University of Copenhagen

Msc in Computer Science

Programming Massively Parallel Hardware

Project Report

Group
Group 10

Authors
Marc Raffy KPF905
Marko Milić DCP353

November 8, 2021

Contents

1 Explanation of simple sorting algorithms **2**
1.1 Radix-sort 2

2 Literature search **3**

3 Our implementation of radix-sort **3**

4 Performance evaluation **4**

5 Difficulties and possible improvements **7**
5.1 Difficulties 7
5.2 Possible improvements 7

1 Explanation of simple sorting algorithms

1.1 Radix-sort

Radix sort is a non-comparative sorting algorithm. This means that it sorts elements without directly comparing them to each other. The algorithm works such that we put numbers into buckets according to the value of the corresponding radix. This is repeated for each significant digit and at each step we preserve the ordering of the previous step. If one wishes to implement this algorithm with parallel basic blocks he could do it the following way (we use Futhark syntax and we sort numbers depending on their binary representation).

```
1  let radix_sort_step [n] (xs: [n]u32) (b: i32): [n]u32 =
2      let bits = map (\x -> (i32.u32 (x >> u32.i32 b)) & 1) xs
3      let bits_neg = map (1-) bits
4      let offs = reduce (+) 0 bits_neg
5      let idxs0 = map2 (*) bits_neg (scan (+) 0 bits_neg)
6      let idxs1 = map2 (*) bits (map (+offs) (scan (+) 0 bits))
7      let idxs2 = map2 (+) idxs0 idxs1
8      let idxs  = map (\x->x-1) idxs2
9      let xs' = scatter (copy xs) (map i64.i32 idxs) xs
10     in xs'
11 let radix_sort [n] (xs: [n]u32): [n]u32 =
12     loop xs for i < 32 do radix_sort_step xs i
```

[1] Let's explain step by step how this code works: Lines 2 and 3 (bits and bits_neg) simply tell us whether the binary radix of the numbers are 0 or 1 since as we mentioned before we work with radix in base 2 not in base 10 in this algorithm.
Then offs is just the offset that will be used for idxs1.
Lines 5 and 6 simply sort the numbers depending on the value of their radix. First we have the smallest, the 0-radix on idxs0 line and then we have the 1-radix on idxs1 line. Note that the idxs1 are obviously offset by the max of idxs0.
Then we simply merge those 2 arrays to have an array consisting of the sorted indexes. We deduce 1 to this array to have a clean 0-indexing.
Finally we do a scatter that will reorder the input array given the new indexes.
This is performed 32 times which corresponds to the size of an integer and the result is the sorted array.

The work complexity is O(k*n) with k the number of bits in each element and n the number of elements. This is because we loop k times and within each loop we do a sequence of operations with the largest complexity of an operation being O(n).
The span of the algorithm is O(k*log(n)) with k the number of bits in each element and n the number of elements. This is because k loop cannot be flattened and within each iteration of the loop the largest span is reduce with a span of O(log(n)).

[1] Code from futhark documentation `https://futhark-lang.org/examples/radix-sort.html`

2 Literature search

At first we started, as recommended, to look at this paper [2]. However we had a hard time understanding how the described radix-sort should be implemented so we tried to look elsewhere.
We looked also in other possible GPU implemented algorithms, where we came across different papers that present sorting algorithms. Some that we saw and checked the outcomes on the GPU are: Bitonic Sort [1] (different implementation of it[2]), Merge Sort[4] which is mentioned in the first paper, Quick Sort[6], Sample Sort[5]. We also found a paper[3] that was comparing the sorts and its run times. This paper was very useful for later measurements that we performed.

We eventually looked back at radix sort and came across this paper [7] which was from our perspective much easier to understand. This paper provides some pseudo code but we did not followed it that much. Instead we chose to focus on what we understood the best, the Figure 2 in the paper. Indeed, this figure explains step by step what should be done. With the help of this repository[3], especially on how to handle memory accesses and synchronization, we ended up having a working implementation of the radix-sort. Our algorithm can be summarized the following way:

3 Our implementation of radix-sort

```
BLOCK 1
for radix between 0 - 16:
    create mask corresponding to radix
    exclusive scan on the mask of this radix
    get the sum of occurrences of the current radix
    exclusive scan on the mask of all radix
    create the histogram with the sum of occurrences of radixes
    compute new indices (not final) based on the values of the mask scan
   + histogram

on global level:
    scan-block-sum = exclusive scan on block_su array from BLOCK 1(arrays
    corresponding to frequency of each radix in each block)

BLOCK 2
 Shuffle input data based on the following index:
 prefix-sum[id] (scan on the scan of mask) + scan-block-sum[id]
```

This algorithm solves the issue about only using the atomic add to make the histogram. Indeed, with the solution first suggested by Cosmin we had the issue that the prefix array

[2] https://github.com/diku-dk/pmph-e2021-pub/blob/master/group-projects/sorting-on-gpu/paper-eff-sort-gpu.pdf

[3] https://github.com/mark-poscablo/gpu-radix-sort

was not ordered anymore. To solve this we first (for each radix) create a mask and then we scan this mask. This way we make sure that we preserve ordering.

The way our algorithm works is that each thread looks at 4 bits at a time. This means that each thread has a sequential loop of length 16. For each radix we merge in shared memory the scan of the mask. We also compute the histogram of each radix. Then we perform an exclusive scan on the histogram and we order the input array at block level with new indexes being histogram (to sort between different radixes) plus prefix sum (being the offset inside a designated radix so we can keep ordering of the previous step of the radix sort).
Once this kernel is done we perform an exclusive scan on all frequencies of each radixes of each block. We call it scan block sum Finally we can reorder the input array with new indices beings: scan block sum + prefixes sum.

In terms of work complexity we have have the same work complexity as in Futhark: $O(n*k)$. This complexity can be seen as linear for a fixed data type.

4 Performance evaluation

For the performance evaluation we tested our CUDA Implementation, CUB, Futhark and sequential sort from the standard library in C++.
We generated random number data in the size of 2^{17} to 2^{30}. The range of generated data was $0..2^{17}$-2^{30}
On the plot measurement we observed that the futhark was not able to calculate the values from range of 2^{24}, because of that the comparison of Futhark sort with that others is done on smaller input size.
First let's compare CUB radix-sort and our own implementation.

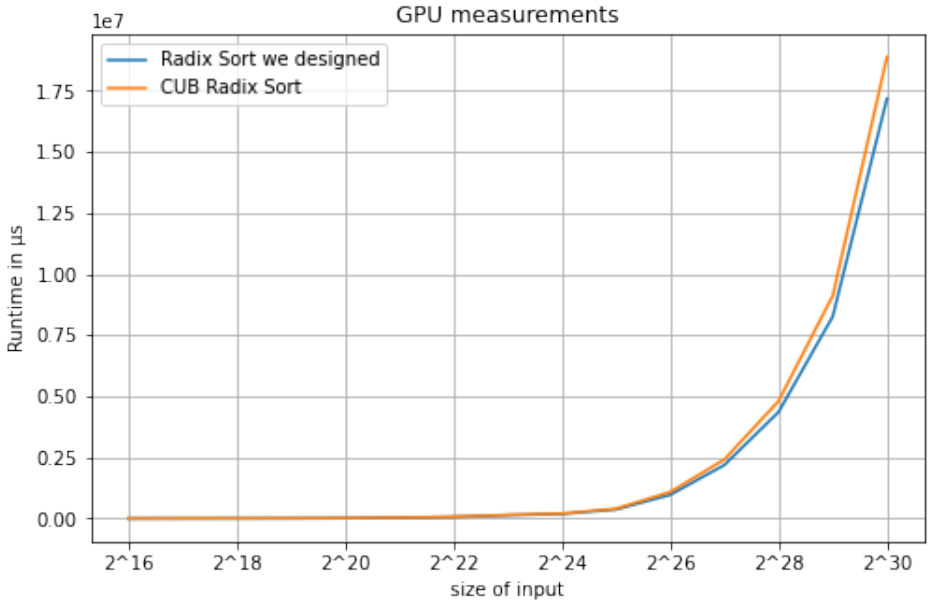

Figure 1: GPU radix-sort on randomly ordered array

First note that the x-axis has an exponential scale which is why the curves appear exponential and not linear. We can see that our implementation is slightly faster than CUB. It is unexpected however there might be an explanation. Indeed, it seems that our implementation is not fully stable on large arrays while CUB appears to be totally stable.

We can now compare those 2 similar results to the other 2 algorithms, the std::sort and the futhrak sort. It gives the results displayed in Figure 2 (randomly ordered array) and Figure 3(descending ordered array)

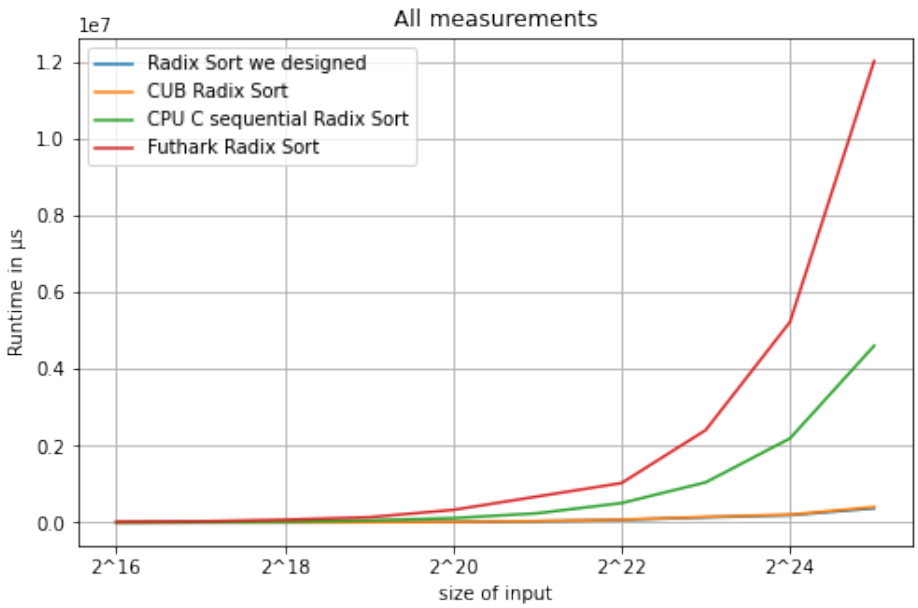

Figure 2: Different sorting algorithms on randomly ordered array

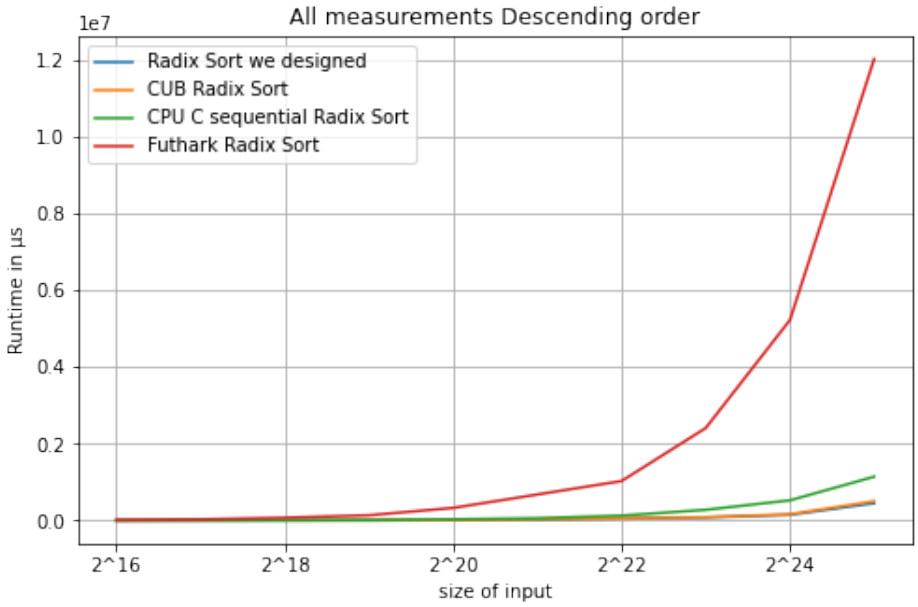

Figure 3: Different sorting algorithms on descending ordered array

We can see that for large randomly ordered arrays, the CPU runtime is 10x larger than GPU runtime which is a huge speedup (it is only 5x for a 2^16 sized array). The difference is even larger with the futhark sort being up to 20x slower than the GPU radix-sort.

For descending ordered arrays things are a bit different. Futhark and the GPU radix-sorts are similar (slightly faster) however std::sort is much faster for this case. After having compared all these algorithms we can conclude that our implementation is quite good. Indeed we out-speed CUB at the cost of a small instability. Additionally, as expected radix-sort on GPU are much faster than the sequential std::sort(), even though the difference is thinner when the input array is sorted in descending order. The Futhark implementation is the slowest of all by a large margin

5 Difficulties and possible improvements

5.1 Difficulties

As mentioned previously the biggest difficulty was to understand the paper about radix-sort provided with the project. We eventually understood it but only at the end of the project when we already had a working version based on a different paper. However we were able to confirm that our algorithm was similar to what was described in the paper.
The other big difficulty was memory management in CUDA. Indeed, once we had the high level idea of the algorithm it was hard to actually write correct CUDA code without errors.

5.2 Possible improvements

The main improvement that could be done would be to extend this algorithm to other data types. It would be easy to implement floats since we can sort them the same way as integers however sorting arrays containing negative numbers would be more challenging and we did not have time to focus on this.

References

[1] Batcher, Kenneth E, *Sorting networks and their applications*, Proceedings of the April 30–May 2, 1968, spring joint computer conference, 307-314, 1968

[2] Peters, Hagen and Schulz-Hildebrandt, Ole and Luttenberger, Norbert, *Fast in-place, comparison-based sorting with CUDA: a study with bitonic sort*, Concurrency and Computation: Practice and Experience, 681-693, 2011, Wiley Online Library

[3] Bozidar, Darko and Dobravec, Tomaz, *Comparison of parallel sorting algorithms*, arXiv preprint arXiv:1511.03404, 2015

[4] Satish, Nadathur and Harris, Mark and Garland, Michael, *Designing efficient sorting algorithms for manycore GPUs*, 2009 IEEE International Symposium on Parallel & Distributed Processing, 1-10, 2009, IEEE

[5] Dehne, Frank and Zaboli, Hamidreza *Deterministic sample sort for GPUs*, Parallel Processing Letters, 03, 2012, World Scientific

[6] Cederman, Daniel and Tsigas, Philippas, *Gpu-quicksort: A practical quicksort algorithm for graphics processors*, Journal of Experimental Algorithmics (JEA), 1–4, 2010, ACM New York, NY, USA

[7] Ha, J and Kruger, Jens and Silva, Claudio T, *Implicit radix sorting on gpus*, GPU GEMS, 2010