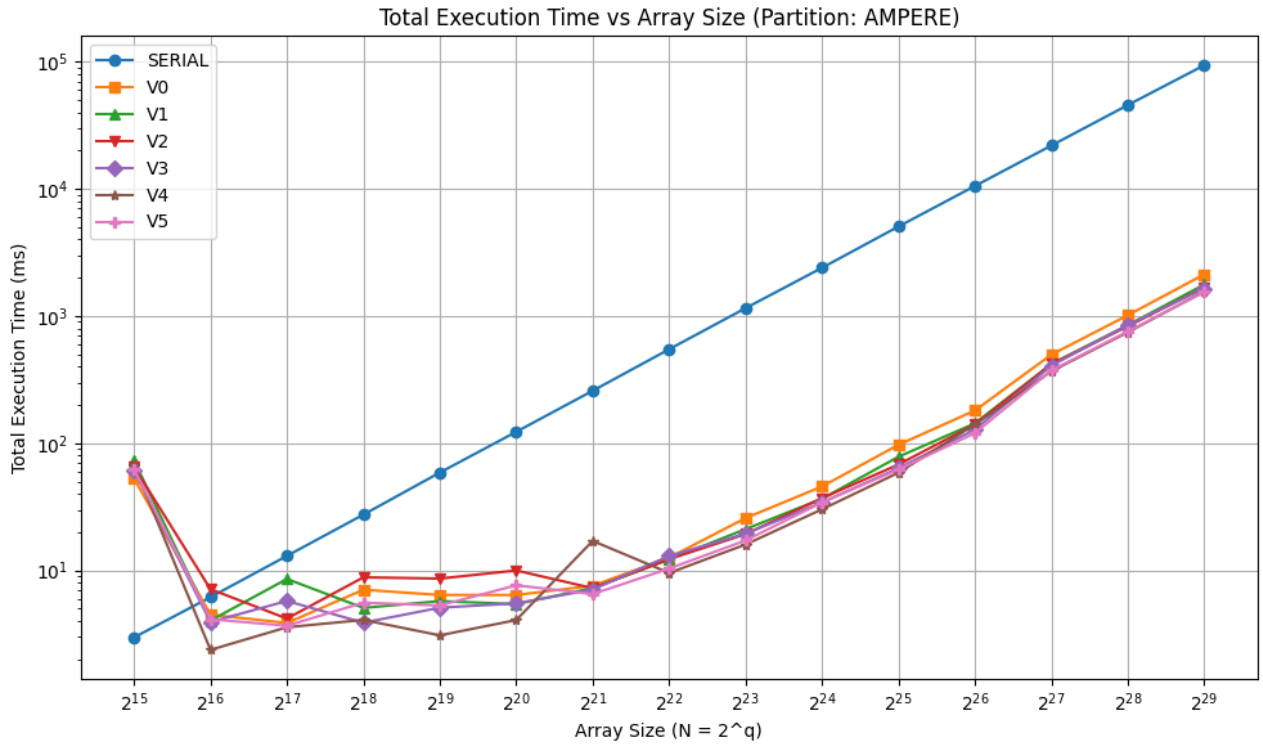

Figure 2: Execution time of each version for input sizes 2^{15} to 2^{29} on the `ampere` partition (NVIDIA A100).

Conclusions & Future Work

In this report, we presented a series of CUDA implementations of the Bitonic Sort algorithm, each introducing targeted improvements over the previous version. We began with a naive global-memory-based implementation and progressively enhanced it by first fusing multiple sorting steps into a single kernel to reduce launch overhead. We then introduced shared memory to accelerate intra-block sorting and minimize global memory traffic. Next, we applied warp-level shuffle operations to reduce synchronization costs for intra-warp comparisons. To further improve efficiency, we halved the number of active threads, eliminating unnecessary branching within warps. Finally, we removed shared memory bank conflicts using an indexing trick, enabling optimal memory throughput.

While Bitonic Sort is well-suited for parallel architectures, it is not asymptotically optimal for large-scale sorting due to its $O(n \log^2 n)$ complexity. In future work, we plan to implement and evaluate an even faster algorithm: *Radix Sort*. As a non-comparison-based method with linear time complexity for fixed-size keys, Radix Sort is capable of outperforming Bitonic Sort on modern GPUs, particularly for large datasets. Optimizing Radix Sort using warp-level operations, memory coalescing, and bucket-based partitioning will be a promising next step in our exploration of high-performance GPU sorting algorithms.

Acknowledgments

Performance measurements were conducted on the Aristotelis High Performance Computing Cluster at the Aristotle University of Thessaloniki.

References

[1] M. Garland, *An Even Easier Introduction to CUDA*, NVIDIA Developer Blog, 2008. [Online]. Available: `https://developer.nvidia.com/blog/even-easier-introduction-cuda/`

[2] M. Harris, *Optimizing Parallel Reduction in CUDA*, NVIDIA Corporation, 2007. [Online]. Available: `https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf`

[3] G. Rousomanis, *Bitonic Sort with CUDA - Source Code Repository*, GitHub, 2025. [Online]. Available: `https://github.com/georrous6/bitonic-sort-cuda`