Flattening a Batch of Rank-Search-k Problems

This project refers to the O(n) selection or rank search algorithm, that computes the k-th smallest element of an array. More precisely, we assume that we have a batch of such problems, in which the array sizes might be regular or irregular, and we would like to come up with an efficient GPU implementation, which probably requires flattening the two-level parallelism.

The motivation for this problem is that many machine-learning algorithms requires fast computation of the median element. A "naive" way of achieving that is by applying a GPU-efficient sorting algorithm, such as CUB's implementation of radix sort, and then selecting the element at mid index as the median. This project aims to investigate whether a parallel (flattened) implementation of rank-search-k problem can possibly beat the naive approach by a significant margin on GPUs.

Please note that your grade will not necessarily suffer if you do not manage to beat it. This is an exploratory project, i.e., we do not know (yet) whether this is a feasible way for finding the median on GPUs.

The following Futhark-like pseudo-code of rank-search-k, instantiated for an array of float values for simplicity, does not yet achieve the linear complexity, but does a good job of high-lighting the parallel structure of the algorithm: notice the use of **filter** and **recursion** inside **rankSearch**, and the outer **map2** inside the **main** function that solves a number of rank-search-k problems:

```
let rankSearch (k: i64) (A: []f32) : f32 =
  let p = random_element A
  let A_lth_p = filter (< p) A
  let A_eqt_p = filter (==p) A
  let A_gth_p = filter (> p) A

  if (k <= A_lth_p.length)
  then rankSearch k A_lth_p
  else if (k <= A_lth_p.length + A_eqt_p.length)
       then p
       else rankSearch (k - A_lth_p.length - A_gth_p.length) A_gth_p

let main [m] (ks: [m]i64) (As: [m][]f32) : [m]f32 =
  map2 rankSearch ks As
```

Please notice that this is not legal Futhark code:

(1) first because Futhark does not supports recursion, and

(2) second because the array As is supposed to be an irregular array of arrays---hence it should probably be represented as a shape and flat data.

Furthermore, please notice that the implementation is somewhat similar to quicksort, but with the major difference that it has ***only one*** direct recursive call that gets executed (either in the **then** or **else** branch) ***rather than two*** direct recursive calls in quicksort. The other difference is that efficient flattening of rank-search will probably require you to utilize the rule for flattening **if-then-else** (while for quicksort we have used that dirty shortcut that was checking whether the flat array was entirely sorted as the stopping condition of the while loop).

This simplified implementation is good enough to show the parallel structure, but does not guarantees O(n) work complexity. A very quick google search has revealed several very accessible links that present the full O(N) algorithm:

Wikipedia link (https://en.wikipedia.org/wiki/Selection_algorithm)

cs.clemson.edu (https://people.cs.clemson.edu/~goddard/texts/algor/A5.pdf)

presentation at cs.princeton.edu (https://www.cs.princeton.edu/~wayne/cs423/lectures/selection-4up.pdf)

https://tildesites.bowdoin.edu/~ltoma/teaching/cs231/2018spring/Lectures/selection.pdf

The link to the scientific paper introducing the algorithm is given below (but it is not a requirement):

"Time Boundsfor Selection", Blum, Floys, Pratt, Rivest, Tarjan

Please feel free to do your own literature search on the matter: perhaps you will discover a paper presenting an efficient GPU implementation of rank-search---you may then reproduce that. We have not checked for such references; these are part of your job.

The goal of the project is to flatten as efficiently as possible the nested parallelism of the batch rank-search-k problem, and then to compare its performance with the naive approach that applies a sorting algorithm and then selects the k'th element of the sorted array. For example, one can use the radix-sort implementation from CUB library, which is included in the tar.gz archive.

Key things that your report should contain are:

1. A section that presents at high level the O(N) algorithm, and maybe summarizes at the end a Futhark-like pseudo-code with nested irregular parallelism and recursion.
2. A section that presents how the batch rank-search-k problem is translated to Futhark (by flattening the nested-parallelism of the high-level pseudo-code of step 1). Flattening is probably the (only) tool by which you can achieve coalesced access to (the GPU) global memory, which

we speculate, is probably a necessary ingredient for achieving a performant GPU implementation.

3. A systematic experimental evaluation that compares your Futhark implementation with the "naive" approach of sorting with an efficient GPU implementation (e.g., CUB).
4. We suggest that you start by sorting unsigned integers (or floats), and after you make it work, if time permits, you may generalized your implementation to work with whatever datatype (for the array element). You may take inspiration of how this can be achieved by looking at the radix-sort implementation of the Futhark sorting library.
5. If time permits, it might prove useful to also present a literature survey of various implementation of rank-search-like algorithms, perhaps aimed at GPU execution.

We make available the CUB library version 1.8.0, together with a sample program that uses this library to sort an array.

You may get the Futhark package by the following simple command:

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

followed by

```
$ futhark pkg sync
```