

Lab C Parallel Functional Programming

Part 1

Write a tutorial about parallel functional programming, concentrating on a particular language or library. You are free to choose your own title. Check with Mary if you are unsure about your choice. Some possible titles might be

The Par Monad for parallel Haskell programming: a tutorial

A tutorial on Parallel Strategies in Haskell

How to use **par** and **pseq** for parallel programming in Haskell

Parallel functional programming in Java 8: a tutorial

GPU programming in Haskell: a tutorial on Accelerate

Parallel sorting in Haskell: how well can we do?

Parallel programming in F#: a tutorial

Deterministic parallel programming in Erlang

The main point, remember, is to make a clear and simple tutorial (more like a technical blog post than a paper). Good tutorials often contain nice pictures! Give readers some information about sequential and parallel running times. Submit your document or web page with all associated images and code. One option would be to implement an interesting deterministic parallel program in Haskell or some other functional language, and to write a blog post documenting your work in developing and optimising the program. This task might be a good way to start exploring possible Masters thesis topics. It gives you the opportunity to explore one approach to parallel functional programming more deeply. It is important to avoid plagiarism!

Part 2

Exercises in Futhark for Parallel Functional Programming at Chalmers

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

April 2020

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 aims at illustrating how simple parallel problems can be expressed in Futhark.

Exercise 1.1: Write a function

Create a Futhark function called `process` that takes as arguments two one-dimensional `i32` arrays (signals) of the same length and computes the maximum absolute difference (pointwise) between the signals (you should not use Futhark's `loop` construct). The function should return the value 0 if two empty signals are passed to the function.

Consider the following two signals:

```
let s1 = [23,45,-23,44,23,54,23,12,34,54,7,2, 4,67]
let s2 = [-2, 3, 4,57,34, 2, 5,56,56, 3,3,5,77,89]
```

What is the result of calling your function on `s1` and `s2`?

Hint: To run the function, define a `main` function that passes the two arrays to the function `process`, then compile this program with `futhark c`, `futhark opencl`, or `futhark cuda`.

Exercise 1.2: Run your function

Use the `futhark dataset` tool to generate seven sets of test data of different length. Each set should contain a pair of 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.

Run the function `process` with the different data sets and with executables obtained both with using `futhark c` and `futhark opencl` (or `futhark cuda`). Map the timings (in microseconds) onto a chart and remember to specify the system on which you're running the executables. Follow the guidelines given in the Futhark book on benchmarking¹.

Exercise 1.3: Extend your function

Create a refined version of the `process` function, called `process_idx`, that also returns the index of the source signals for which the maximum absolute difference is found.

Report the result of calling `process_idx` on the signals `s1` and `s2`. Show evidence that your solution scales as the `process` function.

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 $(0, \text{false})$ is a left-neutral element of

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

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

¹<https://futhark-book.readthedocs.io/en/latest/practical-matters.html#benchmarking>

neutral element 0_{\oplus} can be turned into a segmented scan on an array of type `[] (t, bool)` with operator \oplus' and neutral element $(0_{\oplus}, \text{false})$. A `true` indicates the beginning of a segment, and `false` the continuation of a segment.

Finish the following Futhark definition of segmented scan:

```
let 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:

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

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 `reduce_by_index`

Finish the following implementation of `reduce_by_index`:

```
let reduce_by_index 'a [m] [n]
    (dest : *[m]a)
    (f : a -> a -> a) (ne : a)
    (is : [n]i32) (as : [n]a) : *[m]
    a =
    ...
```

Obviously, do not use the built-in `reduce_by_index` in your definition. Instead, use `radix_sort_by_key`² 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
```

²https://futhark-lang.org/pkgs/github.com/diku-dk/sorts/0.3.3/doc/lib/github.com/diku-dk/sorts/radix_sort.html#4010

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

```
1 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 `reduce_by_index`?

3 Monte Carlo Simulation

In this exercise, we shall use the technique of Monte Carlo simulation for computing the value of π . We shall use a simple technique based on an external generation of random numbers.

Exercise 3.1: Monte Carlo π

In this exercise we shall make use of the “dart-throwing” technique. Observe that if one randomly throws a dart on a square of size 2×2 then the chance of hitting within the enclosed circle of radius 1, provided one hits the square, is $\frac{\pi}{4}$. It is quite easy to determine, using Pythagoras’s theorem, whether a throw (x, y) is successful, which it is if its distance to the center of the circle is less than or equal to 1, that is, if $(x - 1)^2 + (y - 1)^2 \leq 1$. Write a function `estimate_pi` that takes two arrays of `f32` values as arguments, corresponding to x and y coordinates for the throws, and, based on the above observations, gives an estimate on the value of π .

Exercise 3.2: External randomness

Using the `futhark dataset` tool, generate three datasets of different sizes, each containing two arrays of type `[]f32` of the same size, containing `f32` values between 0.0 and 2.0. Use 100, 10000, and 1000000 as the sizes for the data sets. Both for executables generated with `futhark c` and `futhark opencl` (or `futhark cuda`), plot the time for computing π as a function of the different data set sizes. What do you see?

Hint: Use the following command to generate input for the 100-size case:

```
futhark dataset --f32-bounds='0:2' -g [100]f32 -g [100]f32 > pi100.in
```

Exercise 3.3: Monte Carlo integration

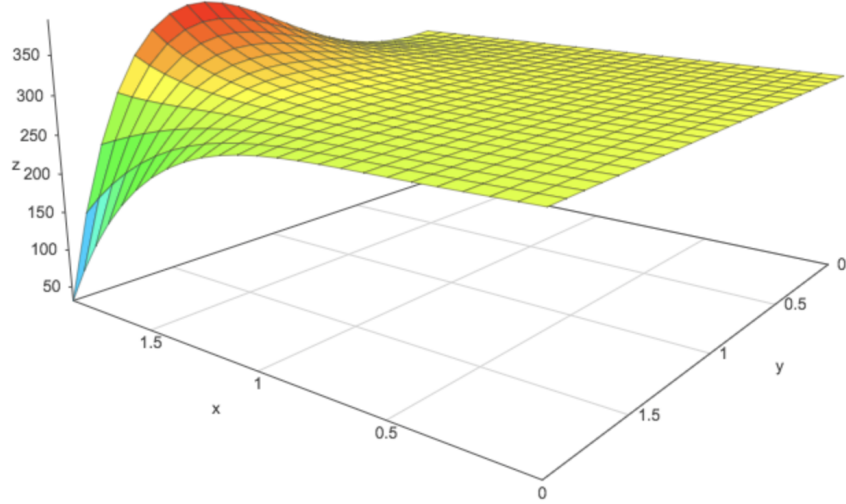
Use the same technique as above but for integrating (i.e., finding the volume under) the following mathematical function (a function of two variables) in the interval $x \in [0; 2]$ and $y \in [0; 2]$:

$$f(x, y) = 2x^6y^2 - x^6y + 3x^3y^3 - x^2y^3 + x^3y - 3xy^2 + xy - 5y + 2x^5y^4 - 2x^5y^5 + 250$$

A Futhark function resembling the mathematical function is given as follows:

```
let f (x:f32) (y:f32): f32 =  
  2*x*x*x*x*x*x*y*y - x*x*x*x*x*x*y  
  + 3*x*x*x*x*y*y*y - x*x*y*y*y +  
  x*x*x*y - 3*x*y*y + x*y -  
  5*y + 2*x*x*x*x*x*y*y*y*y -  
  2*x*x*x*x*x*y*y*y*y*y + 250
```

Here is a graph showing the surface defined by the function in the interval:



For executables generated with both `futhark c` and `futhark opencl` (or `futhark cuda`), plot the time for computing the integral as a function of the different data set sizes for the data sets generated in the previous exercise. What do you see now?

Hint: If n is the number of sample pairs S , the integral $\int_0^2 \int_0^2 f(x, y) dx dy$ can be approximated by $\frac{4}{n} \sum_{(x,y) \in S} f(x, y)$.

4 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 .

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 Hayes³. However, 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.

The code handout also comes with a visualisation, `ising-gui.fut`, written with the help of the Lys library⁴, that permits a visualisation of the computation. If the necessary SDL libraries are installed (see the Lys documentation), simply do `make run`. Running the visualisation is not required to solve the assignment; it's just a lot more satisfying.

³<http://bit-player.org/wp-content/extras/bph-publications/AmSci-2000-09-Hayes-Ising.pdf>

⁴<https://github.com/diku-dk/lys>

Exercise 4.1: Generate initial state

The code handout defines the type of spins as follows:

```
1 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:

```
1 entry random_grid (seed: i32) (h: i32) (w: i32)
2   : ([h][w]rng_engine.rng, [h][w]spin) =
3     ...
```

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

Hint: Generate one-dimensional arrays of size $n \times m$, then use `unflatten n m` at the end.

Exercise 4.2: Computing Δ_e

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

```
1 entry deltas [h][w] (spins: [h][w]spin): [h][w]i8 =
2   ...
```

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 4.3: The step function

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

```
1 entry step [h][w] (abs_temp: f32) (samplerate: f32)
2           (rngs: [h][w]rng_engine.rng)
3           (spins: [h][w]spin)
4 : ([h][w]rng_engine.rng, [h][w]spin) =
5   ...
```

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

Exercise 4.4: 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:

```
1 let main (abs_temp: f32) (samplerate: f32)
2       (w: i32) (h: i32) (n: i32): [h][w]spin =
3   (loop (rngs, spins) = random_grid 1337 h w
4     for _i < n do
5       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 the `--backend=c` and `--backend=opencl` options to switch code generators. Be aware that the Futhark compiler is not very good at generating sequential CPU code, and that stencils in particular are likely to have poor cache behaviour, so don't consider the `--backend=c` results indicative of the full power of your CPU.

Exercise 4.5: 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.