Report of Architecture and Platform For Artificial Intelligence, Module 1

Raffaele Disabato 0001057619

1 Introduction

This report presents the results of a project aimed at implementing and testing two parallel sorting algorithms, radix and bitonic, in CUDA C.

The primary goal of the project was to leverage the power of GPU parallelism to speed up the sorting process and compare the performance of the two algorithms.

I measured the performance of the algorithms under different conditions, including different input sizes and different numbers of threads, and compared the results to determine which algorithm was faster or more efficient.

2 Bitonic Sorting

The Bitonic Sort algorithm is a comparison-based sorting method that effectively sorts a sequence by breaking it down into smaller sub-problems. These sub-problems are then sorted in a specific order, forming a bitonic sequence, which is characterized by an initial increasing phase followed by a decreasing phase or vice versa.

This sorting approach is often described as a recursive algorithm. It starts by dividing the input data into two halves, each of which is recursively sorted using the Bitonic Sort algorithm. Once the two halves are sorted, they are merged using bitonic merge operations. These merge operations are performed in pairs, and parallelization can be employed to enhance efficiency.

During each pair of merge operations, the bitonic sequence is split into sub-sequences that are sorted in opposite directions. To ensure the resulting sequence remains bitonic, comparisons are made between pairs of elements, and elements are swapped if needed.

This merging and splitting process is repeated iteratively until the entire input sequence is sorted into a single bitonic sequence. The time complexity of the algorithm is $O(nlog^2n)$, where n is the size of the input sequence.

An example of basic code for the bitonic sort without parallelism could be this

```
void bitonic_sort(int * arr, int array_size) {
    for (int k = 2; k <= array_size; k <<= 2) {
        for (int j = k >> 1; j > 0; j = j >> 1) {
            for (int i = 0; i < array_size; i++) {
                int ixj = i^j;
                if (ixj > i) {
                    if (i&k == 0 && arr[i] > arr[ixj])
                        swap(arr[i], arr[ixj]);

                    if (i&k != 0 && arr[i] < arr[ixj])
                        swap(arr[i], arr[ixj]);
```

```
                }
            }
        }
    }
}
```

Listing 1: Basic code for bitonic sort in C

In my implementation of the bitonic algorithm the calls of the merge, compare and swap of elements are a unique kernel function, so in the function bitonic_sort I removed the most inner for cycle and used a kernel function instead.

```
void bitonic_sort(int * input, int array_size) {
    [...]
    bitonic_sort_step<<<BLOCKS, THREADS>>>(input_cuda, j, k);
    [...]
}

__global__ void bitonic_sort_step(int * input, int j, int k) {
    unsigned int i = threadIdx.x + blockDim.x * blockIdx.x;
    unsigned int ixj = i^j;

    if ((ixj) > i) {
        if ((i&k) == 0 && input[i] > input[ixj])
            swap(input[i], input[ixj]);

        if ((i&k) != 0 && input[i] < input[ixj])
            swap(input[i], input[ixj]);
    }
}
```

Listing 2: Parallel code for bitonic sort in C CUDA

The unique constraint for this implementation is that the array size must be a power of 2, otherwise the sort will not be correct.

3 Radix Sorting

Radix sort is a non-comparative integer sorting algorithm that sorts data by grouping elements into buckets based on their digit values. It works by processing each digit of the input, starting from the least significant digit, and placing elements into buckets based on that digit value. The elements are then re-arranged based on the order of the digits, and the process is repeated for each subsequent digit until all digits have been processed.

I decided to implement the 4-way radix sort described in this paper, the algorithm is composed of 3 major subsystems:

- **Implicit Four-Way Radix Counting**: the algorithm computes the frequency of each possible element in the input data using a four-way radix counting approach. This step divides the input array into blocks and performs parallel counting using SIMD (Single Instruction, Multiple Data) processing;
- **Prefix Sum Positioning**: after obtaining the local frequency lists for each block, the algorithm calculates the prefix sum to determine global positions for each element. This step is necessary to determine the sorted position of each element within its radix group;

- **Final Mapping**: using the computed global positions, the algorithm rearranges the input data to its sorted order, completing the sorting process;

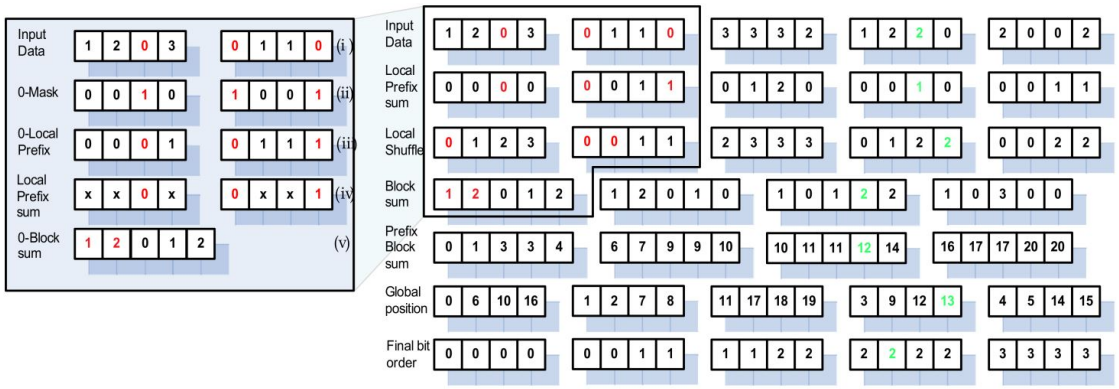

Figure 1: The basic steps of our 4-way radix sort algorithm operating on a 20 component array of 2-bit integers. The right block from top to bottom shows the steps involved in the four-way radix sorting. In the left zoom-in the computation of the local prefix sum array is shown in more detail for the digit "0"

I converted the pseudo-code from the paper into CUDA C, and the three subsystems are the three functions in this for loop:

```
void radix_sort(int* const out, int* const in, unsigned int in_len) {
    [...]
    for (unsigned int bit = 0; bit <= 30; bit += 2) {
        radix_sort_local<<<grid_size, block_size, shared_size>>>(...);

        sum_scan_blelloch(scan_block_sums, block_sums, block_sums_len);

        global_shuffle<<<grid_size, block_size>>>(...);
    }
    [...]
}
```

Listing 3: Main loop for parallel radix sort

In order to perform scan prefix sum on the block sum array I implemented the NVIDIA's algorithm described here, it employ a common algorithmic structure in parallel computing: balanced trees. To compute the prefix sum, a balanced binary tree is constructed from the input data and swept to and from the root. A binary tree with n leaves has $d = log_2(n)$ levels, with 2^d nodes on each level. If one add is conducted for every node, it will perform $O(n)$ additions on a single tree traverse.

The tree constructed is a notion that it is used to decide what each thread performs at each stage of the traversal. It performs the operations on an array in shared memory in work-efficient scan approach. The algorithm is divided into two phases: reduction (also known as the up-sweep phase) and down-sweep. During the reduction step, it walks the tree from leaves to roots, computing partial sums at internal nodes. This is also known as a parallel reduction since the root node (the final node in the array) stores the sum of all nodes in the array after this phase.

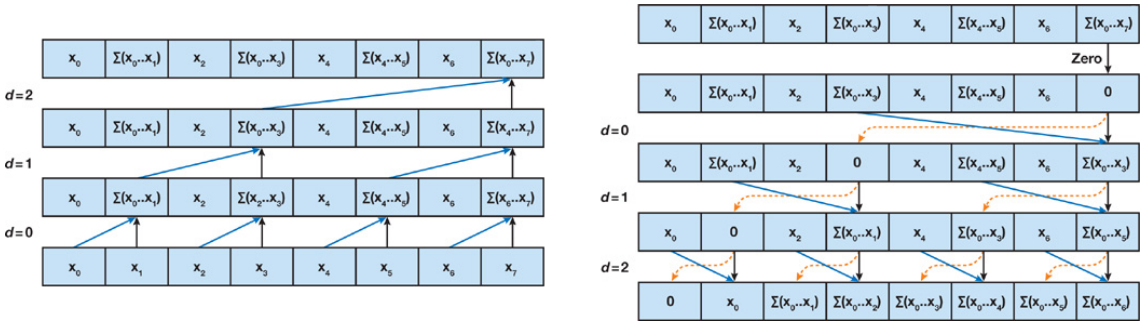

Figure 2: An Illustration of the Up-Sweep, or Reduce, Phase of a Work-Efficient Sum Scan Algorithm on the left and an Illustration of the Down-Sweep Phase of the Work-Efficient Parallel Sum Scan Algorithm on the right.

Based on the explanation provieded by Blelloch, the array can be of an arbitrary size, the basic idea is simple. We divide the large array into blocks that each can be scanned by a single thread block, and then we scan the blocks and write the total sum of each block to another array of block sums. We then scan the block sums, generating an array of block increments that that are added to all elements in their respective blocks. Handling non-power-of-two dimensions is easy. We simply pad the array out to the next multiple of the block size B (B is the number of elements processed in a block). The scan algorithm is not dependent on elements past the end of the array, so we don't have to use a special case for the last block.

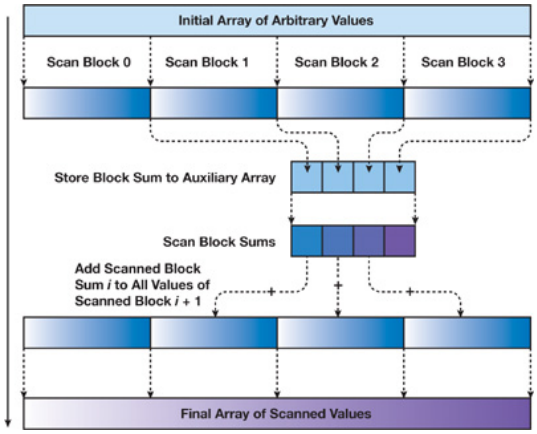

Figure 3: Algorithm for Performing a Sum Scan on a Large Array of Values

At the end, the global position is computed by adding the local block position to the prefix block sum of each bit combination. The resulting fetch returns the new position of the data in the sorted array, a simple mapping function will then permute the data. Similar to the full order checking operation, we only need the position to perform the mapping function, so we can combine these two steps into one, and compute the global position value in shared memory only without explicitly moving it into global memory, this strategy saves us one read/write operation into the global position array which is costly

4 Experiments and Results

The evaluation of the performance involved analyzing the efficiency and scalability of parallel sorting algorithms compared to the sequential sort algorithms.

To check the algorithm's performance, various data sizes were considered, starting with 10 elements for the radix and 8 for the bitonic algorithm, to more than 100.000.000 elements.

For the bitonic algorithm I tried to change the max number of threads in a single block, starting from 256 then 512 and in the end 1024.

With the chart we can see that with 1024 threads we have the best performance, even if the time elapsed is quite similar, so there is not a big change in the performance with different number of threads for block.

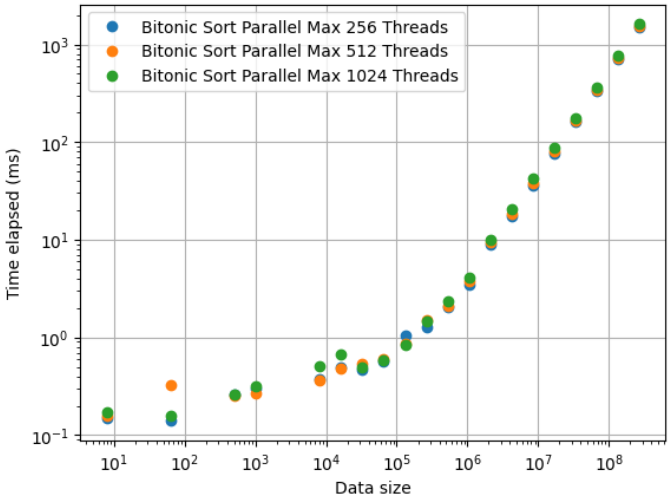

Figure 4: Results for Bitonic Algorithm with different max number of threads

For the radix algorithm I tried to change the max block size from 512 to 1024, and we can see from the chart that with max block size 1024 we have the best performance.

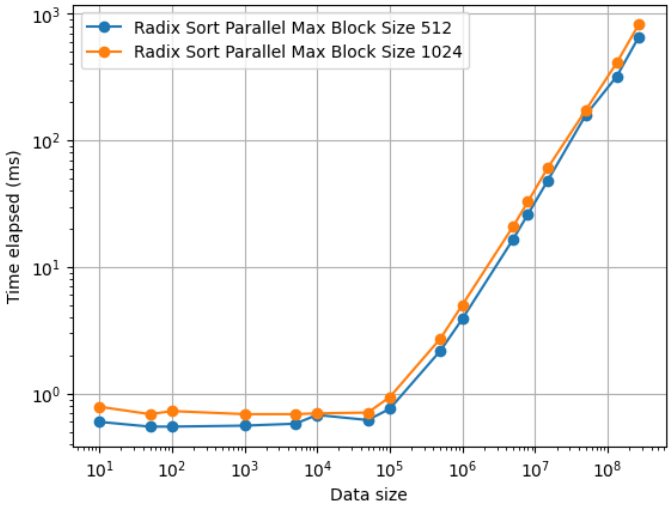

Figure 5: Results for Radix Algorithm with different max block sizes

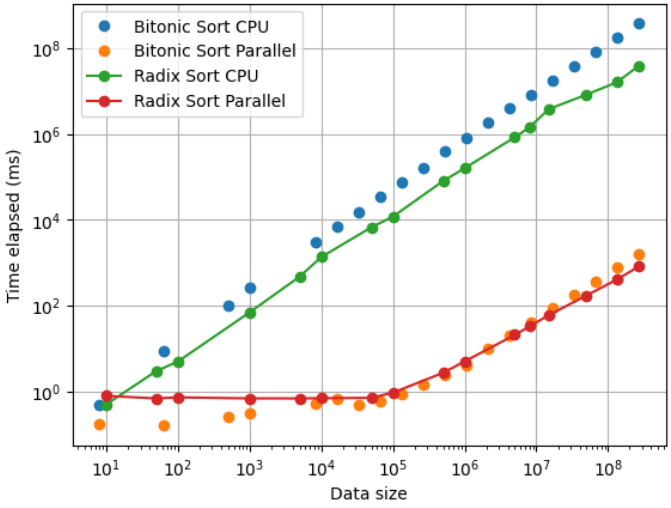

Figure 6: Results for Radix Algorithm and Bitonic Algorithm with CPU and GPU

When contrasting the performance of radix and bitonic sorting algorithms between the CPU and GPU, it becomes evident that the parallel GPU implementation lags behind when handling small data sizes. However, as the dataset size increases, the CPU-based algorithms exhibit an exponential growth in the time required to execute the sorting operations.

The GPU algorithms exhibit a constant time complexity for sorting arrays until the threshold of 10,000 elements is reached. Beyond this point, there is a notable exponential increase in the time required to execute the sorting operation. It is worth noting, however, that even with this increase in computational time, GPU algorithms outperform their corresponding CPU counterparts in terms of sorting efficiency.