Weekly Assignment 3
Parallel Functional Programming

Troels Henriksen and Cosmin Oancea
DIKU, University of Copenhagen

December 2023

Introduction

The handin deadline is the 14th of December.

The handin is expected to consist of a report in either plain text or PDF file (the latter is recommended unless you know how to perform sensible line wrapping) of 4—5 pages, excluding any figures, along with an archive containing your source code. The report should contain instructions on how to run and benchmark your code.

Task 1: Flattening with the New, More Efficient Rules

This refers to flattening the following nested parallel code:

```
def nestedPar [m] (ass: [m][]f32) (bss: [m][]f32)
                  (cs: [m]f32) (inds: [m]i64) : [m]32 =
  map4 (\ as bs c ind ->
          let f a = (f32.sqrt a) * bs[ind] + c
          let tmp1 = map f as
          let ioti = iota (length as)
          let iotf = map f32.i64 ioti
          let tmp2 = map2 (+) tmp1 iotf
          in  reduce f32.max f32.lowest tmp2
       ) ass bss cs inds
```

Please implement function `optimII1Ker` in the handed-out code: `optim-II1-flat-sols/optimize-flat-II1.fut`, and use the provided infrastructure (datasets) for validation and performance measurements.

Once you are in folder `optim-II1-flat-sols` you may create the datasets:

```
$ make datasets
```

and you may run the whole suite with

```
$ make run
```

Function `classicKer` flattens the code above by utilizing the old/inneficient rules of flattening, e.g., that manifest in memory the replication of free variables and the segmented iotas. They validate, but the performance can be significantly improved.

Your first task is to implement function `optimII1Ker` that performs the flattening of the code above by using the “new” more-efficient rules, which for example rely on the $\texttt{II}^1$ array to avoid manifestation of the replicated and iota arrays.

Your report should contain:

- a short statement related to whether your code validates or not
- the implementation of the ‘optimII1Ker‘ function
- the runtimes of both entry-points on the provided datasets
- the speedup of your implementation in comparison with the “classic” approach.

You should handin the same files you received, but now benefitting by your correct and efficient implementation. Of course, currently, the `optimII1` entrypoint does not validate, because it awaits your implementation. As well, please add a small reference input and output dataset for validation, that you build by hand.

Hint: Implementing the segmented reduce by means of

- a histogram results in best performance on the first four datasets (speedups between $2.6 - 4\times$ vs classic on A100);
- a segments scan followed by a gather results in best performance on the fifth provide dataset (speedups between $1.8 - 1.9\times$ vs classical on all five datasets on A100).
- if you wish, you may implement both versions, but not required; if you do then please add a new entrypoint.

Task 2: Flattening Rule for Scatter inside Map (Pen and Paper)

The lecture slides `L5-irreg-flattening.pdf` have presented many flattening rules, but there is none that handles a segmented scatter, i.e., a `scatter` nested inside a `map`.

This was intentionally left for you to implement: your task is to write a rewrite rule for the code below (that of course produces flat-parallel code):

```
map (\xs is vs -> scatter xs is vs
    ) xss iss vss
```

This is a pen and paper exercise, so describe it in your report. You may assume that `xss`, `iss` and `vss` are two-dimensional irregular arrays whose shape and flat data representation are known. (From the semantics of `scatter` it also follows that the shape of `iss` is the same as the shape of `vss` but may be different than the shape of `xss`).

You will probably have to use this rule in the implementation of `partition2` lifted.

Task 3: Flattening Rule for Histogram inside Map (Pen and Paper; very similar with the previous task)

The lecture slides `L5-irreg-flattening.pdf` have presented many flattening rules, but there is none that handles a segmented reduce-by-index, i.e., a `reduce-by-index` nested inside a `map`.

This was intentionally left for you to implement: your task is to write a rewrite rule for the code below (that of course produces flat-parallel code):

```
map (\histo is vs -> reduce_by_index (⊙) 0⊙ histo is vs
    ) histos iss vss
```

This is a pen and paper exercise, so describe it in your report. You may assume that `histos`, `iss` and `vss` are two-dimensional irregular arrays whose shape and flat data representation are known. (From the semantics of `reduce_by_index` it also follows that the shape of `iss` is the same as the shape of `vss` but may be different than the shape of `histos`).

Task 4: Implement the lifted version of `partition2`

The file `quicksort-flat.fut` implements the flat-parallel code for quicksort, but the implementation is incomplete.

Your task is to implement function `partition2L`, which is the lifted version of function `partition2` (provided).

Please see the comments in file `quicksort-flat.fut` and the relevant slides from `L5and6-irreg-flattening.pdf` (pertaining "Flattening Quicksort" section towards the end). To Do:

- Once the implementation of quicksort validates on the small dataset provided in `quicksort-flat.fut`, please add a large dataset to the benchmarking infrastructure, e.g., a random one consisting of tens-to-hundreds million floats. Please report runtime and speedup versus the sequential version (e.g., `futhark bench --backend=cuda` *vs* `futhark bench --backend=c`).
- Show your implementation of `partition2L` in your report and describe
  - for each line (segment) in `partition2`, what rule have been used to flatten, and what is the corresponding code in `partition2L`.
- This task is challenging so the focus is in deriving correct flat code that conforms with the word-depth asymptotics of $\texttt{partition2}^L$, i.e., we are not keen on performance.

Please note that, if you have a bug in your code, then it is likely that the futhark program will run forever because the outer sequential loop (of quicksort) ends only when the array gets sorted (and if you have a bug it might never be).