Implementation of Fast Sorting Algorithms for GPU

The task of this project is to implement an efficient (fast) sorting algorithm for GPU execution.

1. For this, you will need to do first a literature search, for example, to understand and explain the parallel implementation of radix-sort (and perhaps merge sort) and to discuss how it (they) can be implemented from parallel basic blocks, such as map, reduce, scan, scatter, and what is the work and depth of such implementation.
2. Please note, that you do not have to implement them yourselves, for example because they are already implemented in Futhark package "github.com/diku-dk/sorts". You may get a package by the following simple command:

   ```
   $ futhark pkg add github.com/diku-dk/sorts
   ```

   followed by

   ```
   $ futhark pkg sync
   ```

   You can then write some short Futhark programs---which simply call the library---that will serve as a baseline for comparison with your low-level CUDA implementation, which should be significantly faster, because the current Futhark implementations are quite inefficient.
3. The third step is to do a literature search to find what sorting algorithms have been found to be most suited for GPU execution. You will find two such papers saved in the corresponding folder of this project, and even a slide presentation. If the paper is unclear, you may always come to discuss it with Cosmin, and/or you can run your own literature search.
4. Implement in CUDA at least one of the fast sorting algorithms for GPU, and evaluate its performance in comparison with the Futhark baseline and with the implementation provided by the CUB library (written in CUDA). A sample CUDA program that uses the CUB library is also provided in the folder associated with this project. The radix sort of CUB library seems to be much faster than Futhark's libraries, i.e., one-to-two orders of magnitude faster. Hopefully your implementation will be competitive with the one from CUB.
5. If time remains, you may try to generalize your implementation to work with any datatype, for example, single/double precision int or floats, or even tuples. For this, you may take inspiration from how this was achieved in the "pbb" library that you have helped implement in weekly-2.
6. Finally, please provide a detailed performance evaluation on multiple datasets of various lengths (and potentially different datatypes), and at least compare the performance with the ones of the Futhark baselines and with the CUB library.

7. Please write a tidy report in which you:
    1. Start from explaining at least the radix sort algorithm, its assumptions, and its parallel work-depth complexity.
    2. Then move to presenting in detail---including high-level pseudocode and written explanation---the (fast) algorithm that you have chosen to implement. Reason (if possible) about its work-depth asymptotic, but also about why do you expect your algorithm to run efficiently on GPU hardware in practice (i.e., smaller constants).
    3. If the algorithm from step 2 was presented at a high-level, you would probably like to present in more detail your optimized Cuda implementation – in both pseudocode and text. Here you can try to explain (reason about)
        - what basic blocks of parallel programming you have used – e.g., a reduce/scan/scatter at what level of hardware parallelism, e.g., global level, or at Cuda block level,
        - whether the performance depends on Cuda features that are not common tools of data-parallel programming, e.g., atomicAdds in which you use the previous result,
        - the properties of your solution: is it a stable sort, is it deterministic
    4. Present a systematic performance evaluation in which you compare on datasets of different lengths (and possibly different array element types) the performance of your CUDA implementation in comparison with the slow Futhark baselines and with the fast CUB library.

HINT: Possible Structure of an efficient Radix Sort (combination of the ideas in the papers and NVIDIA presentation)

Notation: **B** denotes the CUDA block size, **Q** denotes the number of elements processed by each thread, **lgH** denotes the number of bits sorted at a time, and **H** = 2^lgH. Possible good values are: B = 256, Q=22, lgH=8. The computation is split into several kernels:

First kernel: each block processes Q*B elements and produces a histogram of length H. A bin in the histogram corresponds to a configuration of the current lgH bits of interest, hence the histogram records how many elements are in the corresponding block for each bit configuration.

The result of the first kernel is 2D array, i.e., an array of number-of-blocks histograms of length H. ***This array is transposed and then flattened, and then scanned.***

The last kernel is the challenging one in terms of extracting performance. It uses the same grid-block structure as the first kernel. It has the following computational steps:

1. copy from glb to shared to register memory the Q*B elements processed by each block
2. Inside a loop of size **lgH** write the code for two-way partitioning of the Q*B elements according to the corresponding bit. This should be semantically equivalent with partition2 applied to the predicate that results in true if the corresponding bit is unset and false if it is set. The code should be efficiently sequentialized and data should be hold mostly in register memory and only transiently in shared memory. For example, a scan can be implemented by three stages: (i) each thread sequentially reduces its Q (register) elements, and stores the result to shared memory, then (ii) a parallel block-level scan is performed (by all threads in the block), and finally, (iii) each thread sequentially scans its chunk of elements and applies the corresponding prefix.
3. After the loop:
    1. Copy the original and scanned histograms from global to shared memory
    2. scan in place the original histogram corresponding to this block
    3. each threads writes its Q elements to their final positions in the global-memory array by using the info in the histograms.