Weekly Assignment 2
Parallel Functional Programming

Troels Henriksen and Cosmin Oancea
DIKU, University of Copenhagen

November 2020

Introduction

This weekly assignment focuses on advanced parallel programming, with a particular focus on flattening.

The handin deadline is the 3rd 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—6 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: Matrix Inversion

An $n \times n$ square matrix A is said to be *invertible* if there exists a unique matrix A^{-1} such that

$$AA^{-1} = I_n$$

where I_n is the $n \times n$ identity matrix. We also call such an invertible matrix A *non-singular*, and is characterised by having a determinant $|A|$ different from 0. Computing A^{-1} is called *matrix inversion*.

For this exercise you will be implementing matrix inversion based on the Gauss-Jordan algorithm (without pivoting, for simplicity).

In the Gauss-Jordan algorithm, we augment A with I_n to the right, forming an $n \times 2n$ matrix typically written $[A|I_n]$. We then perform Gaussian elimination on $[A|I_n]$ to compute the reduced row echelon form, which produces a matrix $[B|A^{-1}]$, from which we can then extract the desired A^{-1}.

For this task, you are given a function `gaussian_elimination` for performing Gaussian elimination (without pivoting). It's a simple (bad) implementation that is not numerically stable, but it will do for our purposes. Your task is to

1. Implement `matrix_inverse`: augment the input with the identity matrix, call `gaussian_elimination`, then extract the inverted matrix. **Hint:** in order to satisfy the type checker, you will likely have to define a variable `let n2 = n + n` at the start of `matrix_inverse` and then use this `n2` with either `concat_to` or in a `replicate` followed by in-place updates.

2. Write a `main` function that maps `matrix_inverse` across an array of k square matrices. That is, the input to `main` must have type `[k][n][n]f32`, for some `k` and `n`.

3. Answer: how many levels of parallelism does this program have? Based on the incremental flattening rules, approximately how many versions will be generated?

4. Using a GPU backend (`cuda` or `opencl`), benchmark your program on a range of inputs. Don't worry about whether the input matrices are in fact invertible. Include cases with many small matrices ($n \leq 16$).

5. Use `futhark autotune` to optimise the threshold parameters. How does this affect performance for each of your datasets? Why do you think that is?

Task 2: Flattening an If-Then-Else Nested Inside a Map

Please read the Rule 8 of Flattening at slides 34−35 of lecture `L4-irreg-flattening.pdf`.

Your task is to implement the `flatIf` function in file `flat-if-then-else.fut`, such that `flatIf` is the (flat-parallel) code resulted from flattening the following program:

```
map (\b xs -> if b then map f xs
                   else map g xs
    ) bs xss
```

Please see the comments in `flat-if-then-else.fut`.

Once the provided dataset validates, you should:

- make at least one other smallish dataset and check that it validates (size of the array should be in tens-of-elements range)
- create a large dataset (tens-of-million elements range) and report the speedup with respect to a sequential implementation. The latter could be the one compiled with `futhark c` or you may implement an optimized sequential one in a different program – your choice.

Task 3: Flattening Rule for Scatter inside Map (Pen and Paper)

The lecture slides `L4-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 the next task.

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 `L4-irreg-flattening.pdf` (pertaining "Flattening Quicksort" section). To Do:

- Once the implementation of quicksort validates on the small dataset provided in `quicksort-flat.fut`, run the program on a large dataset, e.g., ten millions floats, and report runtime and speedup versus the sequential version (i.e., `futhark opencl` *vs* `futhark c`). Testing bigger datasets can be achieved with a command such as:

  ```
  futhark dataset -b --f32-bounds=-1000000.0:1000000.0 -g
  [10000000]f32 | ./quicksort-flat -t /dev/stderr -r 3 >
  /dev/null
  ```

  Or you can use `futhark bench` as in first weekly.

- Show your implementation of `partition2L` in your report and describe
  - for each line in `partition2`, what rule have been used to flatten, and what is the corresponding code in `partition2L`.