Bitonic Sort on GPU
CUDA Implementations & Optimizations

February 2025

Author: Aristeidis Daskalopoulos
Github Repo: https://github.com/arisdask/BitonicSortCUDA
A Parallel & Distributed Systems' Project

1 Intro

Problem Introduction

The primary objective of this project is to implement the Bitonic Sort algorithm using CUDA. This work presents three versions of the implementation, each serving as an optimization of the previous one. The Bitonic Sort algorithm is particularly well-suited for GPU execution due to its regular comparison patterns and parallelizable structure. The three versions progressively optimize the algorithm by reducing synchronization overhead and minimizing global memory accesses for improved performance.

Versions Implemented

- **V0 (Baseline Implementation)**:
  The initial version (V0) is a direct translation of the Bitonic Sort algorithm to CUDA, where each step of the algorithm is executed in a separate kernel function call. While simple to implement, V0 suffers from significant performance bottlenecks caused by the large number of kernel launches and inefficient (global) memory access, requiring explicit synchronization between steps, which further increases overhead.
- **V1 (Kernel Optimization)**:
  The second version (V1) optimizes V0 by fusing multiple steps of the Bitonic Sort algorithm into a single kernel. This reduces the number of kernel launches and minimizes the global synchronization overhead. This version is more efficient than V0, but it remains limited by global memory bandwidth, as *all data exchanges still occur in global memory.*
- **V2 (Shared Memory Optimization)**:
  The third version (V2) introduces shared memory to further optimize the algorithm. By leveraging the GPU's faster on-chip memory, V2 minimizes global memory accesses by performing intermediate comparisons and exchanges in shared memory. This allows multiple comparison steps to be completed before writing results back to global memory, achieving the best performance among the three versions.

2 Theoretical Algorithm Analysis

In this section, we introduce the bitonic sort algorithm that has been implemented. We explain its theoretical foundation, detailing how it functions step-by-step. Following this, we calculate its complexity and identify areas for optimization.

2.1 First Algorithmic Approach (V0)

The Bitonic Sort algorithm consists of multiple **stages** and **steps**. For an array of size $N = 2^k$, the algorithm requires $\log_2 N$ stages, and each stage $s \in \{0, \ldots, (\log_2 N) - 1\}$ consists of $s + 1$ steps. At each step, pairs of elements are *compared* and *swapped* based on their positions in the bitonic sequence.

Algorithm 1 Bitonic Sort V0

```
Require: Input: Array data of size $N = 2^k$
Ensure: Output: Sorted array in ascending order
1: stages ← $\log_2 N$                                          ▷ Total number of stages
2: for stage = 0 to stages − 1 do
3:     for step = stage downto 0 do
4:         Launch CUDA kernel: bitonic_kernel_v0—parameters: $(\text{data}, N, \text{stage}, \text{step})$
5:         Global Sync all threads                       ▷ Ensure all threads complete the step
6:     end for
7: end for
```

The `bitonic_kernel_v0` is responsible for executing a single step of the Bitonic Sort algorithm. Here is the detailed algorithm of this kernel:

Algorithm 2 bitonic_kernel_v0

```
Require: Input: data: Pointer to the array in global memory — length: Length of
    the array — stage: Current stage of the Bitonic Sort — step: Current step within
    the stage
Ensure: Output: Partially "sorted" array after the current step (creates a bitonic se-
    quence)
1: tid ← blockIdx.x × blockDim.x + threadIdx.x              ▷ Compute global thread ID
2: if tid $\geq$ (length ÷ 2) then
3:     return                                             ▷ Exit if thread is out of bounds
4: end if
5: idx ← GET_ARR_ID(tid, step)                  ▷ Element index that this thread will handle
6: partner ← idx $\oplus$ (1 $\ll$ step)              ▷ Partner idx that this thread will handle (XOR)
7: if idx < partner $\wedge$ partner < length then
8:     exchange(data, idx, idx, partner, stage)                ▷ Compare and swap elements
9: end if
```

2.2 Explaining the Algorithm

Here, we provide a detailed explanation of how the above algorithm works. We describe the key stages of the algorithm, the role of the kernel function, and how `GET_ARR_ID` & data exchanges occur to achieve the final sorted outcome.

2.2.1 Stages and Steps Overview

For each stage s (from 0 to $\log_2 N - 1$):

- For each step t (from s down to 0):
  - Launch the `bitonic_kernel_v0` kernel.
  - Each thread computes its element index using `GET_ARR_ID` and its partner index using XOR.
  - The thread compares its element with its partner and swaps them if necessary.
  - A global synchronization barrier ensures all threads complete the current step before proceeding.

About Global Sync

This synchronization is necessary to maintain the correctness of the Bitonic Sort algorithm, as *each step depends on the results of the previous step.* This means that after launching a kernel, the CPU must wait for the GPU to complete the kernel execution before launching the next kernel.
However, we should keep in mind that frequent "global" synchronization (between all threads) adds overhead and reduces performance.

2.2.2 Thread and Block Initialization in CUDA

In CUDA, computation is divided into **threads** and **blocks**, which are organized in a grid. Proper initialization of threads and blocks is crucial for executing the Bitonic Sort algorithm efficiently. The following describes the initialization process in the **V0 implementation**.

Grid and Block Configuration

- The grid consists of multiple blocks, each containing a fixed number of threads.
- The number of threads per block is defined by `THREADS_PER_BLOCK`, set to a power of two (e.g., 1024) to align with the GPU's warp size (32 threads).
- The number of blocks is computed based on the input array size and the number of threads per block:

$$\text{num\_blocks} = \left\lceil \frac{N/2}{\text{THREADS\_PER\_BLOCK}} \right\rceil$$

  where N is the array size and $N/2$ represents the number of element pairs to be processed (each thread handles the two partner-elements for the given *step*).

Thread Initialization

- Each thread computes its **global thread ID** as:

$$\text{global\_tid} = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$$

  - `blockIdx.x`: Index of the block in the grid.
  - `blockDim.x`: Number of threads per block.
  - `threadIdx.x`: Index of the thread within the block.
- The global thread ID determines which element pair the thread will process.

Thread Behavior

- Each thread processes *one element pair* per step.
- The indices of its assigned element and its partner are computed using the `GET_ARR_ID` macro and the XOR operation:

$$\text{idx} = \text{GET\_ARR\_ID}(\text{global\_tid}, \text{step})$$

$$\text{partner} = \text{idx} \oplus (1 \ll \text{step})$$

- The thread compares the elements at `idx` and `partner` and swaps them if they are out of order based on the sorting direction of the current stage, using the `exchange` function.

Explanation of `GET_ARR_ID` Macro

Inputs

- tid: The thread ID.
- step: The current step in the Bitonic Sort algorithm.

Steps

- tid $\gg$ step: Right-shift the thread ID by `step` bits, effectively dividing it by 2^{step}.
- (tid $\gg$ step) $\ll$ step: Left-shift the result back by `step` bits, aligning the thread ID to the nearest multiple of 2^{step}.
- Add the original thread ID to the result to ensure that the thread processes the correct element in the Bitonic sequence.

Example — For tid = 5 and step = 1:

$$\text{GET\_ARR\_ID}(5, 1) = 5 + ((5 \gg 1) \ll 1)$$

$$= 5 + (2 \ll 1) = 5 + 4 = 9$$

Thus, thread 5 processes the element at index 9 in the array.

The `exchange` function

The exchange function ensures that elements are compared and swapped according to the current stage's sorting direction.

Algorithm 3 exchange

```
Require: Input:
 1: data: Pointer to the array in global memory
 2: local_idx: Index of the current element in shared (or global) memory
 3: global_idx: Index of the current element in global memory
 4: partner: Index of the partner element in shared (or global) memory
 5: stage: Current stage of the Bitonic Sort
Ensure: Output: Swaps elements if they are out of order
 6: if (global_idx&($2^{stage+1}$)) == 0 then
                         ▷ Check sorting direction (ascending or descending)
 7:     if data[local_idx] > data[partner] then
 8:         Swap data[local_idx] and data[partner]
 9:     end if
10: else
11:     if data[local_idx] < data[partner] then
12:         Swap data[local_idx] and data[partner]
13:     end if
14: end if
```

2.3 Time Complexity

The Bitonic Sort Algorithm can be analyzed in terms of its *algorithmic complexity*. Below is a concise mathematical analysis of its time complexity.

2.3.1 Stages and Steps

- For an array of size $N = 2^k$, the algorithm requires $\log_2 N$ stages.
- Each stage s consists of $s + 1$ steps.
- The total number of steps is:

$$\text{Total Steps} = \sum_{s=0}^{\log_2 N-1} (s+1) = \frac{\log_2 N \cdot (\log_2 N + 1)}{2} = O(\log^2 N)$$

2.3.2 Work per Step

- At each step, $\frac{N}{2}$ element pairs are compared and swapped.
- Each comparison and swap operation takes $O(1)$ time.
- Each comparison and swap is executed in parallel, but the GPU does not perform all $\frac{N}{2}$ comparisons simultaneously. Instead, the actual parallel execution depends on the number of available processing units, denoted as p. This means that instead of assuming $O(1)$ work per step, we should assume $O\left(\frac{N}{2p}\right)$, where p is hardware-dependent.
- The total work performed is:

$$\text{Total Work} = O\left(\frac{N}{2 \cdot p} \log^2 N\right)$$

2.3.3 Comparison with the Serial Sort

The serial approach has a time complexity of $O(N \log N)$. For the parallel version to be faster, we require:

$$\frac{N}{2 \cdot p} \cdot \log N < N \Rightarrow \frac{\log N}{2 \cdot p} < 1 \Rightarrow \log N < 2 \cdot p \Rightarrow N < 2^{2 \cdot p} \overset{N=2^k}{\Longrightarrow} k < 2 \cdot p.$$

In our analysis, we have $k \in [20, 27]$, meaning the largest problem size will be approximately ~ 4.3 GB. Since $2p \gg k$, we expect the Bitonic Sort implementation to dominate in execution time.

2.4 Algorithm's Optimization (V1)

The **V1 implementation** builds on the baseline **V0** by addressing its primary issue: the **high kernel launch overhead**. In **V0**, each step of the Bitonic Sort algorithm requires a separate kernel launch, leading to significant overhead due to frequent global synchronization and kernel invocations. The **V1 implementation** optimizes this by *fusing multiple steps into a single kernel*, reducing the number of kernel launches and improving overall performance.

Firstly, we will handle all the initial stages with a single kernel call using `bitonic_kernel_v1_first_stages`. This kernel is designed to process the first stages of the Bitonic Sort algorithm in a single execution, leveraging the observation that early stages involve fewer steps and can be efficiently computed within a single kernel call, thereby reducing the overhead of multiple kernel launches.
The maximum step that can be handled in a single kernel call is denoted as `step_max`.

2.4.1 What Does `step_max` Represent?

The parameter step_max represents the **maximum step** that can be handled by a single thread block and is defined as:

$$\text{step\_max} = \log_2(\text{THREADS\_PER\_BLOCK})$$

- **THREADS_PER_BLOCK**: The number of threads per block, set to a power of two (e.g., 1024).
- **step_max**: The maximum step that can be processed within a single thread block.For example, if THREADS_PER_BLOCK = 1024, then step_max = 10. Since each block has 1024 threads, it can handle 2048 elements independently. This implies that the maximum step that can be executed while remaining within these 2048 elements is 1024. If the step exceeds 1024, comparisons will involve elements outside the 2048-element range, breaking the independent execution within a block.

2.4.2 bitonic_kernel_v1_first_stages Algorithm

Algorithm 4 bitonic_kernel_v1_first_stages

```
Require: Input: data: Pointer to the array in global memory — length: Length of the
    array — step_max: Maximum step which can be handled by a single thread block
Ensure: Output: Partially sorted array after the first stages (from 0 to step_max)
 1: global_tid ← blockIdx.x × blockDim.x + threadIdx.x    ▷ Compute global thread ID
 2: if global_tid ≥ (length ÷ 2) then
 3:     return                                  ▷ Exit if thread is out of bounds
 4: end if
 5: for stage = 0 to step_max do
 6:     for step = stage downto 0 do
 7:         idx ← GET_ARR_ID(global_tid, step) ▷ Element index this thread will handle
 8:         partner ← idx ⊕ (1 ≪ step)     ▷ Partner idx this thread will handle (XOR)
 9:         if idx < partner ∧ partner < length then
10:             exchange(data, idx, idx, partner, stage)    ▷ Compare and swap elements
11:         end if
12:         __syncthreads()                 ▷ Local synchronization within the kernel
13:     end for
14: end for
```

2.4.3 bitonic_kernel_v1_lower_steps Algorithm

When processing stages that include steps greater than step_max, we need to use the V0 kernel for those higher steps (> step_max). However, once the steps reach step_max, we no longer need to continue running the V0 kernels for the lower steps. Instead, we can execute the `bitonic_kernel_v1_lower_steps` ***once***, handling all steps from step_max down to 0 in a single kernel call.

Algorithm 5 bitonic_kernel_v1_lower_steps

```
Require: Input: data: Pointer to the array in global memory — length: Length of
    the array — stage: Current stage of the Bitonic Sort — step_max: Maximum step
    which can be handled by a single thread block
Ensure: Output: Partially sorted array after the current stage
 1: tid ← blockIdx.x × blockDim.x + threadIdx.x          ▷ Compute global thread ID
 2: if tid ≥ (length ÷ 2) then
 3:     return                                           ▷ Exit if thread is out of bounds
 4: end if
 5: for step = stage downto 0 do
 6:     idx ← GET_ARR_ID(tid, step)        ▷ Element index that this thread will handle
 7:     partner ← idx ⊕ (1 ≪ step)       ▷ Partner idx that this thread will handle (XOR)
 8:     if idx < partner ∧ partner < length then
 9:         exchange(data, idx, idx, partner, stage)     ▷ Compare and swap elements
10:     end if
11:     __syncthreads()                       ▷ Local synchronization within the kernel
12: end for
```

2.4.4 Host bitonic_sort_v1 function

Below is the **pseudo code** for the `bitonic_sort_v1` function, which orchestrates the execution of the **V1 implementation** of the Bitonic Sort algorithm. This function handles both the **first stages** (using `bitonic_kernel_v1_first_stages`) and the **higher stages** (using `bitonic_kernel_v0` for steps beyond `step_max`).

Algorithm 6 bitonic_sort_v1

```
Require: Input: array: The array to be sorted (of type IntArray)
Ensure: Output: Fully sorted array
 1: Launch bitonic_kernel_v1_first_stages for stages 0 to step_max
 2: Synchronize device                      ▷ Ensure all threads complete the first stages
 3: for stage = step_max + 1 to stages − 1 do
 4:     for step = stage downto step_max + 1 do
 5:         Launch bitonic_kernel_v0 for steps beyond step_max
 6:         Synchronize device                     ▷ Ensure all threads complete the step
 7:     end for
 8:     Launch bitonic_kernel_v1_lower_steps for steps step_max downto 0
 9:     Synchronize device                  ▷ Ensure all threads complete the lower steps
10: end for
```

2.5 Adding Shared Memory (V2)

The V2 implementation builds on the optimizations introduced in V1 by addressing its primary limitation: *inefficient global memory access.* In V1, all data exchanges occur in global memory, which is slow compared to shared memory. The V2 implementation optimizes this by leveraging **shared memory** for intermediate computations, significantly reducing the number of global memory accesses and improving overall performance.

Key Idea: Shared Memory Usage
The V2 implementation introduces shared memory to store intermediate results during the sorting process. Shared memory is much faster than global memory, as it resides on-chip and has significantly lower latency.

- Each thread block loads a portion of the data (2 · THREADS_PER_BLOCK) into shared memory before performing comparisons and swaps.
- This allows multiple comparison steps to be completed within shared memory before writing the results back to global memory.

2.5.1 bitonic_kernel_v2_first_stages Algorithm

The V2 implementation introduces a new kernel, `bitonic_kernel_v2_first_stages`, which handles the *first stages* of the Bitonic Sort algorithm using shared memory. This kernel is designed to process the first stages (from stage 0 to step_max) in a single kernel call, leveraging shared memory for faster data processing.

Algorithm 7 bitonic_kernel_v2_first_stages

```
Require: Input: data: Pointer to the array in global memory — length: Length of
    the array — step_max: Maximum step handled by a single thread block
Ensure: Output: Partially sorted array after the first stages
 1: global_tid ← blockIdx.x × blockDim.x + threadIdx.x        ▷ Compute global thread ID
 2: global_idx ← threadIdx.x × 2
 3: Load data[global_idx] and data[global_idx + 1] into shared memory
 4: __syncthreads()                              ▷ Synchronize threads after loading data
 5: for stage = 0 to step_max do
 6:     for step = stage downto 0 do
 7:         local_idx ← GET_ARR_ID(local_tid, step)            ▷ local_tid is threadIdx.x
 8:         partner ← local_idx ⊕ (1 ≪ step)              ▷ Partner idx in shared memory
 9:         if local_idx < partner ∧ partner < (THREADS_PER_BLOCK × 2) then
                                                     ▷ Compare and swap in shared memory
10:             exchange(shared_data, local_idx, local_idx, partner, stage)
11:         end if
12:         __syncthreads()                         ▷ Synchronize threads after each step
13:     end for
14: end for
15: Write shared memory data back to global memory
```

2.5.2 bitonic_kernel_v2_lower_steps Algorithm

For stages beyond step_max, the **V2 implementation** uses the `bitonic_kernel_v2_lower_steps` kernel to handle the **lower steps** (from step_max down to 0) using shared memory. This kernel is similar to `bitonic_kernel_v1_lower_steps` but leverages shared memory for faster data processing.

Algorithm 8 bitonic_kernel_v2_lower_steps

```
Require: Input: data: Pointer to the array in global memory — length: Length of
    the array — stage: Current stage of the Bitonic Sort — step_max: Maximum step
    handled by a single thread block
Ensure: Output: Partially sorted array after the current stage
 1: global_tid ← blockIdx.x × blockDim.x + threadIdx.x        ▷ Compute global thread ID
 2: global_idx ← threadIdx.x × 2
 3: Load data[global_idx] and data[global_idx + 1] into shared memory
 4: __syncthreads()                               ▷ Synchronize threads after loading data
 5: for step = step_max downto 0 do
 6:     local_idx ← GET_ARR_ID(local_tid, step)              ▷ local_tid is threadIdx.x
 7:     partner ← local_idx ⊕ (1 ≪ step)                  ▷ Partner idx in shared memory
 8:     if local_idx < partner ∧ partner < (THREADS_PER_BLOCK × 2) then
                                                       ▷ Compare and swap in shared memory
 9:         exchange(shared_data, local_idx, local_idx, partner, stage)
10:     end if
11:     __syncthreads()                              ▷ Synchronize threads after each step
12: end for
13: Write shared memory data back to global memory
```

Note: The host `bitonic_sort_v2` function follows the exact same algorithm; however, this time, it calls the `v2` functions instead of `v1`. Once again, for higher steps, the `v0` implementation remains mandatory.

3 Validation Tests & Results

In this section, we describe the methodology used for validation and result computation. The performance results and diagrams presented here are derived from the log files stored in the `/log` folder of our repository.
Each time we run the algorithm, we execute two sorting functions. The first is always the `qsort` function, which is executed before our implementation using the same input data. Its results are stored in a separate array for comparison and validation of our sorting algorithm. After that, we run ***one*** of our own implementations.

Note

The code execution takes place on the GPU partition of the Aristotle HPC system. For details on how to submit and run the code, refer to the `README.md` file in the repository.

3.1 Validation Tests Explained

The validation process is performed immediately after the completion of the Bitonic Sort. To verify that the initial array is correctly sorted, we compare the results of our sorting algorithm with those produced by the built-in `qsort` function. *The `qsort` function is executed before our implementation using the same input data, and its results are stored in a separate array for comparison.*

3.2 Explanation of the log files

The `/log` directory contains the results of executing the `bash-submit-test-cases.sh` test script:

```
bash bash-submit-test-cases.sh 20 27
```

The script above creates jobs to run the C implementation for input sizes ranging from 2^{20} to 2^{27} elements, across all three versions (V0, V1, and V2). This means the script submits a total of $8 \times 3 = 24$ jobs (test runs).
For each test run, two files are generated:

- *`.out` Files:* These files include the total sorting time in milliseconds (or seconds), as well as an indicator confirming whether the sorting was successfully completed.
- *`.err` Files:* These files provide detailed information about any issues that occurred during execution, particularly if the sorting was unsuccessful. Common issues might include the job being terminated due to time limits or other runtime errors.

Files Naming Format inside `log/`

Each file follows the naming format `cuda_sort_P_V_C.*`, where:
- `P` represents a total 2^P elements to sort.
- `V` denotes the Bitonic Sort version used in the execution.
- `C` corresponds to the job's unique ID.

3.3 Time Measurements

For each test run, we record and present three distinct time measurements:

- [**qsort**]: The total execution time of the serial `qsort` function, measured in seconds.
- [**Vx internal time**]: The execution time of the Bitonic Sort (version `x`) excluding memory transfers to and from the device (excludes `cudaMemcpy`). This is measured in milliseconds and represents the *pure sorting time.*
- [**Bitonic Sort Vx**]: The total execution time of the Bitonic Sort (version `x`), including all memory allocations and data transfers between host and device. This metric alone is not a reliable performance indicator, as using unified memory would eliminate memory transfers, making the total time almost equal to the internal time.

For each one of the above, we also calculate the "Normalized Execution Time" which is the execution time per element — computed by dividing the total execution time by the number of elements sorted. This metric allows for a fair comparison across different input sizes.

3.4 Results Presentation

The results inside the log files are summarized in the table below. As observed, each time the number of elements we sort is doubled, the execution times generally increase by approximately doubling. However, for the total Bitonic Sort time, this pattern is not perfectly followed due to the overhead introduced by copying data to and from the device (approximately 20 seconds when 2^{20} elements). As a result, for the lower versions of the Bitonic Sort, this overhead becomes significant, taking nearly as much time as the qsort execution itself.

k : 2^k elements	qsort (sec)	Internal V0 (msec)	Total V0 (sec)	Internal V1 (msec)	Total V1 (sec)	Internal V2 (msec)	Total V2 (sec)
20	0,15	5,3	0,2	3,33	0,20	2,16	0,18
21	0,316	9,98	0,2	6,7	0,19	4,3	0,19
22	0,66	19,56	0,21	13,9	0,2	8,7	0,2
23	1,38	40,25	0,24	29,44	0,23	18,66	0,22
24	2,87	84,34	0,286	62,25	0,26	38,58	0,24
25	5,98	179,1	0,4	133,9	0,35	84,35	0,29
26	12,56	382,1	0,63	288,5	0,53	189,6	0,44
27	26,1	815,86	1,39	623,25	0,94	414,87	0,76

Now, lets visualize the results presented in the table by introducing relevant diagrams. The first diagram illustrates the ***total*** execution times of the `qsort` and the `V0` Bitonic Sort implementation as a function of k, where the dataset size is 2^k elements.
As observed, even the `V0` implementation surpasses the serial `qsort` in performance for larger datasets. The reason why `qsort` appears faster for smaller dataset sizes is due to the additional overhead introduced by CUDA memory transfers. Since the total time for `V0` includes the time required for copying data to and from the device, this overhead significantly impacts performance at smaller scales. However, it is important to note that the internal execution time of `V0` (excluding memory transfer) is already faster than `qsort` even for 2^{20} elements.

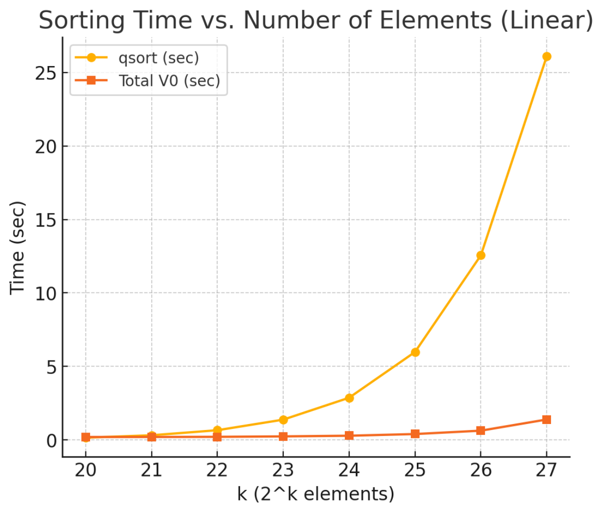

Figure 1: qsort - total V0 times

Next, we present a diagram that compares the total execution time and the internal execution time of the `V0` implementation. This visualization helps us better understand the additional overhead introduced by memory transfers between the host and the device. As evident from the diagram, this overhead increases for larger dataset sizes. This behavior is expected, as the memory copy operation exhibits linear complexity with respect to the number of elements. Consequently, for larger datasets (2^{27} elements), the data transfer time becomes a more significant portion of the total execution time.

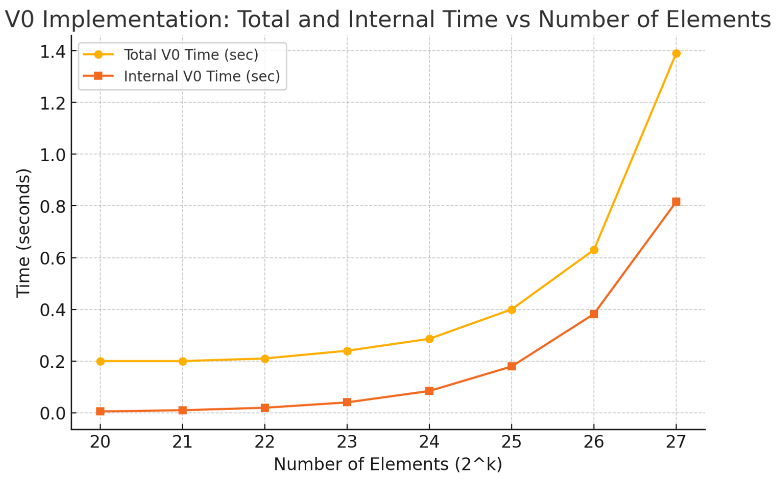

Figure 2: Bitonic Sort V0: Total & Internal time

Lastly—after illustrating the difference between the internal and total execution times—we proceed to compare the internal execution times of all our implementations. We choose to present the pure sorting times (internal) to better highlight the acceleration achieved through our optimizations.
As evidenced in the results, the `V0` implementation is the slowest, which aligns with our theoretical analysis. On the other hand, the `V2` implementation, leveraging both shared memory and reduced kernel calls, demonstrates the best performance among all versions.

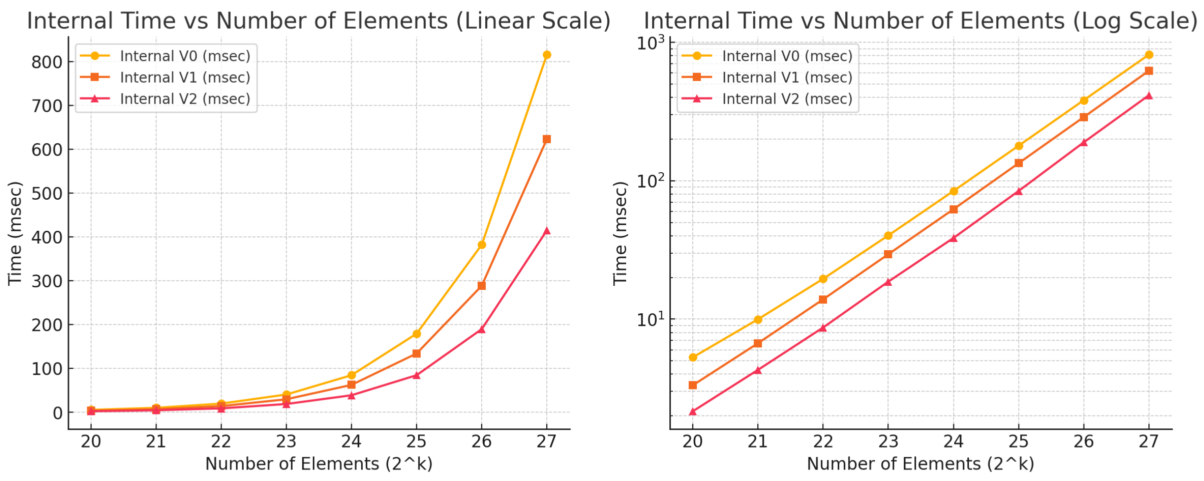

Figure 3: Compare Versions' Times