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 sorting approach on GPUs. In particular, the current sorting algorithms written in Futhark are one-to-two orders of magnitude slower than CUB’s sorting, so even if the rank-k search is not more efficient than CUB’s sort, it might be useful to have one in Futhark.

Please note that your grade will not necessarily suffer if you do not manage to beat CUB’s sorting. We have a Futhark implementation that is comparable with CUB’s sorting, but does not outperform it, i.e., in some cases it is faster and in other cases slower.

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. Please notice (i) the use of **filter** as innermost parallelism, (ii) the use of **tail recursion** inside **rankSearch**, and (iii) the use of **map2** inside the **main** function that solves a number of (irregular) rank-search-k problems. Also note that the numbering for k values in the algorithm below starts from 1 and not from zero, i.e., to search the smallest element one calls rankSearch with k=1.

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

Please notice that the implementation 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 come up with a solution for flattening the **if-then-else;** such a rule was not discussed in the PMPH course.

This simplified implementation is good enough to show the parallel structure, but does not guarantees O(n) work complexity. For the purpose of this project, you do not need to implement a O(n) algorithm, i.e., the focus is on flattening the algorithm presented above; if you can implement an O(n) work parallel algorithm, that is definitely bonus!

If interested, for completeness, 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 available in the folder `group-projects/cub-code`.

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. With respect to flattening, you may derive
   a) a version that is obtained by "think-as-a-compiler" flattening, i.e., faithfully apply flattening rules or derive new ones (in the spirit of what we learned).
   b) another version that uses human reasoning to optimize some algorithmic inefficiencies.
4. A systematic experimental evaluation that compares your Futhark implementation with (i) the "naive" sorting approach but which uses an efficient GPU implementation, such as CUB's; this is a **must-do** comparison, and (ii) whatever else you think it might be relevant.
5. 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).
6. 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
```

Hint: structure of code version using human reasoning to improve flattening.

The key observation is that the flattened version does not have to move around the elements of the original array such that the ones that are less than the pivot come first, followed by the ones that are equal to the pivot, followed by the one that are greater to the pivot. (This step is very computationally expensive.) Instead, what is essential for the algorithm is to compute ***how many elements*** are in each class. Once we have that we can determine for each subproblem on which branch to "recurse", and we can eliminate the non-useful elements with a global filter.

The pseudocode below uses the array denoted II1, which is intended to record for each element of a flat array the index of the subarray to which the current element belongs to. For example, the II1 array for an array of shape `shp = [3,2,4]` is `II1 = [0,0,0,1,1,2,2,2,2]`.

```
def rankSearchBatch (ks: [m]i32) (shp: [m]i32) (II1: *[n]i32) (A:
[n]f32) : *[m]f32 =
  let result = replicate m 0f32
  let (_,_,_,_,result) =
    loop (ks: [m]i32, shp: [m]i32, II1, A, result)
    while (length A > 0) do
      1. compute the pivot for each subproblem, e.g., by choosing the
         last element. This is a small parallel operation of size m.

      2. for each subproblem compute the number of elements less than
         or equal to the pivot. This is a large-parallel operation of
         size n. Hint: use a histogram or reduce_by_index construct.

      3. Use a small-parallel operation of size m to compute:
         3.1 kinds → the kind of each subproblem, e.g.,
              (a) -1 means that this subproblem was already solved
              (b) 0  means that it should recurse in “< pivot” dir
              (c) 1  means that the base case was reached
              (d) 2  means that it should recurse in “> pivot” dir
         3.2 shp’ → the new shape after this iteration, e.g., if
                    we just discovered kinds==1 for some subproblem
                    then we should set the corresponding element of
                    shp’ to zero.
         3.3 ks’  → the new value of k for each subproblem
                    (the inactive subproblems may use -1 or similar)

      4. write to result the solutions of the subproblems that have
         just finished (have kinds 1)

      5. filter the A and II1 arrays to contain only the elements of
         interest of the subproblems that are still active.

  in  result
```