Parallel sorting of large arrays using Bitonic Sort in CUDA

Epameinondas Bakoulas

February 2025

1 Objective

Sorting algorithms usually take $O(nlogn)$ time to sort an array of size n. Bitonic sort is a parallel sorting algorithm that can sort very large arrays faster. By leveraging the power of modern GPUs we can achieve a noticeable speedup compared to CPU sorting algorithms, either single threaded or multi threaded.

2 Algorithm Analysis

2.1 Overview

Our goal is to sort an array of size 2^q, where $q = [15 : 28]$. We will present a total of 3 implementation, V0, V1 and V2. The first implementation, V0, is a simple implementation of the algorithm that is noticeably slower than the other two.

2.2 Algorithm implementation in CUDA (V0)

The algorithm will follow the standard **Bitonic Sort Network** pattern as shown in Figure 2. Our goal is to create bitonic sequences of size $2, 4, 8, ..., p$ and then, with the correct exchanges between processes, make bigger bitonic sequences. The main logic of the algorithm is shown in Figure 1. The variable `group_size` is the size of the bitonic sequence that we create in each iteration. On every iteration, we double the size of `group_size` to create bigger bitonic sequences.

The variable `distance` is the distance between the elements that we compare in each iteration. We start with `group_size/2` and we divide it by 2 in each iteration. This way, we compare the elements of the bitonic sequence in a specific order, as shown in Figure 2.

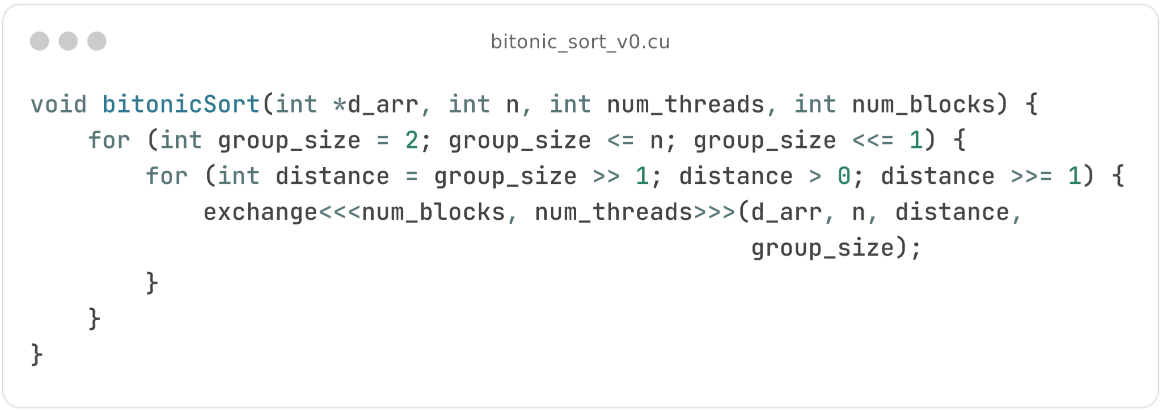
For each `group_size` we need to perform $\log_2$ `group_size` iterations. For example, if we have 8 elements in each group, we need to perform 3 iterations with a specific distance each time.

The kernel function `exchange` is responsible for exchanging the elements of the bitonic sequence. Before we run it, we need to specify the number of blocks and threads (in each block) that we will use. Since we're performing $n/2$ exchanges in each iteration, we need to have **n/2 threads in total**. We will use the max number of threads allowed in each block, which is 1024. This means that we need to have $\mathbf{\frac{n/2}{1024}}$ **blocks in total**.

Each thread inside the kernel function is responsible for exchanging two elements of the bitonic sequence. It needs to compute the following things:

1. Who am I? What's my thread ID?
2. What's the index of the element that I'm going to work with?
3. What's the index of the element that I'm going to compare with?
4. Which index should keep to minimum and which the maximum?

The answers to those questions can be found in Figure 3. The method of finding the partner is explain later on in detail with the hypercube model.



bitonic_sort_v0.cu

```
void bitonicSort(int *d_arr, int n, int num_threads, int num_blocks) {
    for (int group_size = 2; group_size <= n; group_size <<= 1) {
        for (int distance = group_size >> 1; distance > 0; distance >>= 1) {
            exchange<<<num_blocks, num_threads>>>(d_arr, n, distance,
                                                  group_size);
        }
    }
}
```

Figure 1: Bitonic Sort Algorithm

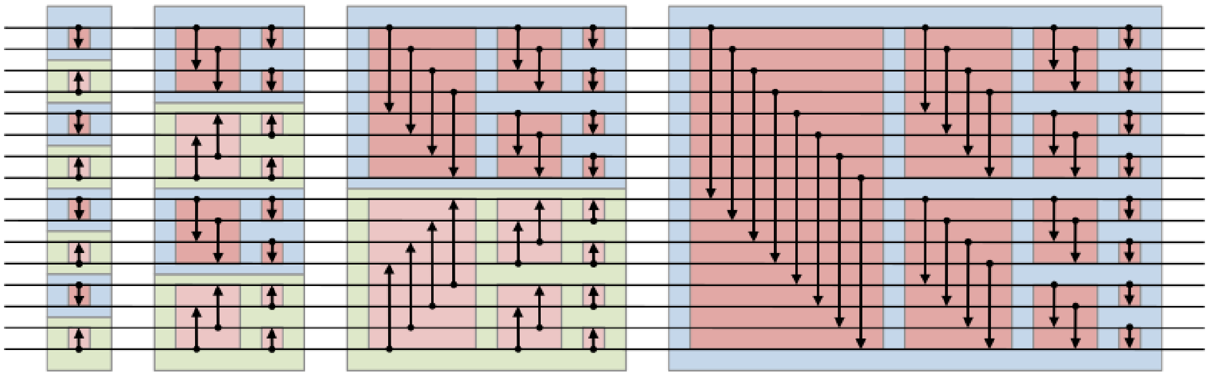
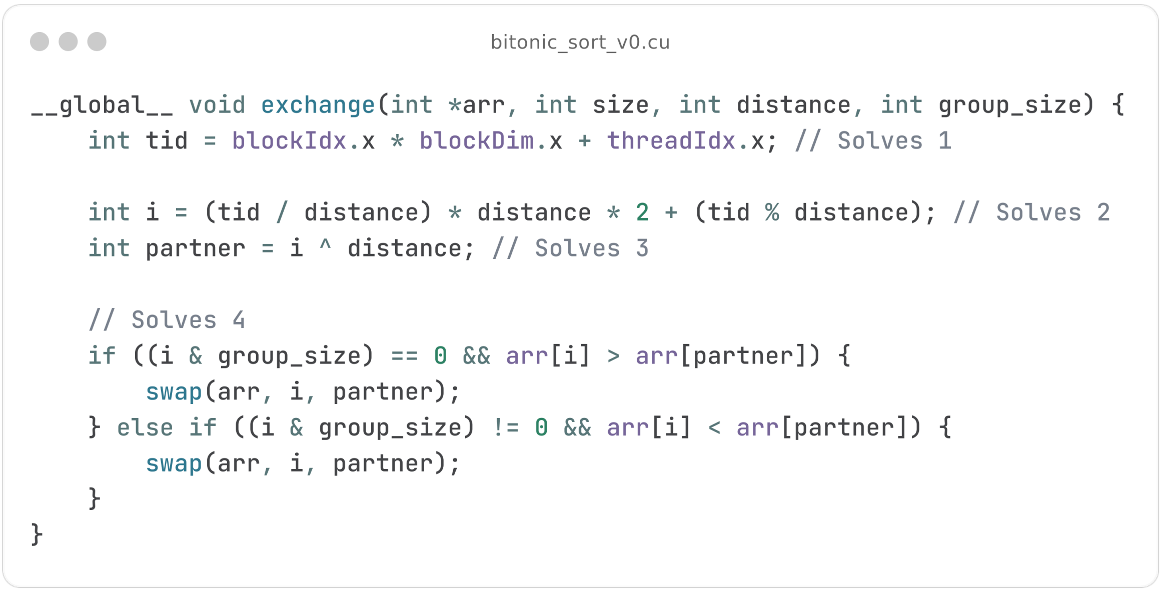


Figure 2: Bitonic Sort Network. The arrow points to the element that keeps the maximum.



bitonic_sort_v0.cu

```
__global__ void exchange(int *arr, int size, int distance, int group_size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x; // Solves 1

    int i = (tid / distance) * distance * 2 + (tid % distance); // Solves 2
    int partner = i ^ distance; // Solves 3

    // Solves 4
    if ((i & group_size) == 0 && arr[i] > arr[partner]) {
        swap(arr, i, partner);
    } else if ((i & group_size) != 0 && arr[i] < arr[partner]) {
        swap(arr, i, partner);
    }
}
```

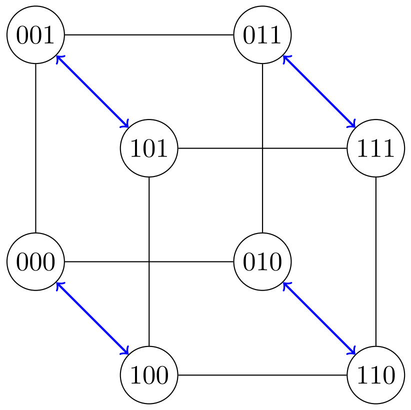
Figure 3: Exchange Kernel function. Each thread is responsible for only one exchange.

2.3 Hypercube model

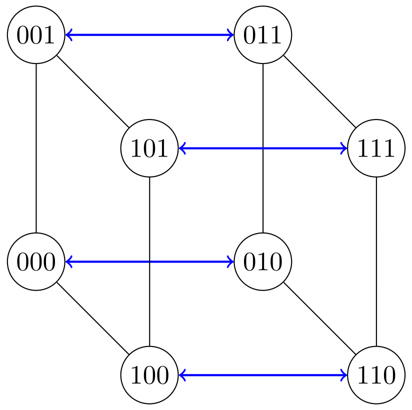
Finding the right partner to exchange data with in each stage of the algorithm can be quite tricky. One possible way is to leverage the power of the hypercube model. The index of the element that we're going to exchange with can be found by performing a bitwise XOR operation between the current index and the distance:

$$i \oplus distance$$

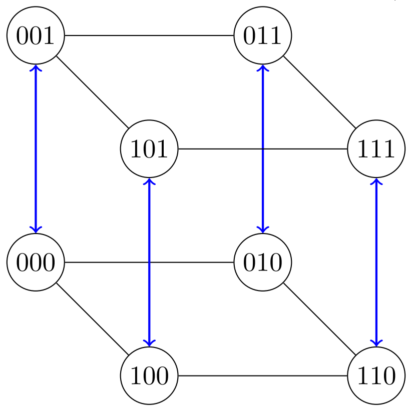
In Figure 4 we can see the exchanges performed between 8 elements.



(a) Exchanges at distance 4



(b) Exchanges at distance 2



(c) Exchanges at distance 1

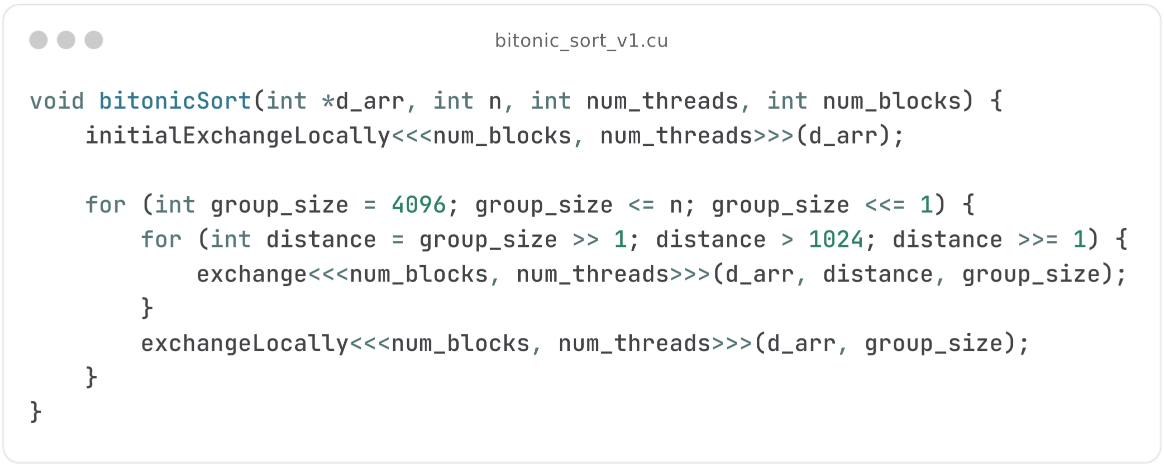
Figure 4: (Hyper)cube model with 8 processes.

2.4 Minimizing the kernel calls (V1)

In the previous implementation, we had to call the kernel function `exchange` for each exchange. This is necessary if we want to achieve synchonization. However, we can leverage the fact that threads inside a block can synchronize with each other. This means that all 1024 threads inside the block can be synchronized.

For the V1 implementation, we will create 2 new kernel functions `initialExchangeLocally` and `exchangeLocally`. The first one will be responsible for the initial exchanges at the start of the algorithm, up until we reach $group\_size = 2048$. This means that all of the exchanges with $distance <= 1024$ will be performed in a **single kernel call**. After that, we will call the same V0 kernel function up until we reach $distance = 1024$, where we will call the `exchangeLocally` function.

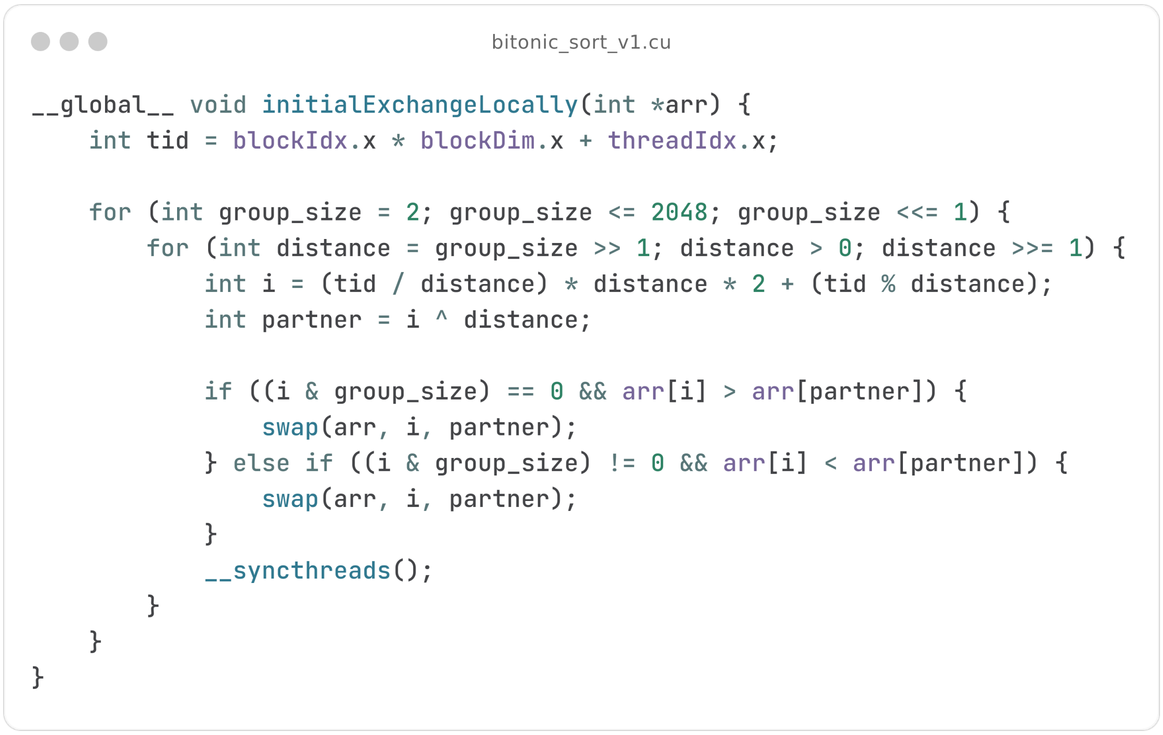
The 2 new functions might seem similar, but they have a key difference. The `initialExchangeLocally` function will increase the `group_size` by 2 in each iteration, while the `exchangeLocally` function runs on a specific `group_size` value. The code is shown in Figures 5 and 6.


bitonic_sort_v1.cu

```
void bitonicSort(int *d_arr, int n, int num_threads, int num_blocks) {
    initialExchangeLocally<<<num_blocks, num_threads>>>(d_arr);

    for (int group_size = 4096; group_size <= n; group_size <<= 1) {
        for (int distance = group_size >> 1; distance > 1024; distance >>= 1) {
            exchange<<<num_blocks, num_threads>>>(d_arr, distance, group_size);
        }
        exchangeLocally<<<num_blocks, num_threads>>>(d_arr, group_size);
    }
}
```

Figure 5: Bitonic Sort Algorithm, modified to perform less amount of kernel calls (V1)


bitonic_sort_v1.cu

```
__global__ void initialExchangeLocally(int *arr) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    for (int group_size = 2; group_size <= 2048; group_size <<= 1) {
        for (int distance = group_size >> 1; distance > 0; distance >>= 1) {
            int i = (tid / distance) * distance * 2 + (tid % distance);
            int partner = i ^ distance;

            if ((i & group_size) == 0 && arr[i] > arr[partner]) {
                swap(arr, i, partner);
            } else if ((i & group_size) != 0 && arr[i] < arr[partner]) {
                swap(arr, i, partner);
            }
            __syncthreads();
        }
    }
}
```

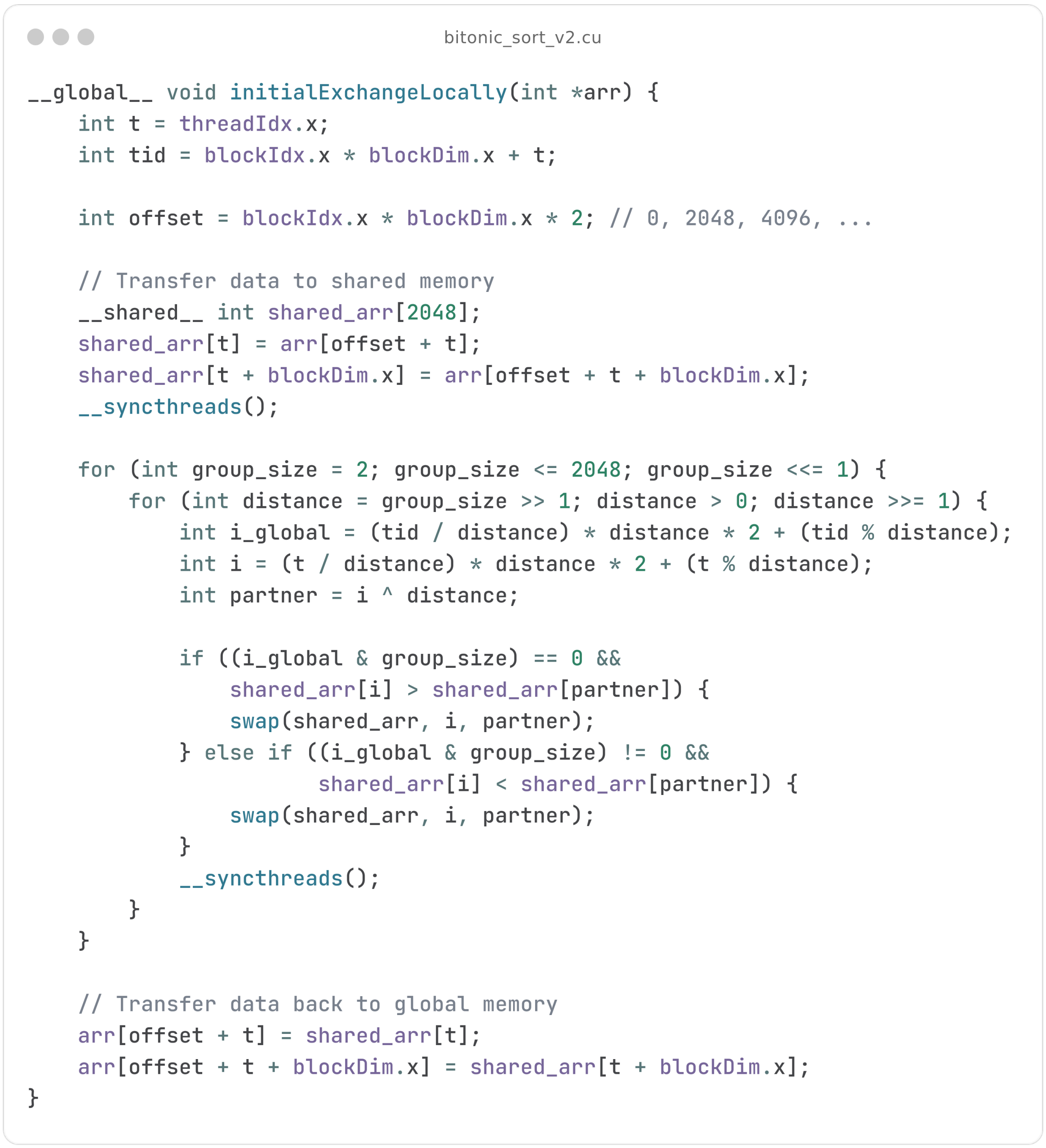
Figure 6: Initial exchanges performed at the start of the algorithm (V1)

2.5 From Global memory to Shared (V2)

We can further optimize the algorithm by using shared memory. Generally, shared memory has 100x faster access time than global memory. The V2 implementation will use shared memory inside the functions `initialExchangeLocally` and `exchangeLocally`.

First, we need to transfer the data from the global memory to the shared memory. Each thread will be responsible for copying 2 elements, thus we'll transfer a total of $1024 * 2 = 2048$ elements in each block. The explanation behind this is the fact that we're creating a total of $n/2$ threads, so each thread is responsible for 2 elements. After transferring, we need to sync all of the threads in the block before we start exchanging.

Lastly, all array indexing needs to be modified to be based on the shared memory. After we finish the exchanges, we will transfer the elements from the shared memory back to the global memory. The `initialExchangeLocally` function is shown in Figure 7. The full code can be found on the Github repository.



```
__global__ void initialExchangeLocally(int *arr) {
    int t = threadIdx.x;
    int tid = blockIdx.x * blockDim.x + t;

    int offset = blockIdx.x * blockDim.x * 2; // 0, 2048, 4096, ...

    // Transfer data to shared memory
    __shared__ int shared_arr[2048];
    shared_arr[t] = arr[offset + t];
    shared_arr[t + blockDim.x] = arr[offset + t + blockDim.x];
    __syncthreads();

    for (int group_size = 2; group_size <= 2048; group_size <<= 1) {
        for (int distance = group_size >> 1; distance > 0; distance >>= 1) {
            int i_global = (tid / distance) * distance * 2 + (tid % distance);
            int i = (t / distance) * distance * 2 + (t % distance);
            int partner = i ^ distance;

            if ((i_global & group_size) == 0 &&
                shared_arr[i] > shared_arr[partner]) {
                swap(shared_arr, i, partner);
            } else if ((i_global & group_size) != 0 &&
                       shared_arr[i] < shared_arr[partner]) {
                swap(shared_arr, i, partner);
            }
            __syncthreads();
        }
    }

    // Transfer data back to global memory
    arr[offset + t] = shared_arr[t];
    arr[offset + t + blockDim.x] = shared_arr[t + blockDim.x];
}
```

Figure 7: Initial exchanges performed at the start of the algorithm, using shared memory (V2)

3 Benchmarks

We ran 2 different benchmarks on 2 machines. The first one is on a home computer with an RTX 4060 (8GB VRAM) and an i5-11400F CPU. The other one is on the Aristotelis cluster with a Tesla P100 and an Intel Xeon E5-2640 v4 CPU. For comparison, we also run some CPU sorting algorithms, such as `QuickSort` and a parallel sorting algorithm `__gnu_parallel::sort` from the library `parallel/algorithm` with 8 threads (with the help of LLMs).

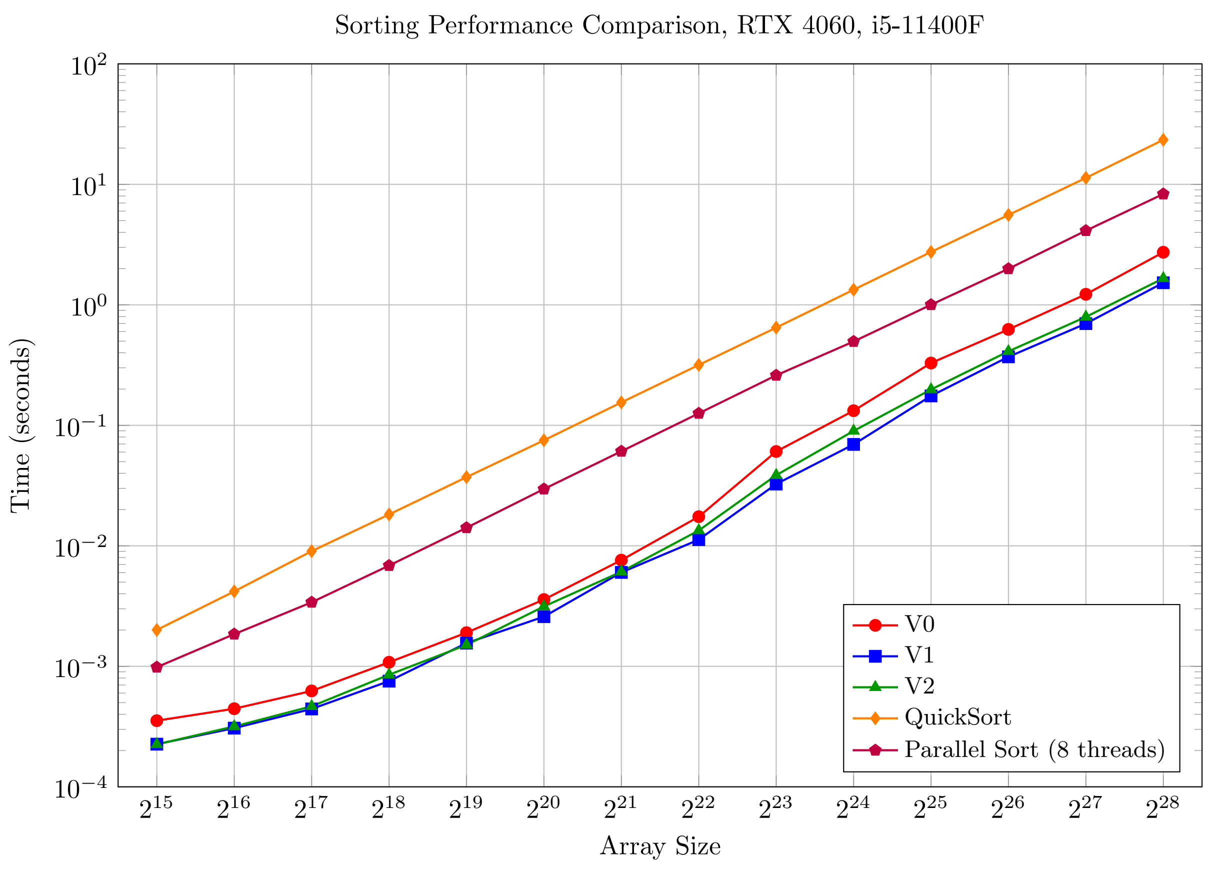


Figure 8: Performance comparison of different sorting implementations.

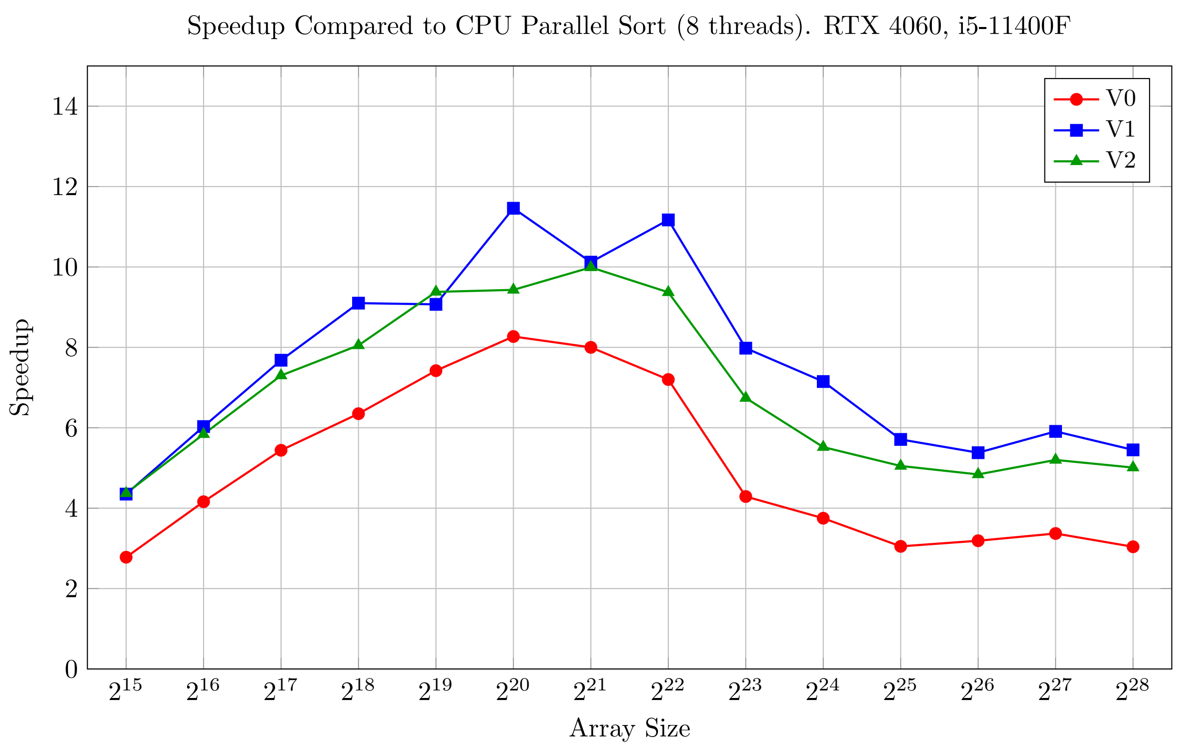


Figure 9: Speedup comparison of GPU implementations relative to CPU parallel sort with 8 threads.

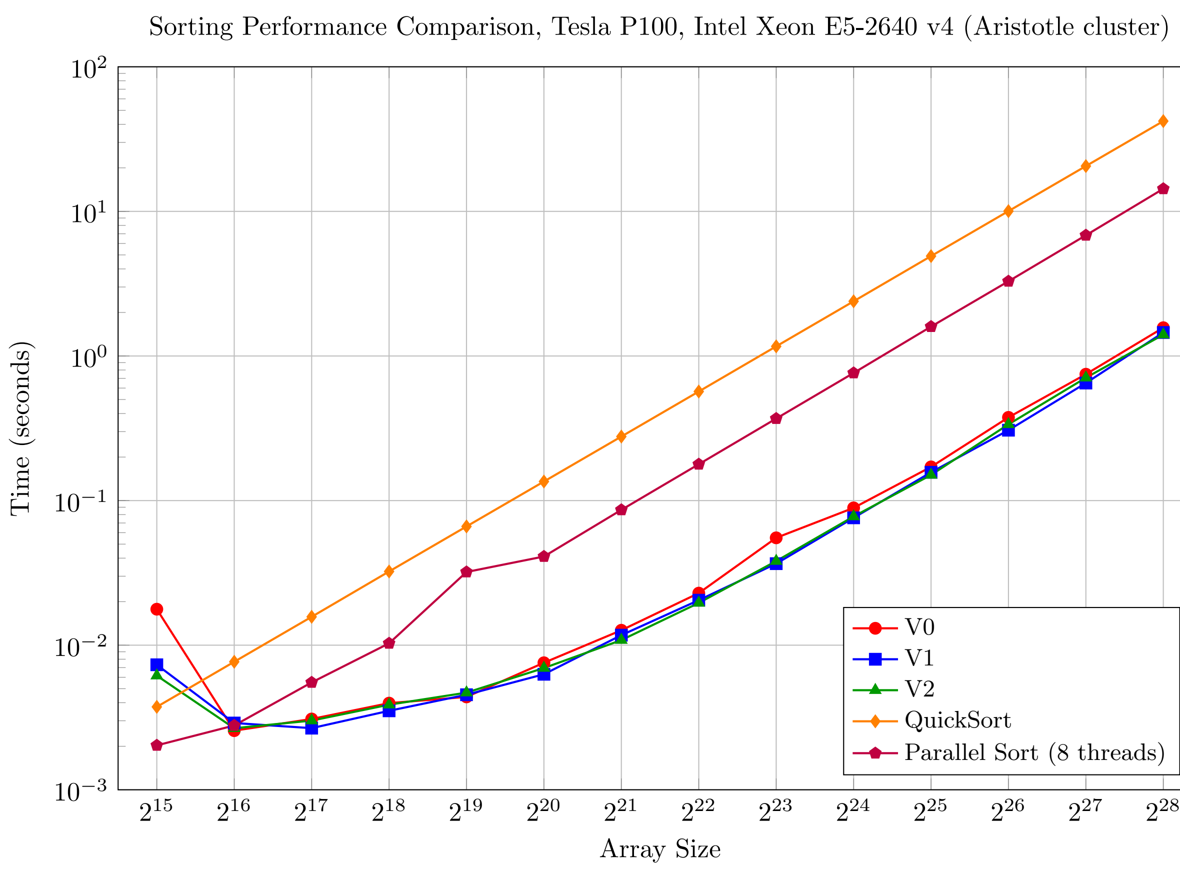


Figure 10: Performance comparison of different sorting implementations on Aristotelis cluster.

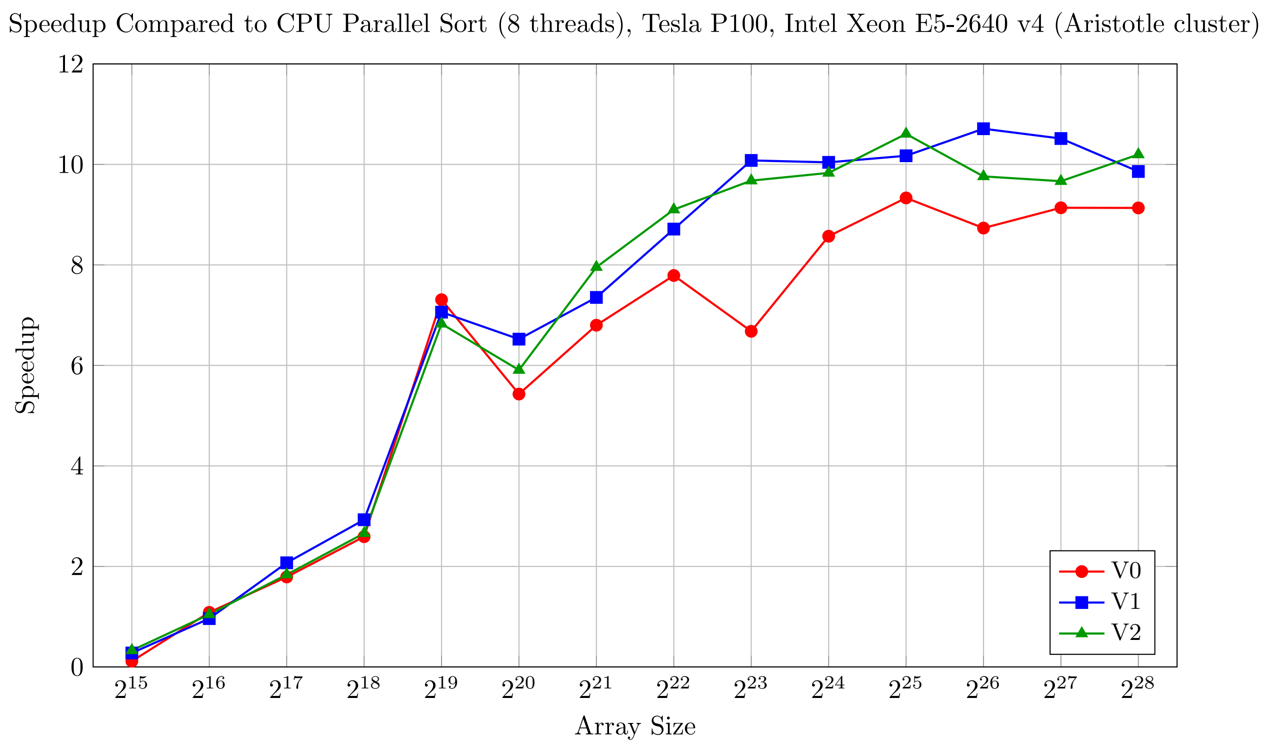


Figure 11: Speedup comparison of GPU implementations relative to CPU parallel sort with 8 threads on Aristotelis cluster.

In both tests we can see that the V1 implementation is generally the fastest, sometimes surpassed by V2. This result is not expected since the V2 implementation uses shared memory, which is faster than global memory. However, the synchronizations required between the threads when transferring the memory could be the reason for the slowdown. A thorough measurement of the execution time in each step of the algorithm was performed, with no apparent bottleneck (they can be found on the Github repository). Though as expected, the V0 implementation is the slowest.

As for the speedup, we can see that there's a noticeable speed bump compared to a parallel CPU algorithm, indicating that the algorithm can utilize the GPU resources to sort arrays **up to 12 times faster**.

4 Conclusion

Bitonic sort is a powerful algorithm that can sort large arrays in a parallel manner. By efficiently utilizing the GPU resources with CUDA, we can achieve a noticeable speedup compared to CPU algorithms. We would expect that the V2 implementation is the fastest, but further analysis is needed to evaluate the result.

The source code can be found on the Github repository.