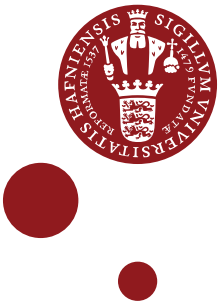
Programming Massively Parallel Hardware

Implementation of Fast Sorting Algorithms for GPU

Axel Schwarzenbach
Gilles Souton

November 9, 2022



Contents

Chapter 1

Introduction

The radix sort algorithm is a stable counting sort, meaning that it does not use comparisons to sort the elements of an array. The algorithm is fairly easy to parallelize using parallel blocks such as prefix sum and scatter.

1.1 Radix sort

Explanation Considering an array of unsorted elements (keys), we compute each key's rank for each digit. The rank will be used to place the corresponding key in its corresponding bucket. Having computed the rank of each key we scatter the elements, into the output array with respect to their corresponding rank. The sort is done from the least significant digit to the most significant to guarantee to have the sequence sorted after all digit passes are complete.

Assumptions In order to work radix sort algorithm assumes that the keys can be ordered lexicographically.

1.1.1 Parallel blocks and work, depth complexity

The algorithm can be implemented with parallel blocks, the position where elements should be placed can be computed with the scan operation, and elements are moved to their corresponding position with the scatter operation.

Depth complexity When looking at the depth complexity of the underlying operations such as scan, scatter, and map, we can expect a depth complexity of $O(n \cdot log(n))$

Work complexity Each pass on the input array (keys) performs a linear number of operations on the input and can be resumed to $O(n)$

1.2 High Level Implementation

We chose to implement the radix sort algorithm as presented by Satish et Al. The algorithm below shows the high-level conceptual algorithm. A radix sort pass consists of computing the histogram, scanning the result and scattering the array with the computed indices. In total we compute $k = 32/b$ passes in which we look at b bits simultaneously.

Algorithm 1 Radix Sort

```
let radix-sort (lst: [n]i32) : [n]i32 =
    for nth_bits < size_in_bits i32 do
        let histogram = computeHistogram lst
        let indices = scan (+) 0 histogram
        in scatter lst indices lst
```

Word depth asymptotic The work depth asymptotic of the proposed high-level implementations is identical as described in chapter 1. The compute histogram function essentially increases a counter corresponding to the value of a rank computed. This amounts to a work complexity of $O(n)$. Furthermore, it is fully parallelizable as all computations can be done independently, and hence has a depth complexity of $O(1)$. The scan function has a work complexity of $O(n)$ and a depth complexity of $O(log(n))$, as described in the lecture document. The scatter function has a work complexity of $O(n)$ and a depth complexity of $O(1)$. As our algorithm calls up these three functions a constant amount of time, radix-sort amounts to a total work complexity of $O(n)$ and depth complexity of $O(log(n))$.

Expectations When looking at the work and depth complexity of the algorithm we expect the algorithm to run efficiently on GPU, since that the operations that compose our implementation consist of basic parallel block operations such as *scan map* and *scatter*. Those operations are already highly parallelizable on GPU and has a low work depth complexity, the composition of these functions would theoretically make the algorithm efficient on GPU.

1.3 CUDA implementation

Our CUDA implementation focuses on using block-level parallelism, the main focus for the radix sort is to keep the memory access as coalesced as possible to maximize performance.

To summarize our implementation it can be seen as the following. (on a block level)

Kernel 1

1. Creating a flag array of the input array (keys), meaning marking where are each element.
2. Applying a prefix sum on the flag array.
3. Extracting the last column of the scanned flag array, representing the histogram of the input array (occurrences, for each rank of each key)
4. Scan the histogram
5. Using the scanned flag array, histogram, and scanned histogram to sort the array locally on a block level with the rank of each key (bucket).

Kernel 2

1. Transpose the histogram from kernel 1.
2. Scan the transpose histogram.

Kernel 3

1. Using the transposed scan histogram and histogram, scatter the element to their corresponding computed index in a coalesced fashion.

Implementation details The parallel version of radix sort will make use of base 2 to sort elements instead of using base 10 such as in most of the sequential implementations, leading to sorting the elements, by looking at group of bits. For our base case, we decided that by default the algorithm would look at 4 bits, and each thread would process 4 elements at the time. Previous papers have suggested that this bucket size and the number of elements processed per thread, combined with a block size of 256 optimally harness the computing power provided by GPUs. We theorize that this is due to the standard configuration of 8 SMs per processor cluster in most GPUs.

The three kernels above are executed 8 times in a row, starting with the 4 least significant bits first, up to the 4 most significant bits. Executing them in this order guarantees elements with equal values to maintain their relative order, hence meaning that this implementation is a stable sorting algorithm. Furthermore, this guarantees a deterministic outcome of the algorithm.

1.3.1 Computing the histogram (Kernel 1)

The computation of the histogram is done the first kernel, the challenge here was to compute the histogram while sorting the array locally on the block level.

Sorting the array on the block level is part of the algorithm since it is beneficial for memory-coalesced access in the other kernels, and especially the last one where we have to scatter the elements to their right position. Furthermore, it has the benefit of eliminating the use of atomic adds otherwise required to compute the histogram, which have a bad impact on performance.

Flag array

Main idea To be able to sort the array locally using radix sort itself it was necessary to compute a flag array, indicating the position of each element of the input array for a given index and rank.

`flag[rank][position]`

Using this array for a given rank and position we know if a the certain key exists.

Challenges The challenge here is to minimize the accesses to global memory, to do so, we decided to compute the flag array in shared memory since we compute on a block level it makes sense that this flag array is shared between the same block threads.

Implemenation details

- Each thread will compute n elements from the input array.
- Compute the rank of the given key.
- Mark for the given index in the flag array.
- store in shared memory the key, (for later use to limit access to global memory).

```
for (uint32_t i = 0; i < elem_pthread; ++i) {
    uint32_t block_index = blockDim.x * i + threadIdx.x;
    uint32_t index = block_offset + block_index;
    if (index < d_keys_size) {  // check that we're in bounds
        // get b bits corresponding to the current iteration
        uint32_t key = d_keys[index];
        uint32_t rank = (key >> nth_bits) & class_range;
        flag_arrays[rank * width + block_index] = 1;
        shared_keys[block_index] = key;
    }
}
```

Scanning the flag array

The next step in the first kernel is to scan the flag array.

Main idea Scanning the flag array serves two purposes

- The last column of this array corresponds to the histogram of the input array, where for each rank we get the number of an existing key.
- The value in the scanned array for a given rank and position gives us the relative position of the key to the same key with a different position. (meaning that for two keys of the same value (for nth bits) we can determine which key should come before the other.

Implemenation details To scan each row of the flag array we used the scan (prefix sum) implemented in the cub library.

`flag[rank][position]`

```
typedef cub::BlockScan<uint16_t, 256> BlockScan16;
__shared__ typename BlockScan16::TempStorage temp_storage16;

uint32_t stride = threadIdx.x * elem_pthread;
#pragma unroll
for (int rank = 0; rank < num_classes; ++rank) {
    uint16_t thread_data[4];
    // load the data from shared
    uint32_t offset = (rank * width) + stride;

    #pragma unroll
    for (int i = 0; i < elem_pthread; ++i) {
        int index = offset + i;
        thread_data[i] = flag_arrays[index];
    }
    // scan
    BlockScan16(temp_storage16).InclusiveSum(thread_data, thread_data);

    __syncthreads();  // required (by documentation)

    // write back in shared_memory

    #pragma unroll
    for (int i = 0; i < elem_pthread; ++i) {
        int index = offset + i;
        flag_arrays[index] = thread_data[i];
    }
}
```

Scan histogram

Main idea The scanned histogram will be used for sorting the block locally, but this histogram give us the number of elements in the previous rank for a given rank, therefore the first position (index) in the sorted input array for a given rank.

implementation Same as before, here is the scan from the CUB library is used.

Local sort on block level

Now we have all the data to be able to sort the array locally (block level).

- Knowing for a given key its relative position to a similar key (scanned flag array).
- For a given rank we can know the first position (sorted index) in the input array (scanned histogram).

We just need to iterate through each element and compute their rank. And with their rank and relative position where to place them.

```
for (uint32_t i = 0; i < elem_pthread;
     ++i) {
    uint32_t block_index = blockDim.x * i + threadIdx.x;             //
    uint32_t index = block_offset + block_index;                     // 11
    if (index < d_keys_size) {  // check that we're in bounds
        uint32_t key =
            shared_keys[block_index];  // save keys in shared mem //11
```

```
            uint32_t rank = (key >> nth_bits) & class_range;
            uint32_t number_elem_previous_rank = shared_scan_hist[rank];
            uint32_t nth_element = flag_arrays[rank * width + block_index];
            uint32_t sorted_position =
                nth_element - 1 + number_elem_previous_rank;
            shared_sorted_keys[sorted_position] = key;
        }
    }
```

Writing to global memory in a coalesced fashion

To write back to global memory, in a coalesced fashion we can write in the same manner that we wrote from global memory to shared memory.

1.3.2 Transpose and scan histogram (Kernel 2)

The second step is to scan the per-block histograms to obtain the indices for the scattering operation in kernel 3. This is done in two steps, first, the per-block histogram matrix is transposed and then the transposed matrix is scanned.

Transposing the histogram matrix

Main idea The block histogram matrix is currently storing each block's histogram on a different row. By transposing the matrix, each block histogram will be stored in a column, and each row will store all the number of values for a given rank. As the matrix is stored in row-major order, this allows us to run a scan on the complete matrix represented as a 1-dimensional array in memory.

Implementation To facilitate the implementation of the matrix transposition we chose to extend the size of the histogram matrix to a multiple of 256 (=16x16). The remaining values not populated by the previous kernel are filled with zeroes, as this does not affect the result of a scan with addition. By doing this, we can call the transpose kernel with a fixed block size of 256 threads and we do not need to check for a given element being inbound. To maximize the speed of the kernel, the input data is read in a memory-coalesced fashion. The values are temporarily stored in shared memory at their correct indices, before being written back to global memory in a memory-coalesced fashion.

```
__shared__ uint32_t tile[256];
uint32_t width = hist_size/16;
uint32_t globalId = blockIdx.x * blockDim.x + threadIdx.x;
uint32_t index = (threadIdx.x & 15) << 4 + (threadIdx.x >> 4);

tile[index] = idata[globalId];
__syncthreads();

int globalIdOut = (threadIdx.x/16) * width + blockIdx.x*16 + threadIdx.x&15;

odata[globalIdOut] = tile[threadIdx.x];
```

Scanning the transposed histogram matrix

Main idea The idea is to sort the transposed histogram matrix in order to obtain the indices for scattering the keys in kernel 3. This allows us to read out the indices for a given block and rank from the scanned matrix.

Implementation As per the project description, we are using the cub library's implementation of the exclusive sum (exclusive scan with addition) provided by the Cuda library. This is one of the fastest known implementations of scan and is required for achieving a fast implementation. The block size and number of blocks are dynamically chosen by the library to ensure efficient computation.

```
cub::DeviceScan::ExclusiveSum(d_temp_storage, temp_storage_bytes, // cub mandated variables
                              env->d_hist_transpose, // input array
                              env->d_hist_transpose, // output array
                              env->d_hist_size); // number of elements in the histogram
```

1.3.3 Scatter (Kernel 3)

The third step is to scatter the elements from each block into global memory in a memory-coalesced fashion. The presorting of the elements in kernel 1 allows us to copy consecutive elements with the same rank to consecutive locations in the output array in global memory. This constitutes memory coalesced accesses and significantly improves the performance of the whole algorithm.

Copying the scanned block histogram to shared memory

Main idea We copy the scanned histogram of the block to shared memory. This minimizes the number of reads to global memory in the later steps.

```
uint32_t num_buckets = 1 << bits;
__shared__ uint32_t scanned_hist[16];

if (threadIdx.x < num_buckets) {  // copy histogram corresponding to block
                                  // into shared memory
    scanned_hist[threadIdx.x] =
        hist_rowwise_scanned[num_buckets * blockIdx.x + threadIdx.x];
}
```

Scattering elements from global memory to global memory in memory coalesced fashion

Main idea This part of the kernel scatters the elements into their correct position in global memory.

Implementation For this implementation to work, a key assumption has to be fulfilled. Namely, the arrays have to be presorted on a bloc level according to their rank. As this implementation sorts 4 bits at a time, it gives 16 different radix buckets in which the values can be stored. With 1024 elements sorted by each block, the expected number of identical ranks per block is 64. Because the array is presorted at the block level it allows us to scatter these elements with identical rank to consecutive addresses in global memory, opening the possibility for coalesced writes. To compute the exact index in global memory at a thread level, the scanned transposed block histogram matrix and the scanned block histogram are required. The former allows us to compute the range of indices for the elements of a given rank and block in the output array. The latter allows a given thread to identify which exact index in global memory corresponds to the element it is looking at. The code below illustrates the precise computation done.

```
uint32_t global_offset = blockIdx.x * blockDim.x * elem_pthread;
uint32_t max_bucket = num_buckets - 1;
uint32_t width = hist_size >> bits;

#pragma unroll
for (int i = 0; i < 4; ++i) {  // each thread scatters 4 elements
    uint32_t next_local = i * blockDim.x + threadIdx.x;
    uint32_t next_global = global_offset + next_local;

    if (next_global < key_size) {
        uint32_t element = keys[next_global];
        uint32_t rank = (element >> (it * bits)) & (max_bucket);
        uint32_t number_of_elems_in_previous_ranks_locally =
            scanned_hist[rank];
        int local_rank_offset =
            next_local -
            number_of_elems_in_previous_ranks_locally;  // the nth element
                                                        // with this rank in
```

```
                                                    // our block

            // hist_T_scanned[rank][blockIdx.x]
            int global_rank_index_start =
                hist_T_scanned[rank * width + blockIdx.x];
            int global_index = global_rank_index_start + local_rank_offset;
            output[global_index] = element;
        }
    }
```

1.4 Performance

We did a benchmark on our implementation, CUB, and futhark. We can see the order of magnitudes on the following graph. Our implementation is around 10 times slower on average than CUB.

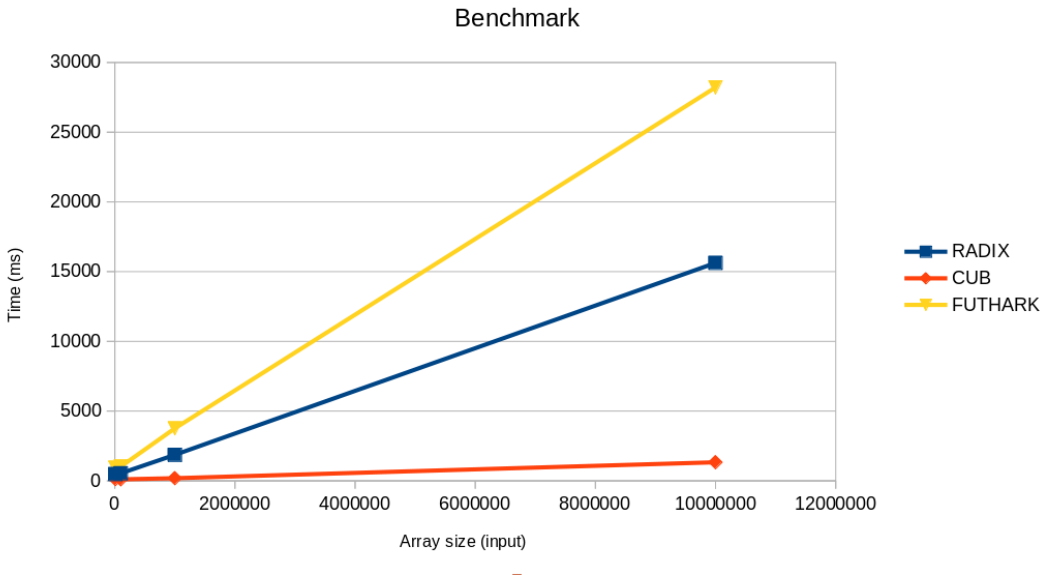


Figure 1.1: Benchmark comparaison

1.5 Limitations and proposed improvements

This section looks at the limitations of this implementation and possible improvements.

Transposing the histogram in kernel 1

Another idea to improve the transpose would be to do it directly in in kernel 1 when writing back the histogram to global memory. This would greatly increase the speed of the algorithm, as it would eliminate the need for the transpose kernel and all its associated memory accesses.

Copying the scanned transposed histogram matrix to shared memory

In kernel 3 this implementation currently reads from the scanned transposed histogram in global memory. This is not optimal, as threads in the same warp could read from different memory locations, slowing down the algorithm. As the matrix is transposed, threads from the same block but with different ranks read from the same column but different rows, meaning that if the matrix is too large, different memory rows have to be read. This could be improved by copying the 16 elements corresponding to the current block to shared memory. This would eliminate unnecessary reads from global memory. We hypothesize that it would only have a small impact on runtime, as the elements in the block are presorted, meaning that consecutive threads will read the same matrix entry with a high probability.

Setting a cap to the number of iterations

In the current implementation, we run the series of 3 kernels 8 times in a row. A possible improvement would be to set a cap to the number of iterations corresponding to the number of bits of the largest number. The largest element can be identified by running a reduce with the maximum operator on the list. This would run in logarithmic time concerning the length of the input. Correctness would be preserved, as the most significant bits would be zeros, and running the kernels over them would not change the result.