Contents

1 Introduction

To quote the official website:

> Futhark is a small programming language designed to be compiled to efficient parallel code. It is a statically typed, data-parallel, and purely functional array language in the ML family, and comes with a heavily optimizing ahead-of-time compiler that presently generates either GPU code via CUDA and OpenCL, or multi-threaded CPU code.

I sought to write idiomatic Futhark code without digging too much into compiler internals. This allowed me to see the performance one could expect from Futhark without knowing exactly how code is executed.

Throughout the project, my model of Futhark semantics was that of MapReduce. Hence why I avoided algorithms with explicit memory operations and searched for methods to calculate the Fourier Transform using only parallel array operations.

2 Fourier Transform

2.1 Discrete Fourier Transform

For any array of complex numbers $\mathbf{a}$ of length N, its Discrete Fourier Transform (DFT) is defined component-wise as:

$$\hat{a}_k = \sum_{i=0}^{N-1} a_i \zeta^{ik}$$

where

$$\hat{\mathbf{a}} = (\hat{a}_0, \ldots, \hat{a}_{N-1})$$

and ζ is a principal root of unity in the ring of complex numbers.

2.2 Parallel Fast Fourier Transform

The following algorithm is described in the MapReduce-SSA paper by Tsz-Wo Sze, where a MapReduce-friendly, parallel and relatively simple FFT algorithm is needed to perform large integer multiplication. The exception being that the paper applies FFT in the ring of integers modulo $2^n + 1$ while I work with complex numbers. I will avoid going into the tedious proof.

If we write $N = PQ$ for some positive P and Q (in the code I assume N is a perfect square) we can compute the Fourier Transform of $\mathbf{a}$ as:

1. P DFTs of Q point arrays, in parallel
2. then, Q DFTs of P point arrays, in parallel

In fact, write for all $0 \leq p < P$:

$$\mathbf{a}^{(p)} = (a_p, \ldots, a_{(Q-2)P+p}, a_{(Q-1)P+p})$$

in the code, this is referred to as `aslices`. These constitute the P DFTs of Q points needed; they are computed in parallel because there are no inter-dependencies.

Next, for all $0 \leq q < Q$, we define:

$$\mathbf{z}^{[q]} = (z_{qP}, \ldots, z_{qP+(P-2)}, z_{qP+(P-1)})$$

where

$$z_{qP+p} = \zeta^{pq} \widehat{a^{(p)}}_q$$

This corresponds to `zslices`. Likewise, these are the remaining Q DFTs.

Finally, we get for all p and q:

$$\hat{a}_{pQ+q} = \widehat{z^{[q]}}_p$$

2.3 Implementation

At first, I naively wrote two separate functions to compute `aslices` and `zslices` respectively:

```
def aslice [n] (f: factorize) (a: [n]complex.complex) (p: i64) =
  let (p_max, q_max) = f n
  in map (\q -> a[q * p_max + p]) (0..<q_max)

def zslice [n] (f: factorize) (a: [n]complex.complex) (q: i64) =
  let (p_max, _) = f n
  let root' p q = root n complex.** (complex.mk_re (f64.i64 (p * q)))
  in map (\p -> root' p q complex.* dft (aslice f a p) q) (0..<p_max)
```

The problem with this was that dependencies were not computed in the correct order, but rather re-computer several times in `zslice`. This is an example of how the Futhark compiler isn't a magically parallelizing tool; care should be taken to ensure that the parallel parts of one's algorithm map to the few parallel operations: `map`, `reduce`, `scatter`, ...

Aside from this, Futhark *feels* like any other ML-style language in that many familiar idioms transfer naturally (e.g higher-order functions).

3 Results

Having had some difficulties running OpenCL/CUDA on my machine (complete lack of support), and on Grid'5000 (random exceptions from the Futhark runtime) I present benchmarks only from the Multicore and Sequential backends.

The `bench-fft` benchmark uses an existing FFT implementation by the developers of Futhark, it uses the Stockham algorithm and is generally more optimized than mine.

3.1 Multicore backend

```
lib/github.com/fuzzypixelz/fft/bench-dft.fut:
#0 ("4i32"):        249s (95% CI: [     248.0,      250.9])
#1 ("5i32"):       4187s (95% CI: [    4124.1,     4269.4])
#2 ("6i32"):      59851s (95% CI: [   58871.0,    63095.1])

lib/github.com/fuzzypixelz/fft/bench-fft.fut:
#0 ("4i32"):         72s (95% CI: [      71.8,       71.9])
```

```
#1 ("5i32"):        356s (95% CI: [     355.6,      356.2])
#2 ("6i32"):       2808s (95% CI: [    2795.5,     2824.5])

lib/github.com/fuzzypixelz/fft/bench-parallel-dft.fut:
#0 ("4i32"):        121s (95% CI: [     120.7,      122.5])
#1 ("5i32"):        576s (95% CI: [     572.4,      578.9])
#2 ("6i32"):       3960s (95% CI: [    3936.6,     3985.5])
```

3.2 Sequential C backend

```
lib/github.com/fuzzypixelz/fft/bench-dft.fut:
#0 ("4i32"):        569s (95% CI: [     568.4,      570.2])
#1 ("5i32"):      15077s (95% CI: [   15040.9,    15125.7])
#2 ("6i32"):     242988s (95% CI: [  242339.8,   244602.9])

lib/github.com/fuzzypixelz/fft/bench-fft.fut:
#0 ("4i32"):         52s (95% CI: [      52.4,       52.6])
#1 ("5i32"):        274s (95% CI: [     273.8,      274.4])
#2 ("6i32"):       2836s (95% CI: [    2829.2,     2845.0])

lib/github.com/fuzzypixelz/fft/bench-parallel-dft.fut:
#0 ("4i32"):        105s (95% CI: [     105.1,      105.3])
#1 ("5i32"):        795s (95% CI: [     793.0,      797.7])
#2 ("6i32"):       7151s (95% CI: [    7136.9,     7164.9])
```

4 Conclusion

The above results show that Futhark is quite useful when the problem can be decomposed neatly into parallelizable array operations. Thanks to its familiar syntax, one could get pretty far without really understanding its semantics. Still, the Futhark compiler is not a parallelizing compiler and one should be explicit about which operations would be done in parallel.