Exercises in Futhark for Parallel Functional Programming at Chalmers

Troels Henriksen
based on work by Martin Elsman
DIKU, University of Copenhagen

April 2023

Introduction

This lab assignment aims at exercising the use of Futhark for writing parallel programs in a functional setting. The exercises assume access to a computer with Futhark installed. For information about installing Futhark, please consult `https://futhark-lang.org`.

1 Getting started

This exercise has two goals:

1. Illustrating how simple parallel problems can be expressed in Futhark.
2. Showing how to test and benchmark a Futhark function.

Exercise 1.1: Write a function

Create a Futhark function called `process` of the following type:

```
val process [n] : (xs: [n]i32) -> (ys: [n]i32) -> i32
```

Where `process x y` computes the maximum absolute difference (pointwise) between the arrays using parallel operations. The function should return the value 0 if two empty arrays are passed to the function.

Hint: Use `map`/`map2` and `reduce`. Do not use `loop`.

If you write your function in a file `solutions.fut`, you can load it into `futhark repl` and try it as follows:

```
> :load solutions.fut
> process [23,45,-23] [-2,3,4]
42
```

Exercise 1.2: Test your function

While `futhark repl` is convenient, it is not a replacement for proper testing. Instead we will use the `futhark test` tool.

Follow the instructions in the Futhark book[1] and add a test block that uses the following two input arrays:

```
[23,45,-23,44,23,54,23,12,34,54,7,2, 4,67]
[-2, 3,  4,57,34, 2, 5,56,56, 3,3,5,77,89]
```

Hint: Remember to define `process` with `entry` rather than `def`.

Test that your function gives the expected result.

Hint: The `futhark test` manpage may be useful reading[2].

Exercise 1.3: Benchmark your function

Follow the guidelines given in the Futhark book on benchmarking[3] and use `futhark bench` to benchmark your program.

In order to benchmarking to be interesting, you must use inputs of non-trivial size. Use the `futhark dataset` tool to generate seven sets of test data of different lengths. Each set should contain two one-dimensional `i32` arrays, each containing integers in the range $[-10000; 10000]$. The array lengths for the seven different sets should be 100, 1000, 10000, 100000, 1000000, 5000000, and 10000000.

Hint: The notation for referencing data file in test blocks is explained in the Futhark Book[4].

Benchmark your function using the `c` and `multicore` backends, as well as either `cuda` or `opencl` if you have access to a GPU system.

Map the timings (in microseconds) onto a chart and remember to describe the system on which you're benchmarking.

[1] https://futhark-book.readthedocs.io/en/latest/practical-matters.html#testing-and-debugging

[2] https://futhark.readthedocs.io/en/latest/man/futhark-test.html

[3] https://futhark-book.readthedocs.io/en/latest/practical-matters.html#benchmarking

[4] https://futhark-book.readthedocs.io/en/latest/practical-matters.html#external-data-files

Hint: The `futhark bench` manpage may be useful reading [5].

Exercise 1.4: Extend your function

Create an extended version of `process`, called `process_idx`:

```
process_idx [n] : (xs: [n]i32) -> (ys: [n]i32) -> (i32, i64)
```

This function also returns the index (of type `i64`) of the pair for which the maximum absolute difference is found.

Test and benchmark `process_idx` similarly to `process`.

Hint: The lecture slides should give you a hint to solving this problem.

2 Segmented operations

Exercise 2.1: Proof of associativity

Assuming $\oplus$ is an associative operator with neutral element `0`, show that `(0,false)` is a left-neutral element of

$$(v_1, f_1) \oplus' (v_2, f_2) = (\texttt{if } f_2 \texttt{ then } v_2 \texttt{ else } v_1 \oplus v_2,\ f_1 \vee f_2)$$

Exercise 2.2: Segmented scans and reductions

The operator in the previous question can be used to implement a segmented scan. Specifically, a `scan` on an array of type `[]t` with operator $\oplus$ and neutral element $0_\oplus$ can be turned into a segmented scan on an array of type `[](t, bool)` with operator $\oplus'$ and neutral element `(`$0_\oplus$`, false)`. A `true` indicates the beginning of a segment, and `false` the continuation of a segment.

Finish the following Futhark definition of segmented scan:

```
def segscan [n] 't (op: t -> t -> t) (ne: t)
                   (arr: [n](t, bool)): *[n]t =
  ...
```

A segmented reduction is more complicated, but can be implemented by first performing a segmented scan, and then making use of `scatter`.

Finish the following Futhark definition of segmented reduction:

```
def segreduce [n] 't (op: t -> t -> t) (ne: t)
                     (arr: [n](t, bool)): *[]t =
  ...
```

[5] `https://futhark.readthedocs.io/en/latest/man/futhark-bench.html`

Note that we cannot provide the size of the returned array in the type, as we do not know the number of segments.

Benchmark the performance of segmented scan versus ordinary scan, and segmented reduce versus ordinary reduction, and show the result.

Exercise 2.3: Implementing generalised histograms

Finish the following implementation of `hist`:

```
def hist 'a [n]
         (op : a -> a -> a) (ne : a)
         (k: i64) (is : [n]i64) (as : [n]a) : [k]a =
  ...
```

Obviously, do not use the built-in `hist` or `reduce_by_index` in your definition. Instead, use `radix_sort_by_key`[6] from the `github.com/diku-dk/sorts` package and the `segreduce` you wrote previously.

Hint: To use the `github.com/diku-dk/sorts` package, run

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

This will create a directory tree `lib/`, from which you can import the sorting function with

```
import "lib/github.com/diku-dk/sorts/radix_sort"
```

Hint: After having sorted the values, you can use the `rotate` function together with `map2` to create a flag vector, which can be used as input to a call to `segreduce`.

What is the asymptotic complexity (work and span) of your implementation? How does it perform compared to Futhark's built-in `hist`?

3 2D Ising Model

The 2D Ising Model is a mathematical modeling of the behaviour of a simple idealised ferromagnet, in which we compute the *spin* (roughly, polarity) of a grid of electrically charged atoms over a period of time. From a Futhark point of view, this grid is a two-dimensional array of integers that are either -1 or 1.

[6] `https://futhark-lang.org/pkgs/github.com/diku-dk/sorts/0.3.3/doc/lib/github.com/diku-dk/sorts/radix_sort.html#4010`

At any given discrete time step, the charge of a spin can be either positive or negative. A spin interacts only with its immediate neighbors, which makes Ising simulation a *stencil*. An atom prefers to have the same polarity as its neighbors, although it also has a small chance to flip polarity randomly, based on the temperature. This is the Monte Carlo aspect.

To update the grid, we compute for each spin c its corresponding *energy gradient* Δ_e, as follows:

$$\Delta_e = 2c(u + d + l + r)$$

where u, d, l, r are the spins directly adjacent to c in the grid (we ignore diagonals). Further, for each spin we compute two random numbers, a and b, in the range $(0, 1)$. *It is very important that each spin receives its own* a, b. Then we compute the new value of c as

$$c' = \begin{cases} -c & \text{if } a < p \wedge (\Delta_e < -\Delta_e \vee b < e^{-\Delta_e \div t}) \\ c & \text{otherwise} \end{cases}$$

where $0 \leq p \leq 1$ is the *sample rate*, and $t \in \mathbb{R}$ is the *temperature*. Put in words, p is the fraction of spins that are candidates for flipping in a given time step. Of these, we flip those where it would locally reduce the energy of the system, or randomly, where the chance of random flips is proportional to the temperature. For more information on Ising models, I recommend the very readable 6-page article *The World in a Spin* by Brian Hayes[7], but we need not understand the physics to implement the model in Futhark.

For this exercise, you will be modifying the, `ising.fut` file in the code handout `ising-handout.tar.gz`, that contains the skeleton for an implementation of the 2D Ising model. Read the existing code carefully and run `futhark pkg sync` to download the needed dependencies.

The code handout also comes with a program, `ising-tui.c`, that permits a console visualisation of the computation. Running the visualisation is not required to solve the assignment; it's just more fun that way. To run the visualisation, do

```
$ make ising-tui
$ ./ising-tui
```

This may look more impressive (and take a longer time to converge) if you make your console font smaller.

[7] `http://bit-player.org/wp-content/extras/bph-publications/AmSci-2000-09-Hayes-Ising.pdf`

Exercise 3.1: Generate initial state

The code handout defines the type of spins as follows:

```
type spin = i8
```

However, we also need to generate (potentially) distinct random numbers for every spin. For this exercise, we will use *one RNG state per spin*. Thus, the function for generating an initial grid state is:

```
entry random_grid (seed: i32) (h: i64) (w: i64)
  : ([h][w]rng_engine.rng, [h][w]spin) =
  ...
```

Where `rng_engine.rng` is the type of RNG states. See comments in the code handout for more information.

Hint: Generate flat arrays of size $n \times m$, then use `unflatten n m`.

Exercise 3.2: Computing Δ_e

This is a stencil where we, for every spin, compute Δ_e of type `i8`:

```
entry deltas [h][w] (spins: [h][w]spin): [h][w]i8 =
  ...
```

One question that must be answered for every stencil is how to handle the edges of the grid, where there are no neighbors. For the 2D Ising model, we pick the easy solution, and use *wraparound*, also known as a *torus world*. Intuitively, when we go over one edge, we arrive on the opposite edge. This can be done with `rotate`. If `xss` is a two-dimensional array, then `rotate 1 xss` corresponds to rotating the array by one element vertically (along the first dimension), while `map (rotate (-1)) xss` rotates it by negative one element horizontally (along the second dimension).

Hint: Use the `map2`/`map3`/`map4`/`map5` functions to map across multiple arrays simultaneously.

Exercise 3.3: The step function

Define the step function, which computes one time step of the simulation:

```
entry step [h][w] (abs_temp: f32) (samplerate: f32)
                  (rngs: [h][w]rng_engine.rng)
                  (spins: [h][w]spin)
  : ([h][w]rng_engine.rng, [h][w]spin) =
  ...
```

Note that it computes not just new spins, but also new RNG states.

Exercise 3.4: Benchmarking

If you have defined the above functions correctly, then you should now be able to use the predefined `main` function, which creates a grid and runs a few steps of the simulation; returning the final grid at the end:

```
def main (abs_temp: f32) (samplerate: f32)
         (w: i64) (h: i64) (n: i32): [h][w]spin =
  (loop (rngs, spins) = random_grid 1337 h w
   for _i < n do
     step abs_temp samplerate rngs spins).1
```

Show benchmarks that demonstrate how sequential versus parallel performance varies for different values of `w`, `h`, and `n`. Explain your results.

Hint: Use `futhark bench` with `--backend` to try different backends; for example `--backend=c`, `--backend=cuda`, or `--backend=multicore`.

Exercise 3.5: Now do it again (optional)

Rewrite the Ising simulation in some other parallel framework or language from the course, and reflect on how Futhark compares in convenience and performance.