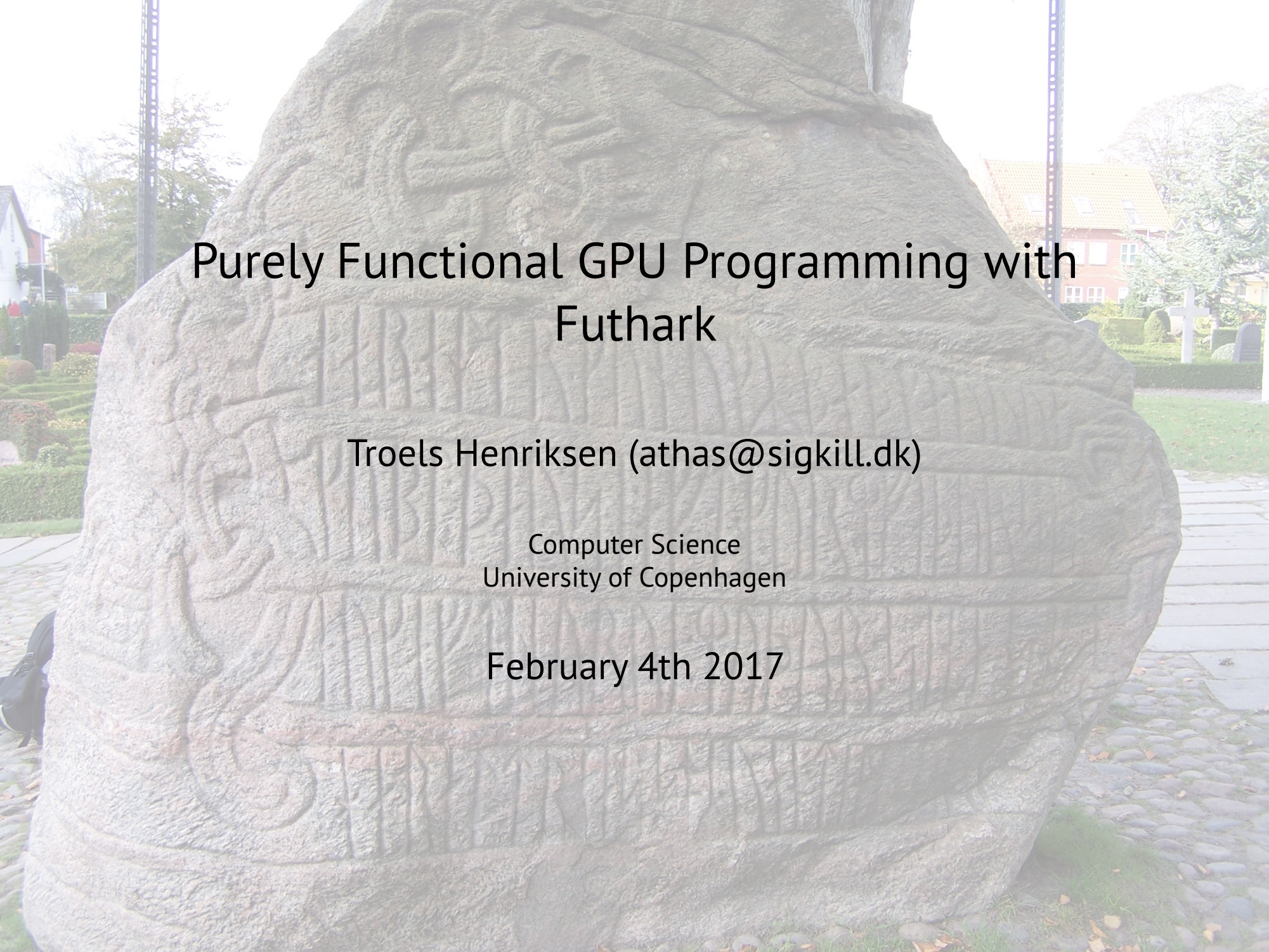


Purely Functional GPU Programming with Futhark

Troels Henriksen (athas@sigkill.dk)

Computer Science
University of Copenhagen

February 4th 2017

Agenda

The Problem

Modern hardware can handle (and requires) tens to hundreds of thousands of parallel threads. The human mind cannot handle this.

Agenda

The Problem

Modern hardware can handle (and requires) tens to hundreds of thousands of parallel threads. The human mind cannot handle this.

The Solution

Functional array programming is a restricted programming paradigm that performs well in practice and is easy to reason about (for some problems).

Agenda

The Problem

Modern hardware can handle (and requires) tens to hundreds of thousands of parallel threads. The human mind cannot handle this.

The Solution

Functional array programming is a restricted programming paradigm that performs well in practice and is easy to reason about (for some problems).

Agenda:

1. Array programming with Futhark
2. Inter-operability (with Python)
3. GPU performance compared to hand-written code

Two Kinds of Parallelism

Task parallelism is the simultaneous execution of different functions across the same or different datasets:

```
spawn_thread(f, x)
spawn_thread(g, y)
```

Data parallelism is the simultaneous execution of the same function across the elements of a dataset:

$$\mathtt{map}\ f\ [v_0, v_1, \ldots, v_{n-1}] \ = \ [f\ v_0,\ f\ v_1,\ \ldots,\ f\ v_{n-1}]$$

Array programming is in the latter category.

Array Programming

- Programs are expressed as bulk operations on arrays.
  In Python with Numpy:

```
>>> import numpy as np
>>> a = np.arange(10)
>>> b = a * 2
>>> sum(a*b)
570
```

- Popular because it resembles mathematics.

Array Programming

- Programs are expressed as bulk operations on arrays.
  In Python with Numpy:

```
>>> import numpy as np
>>> a = np.arange(10)
>>> b = a * 2
>>> sum(a*b)
570
```

- Popular because it resembles mathematics.
- *Old*—first seen in APL from 1964:

```
a ← ⍳10
b ← a×2
+/a×b
```

- Less popular.

Futhark at a Glance

Small eagerly evaluated pure functional language with data-parallel looping constructs. Syntax is a combination of C, SML, and Haskell.

- **Data-parallel loops**

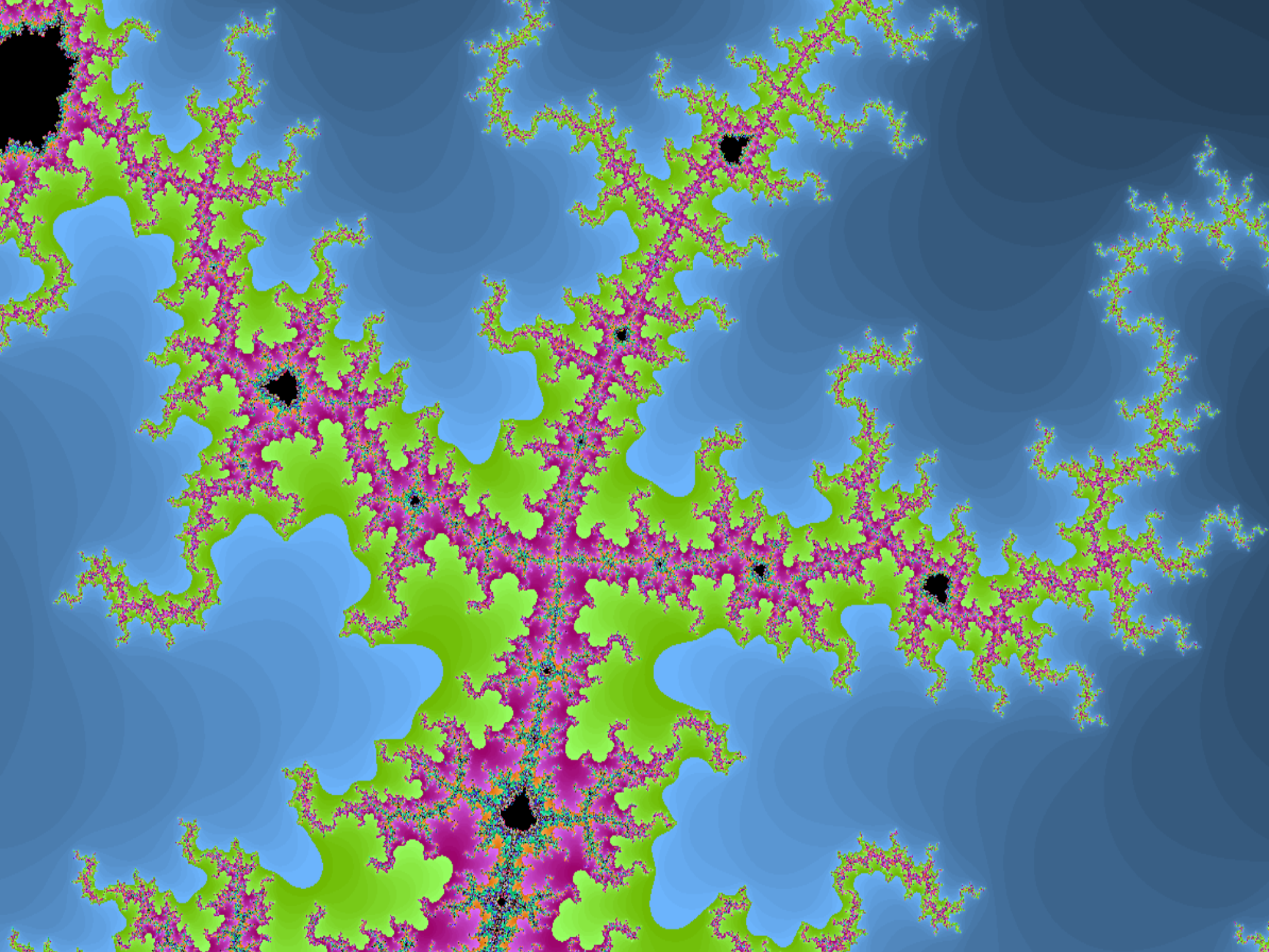
```
fun add_two (a:  [n]i32):     [n]i32 = map (+2) a
fun     sum (a:  [n]i32):        i32 = reduce (+) 0 a
fun sumrows (as: [n][m]i32): [n]i32 = map sum as
```

- **Sequential loops**

```
fun main (n: i32): i32 =
  loop (x = 1) = for i < n do
    x * (i + 1)
  in x
```

- **Array Construction**

```
iota 5           =         [0,1,2,3,4]
replicate 3 1337 = [1337, 1337, 1337]
```



Computing the Mandelbrot Set

The root of those pretty visuals is calling this function (here in Python) with a bunch of complex numbers:

```
fun divergence(c, d) =
  i = 0
  z = c
  while i < d and dot(z) < 4.0:
    z = c + z * z
    i = i + 1
  return i
```

Mandelbrot in Numpy[1]

```
def mandelbrot_numpy(c, d):
    output = np.zeros(c.shape)
    z = np.zeros(c.shape, np.complex64)
    for it in range(d):
        notdone =
          np.less(z.real*z.real + z.imag*z.imag, 4.0)
        output[notdone] = it
        z[notdone] = z[notdone]**2 + c[notdone]
    return output
```

Problems

- Control flow obscured.
- Always runs for *maxiter* iterations.
- *Lots* of memory traffic - three arrays written for every iteration of loop.

---

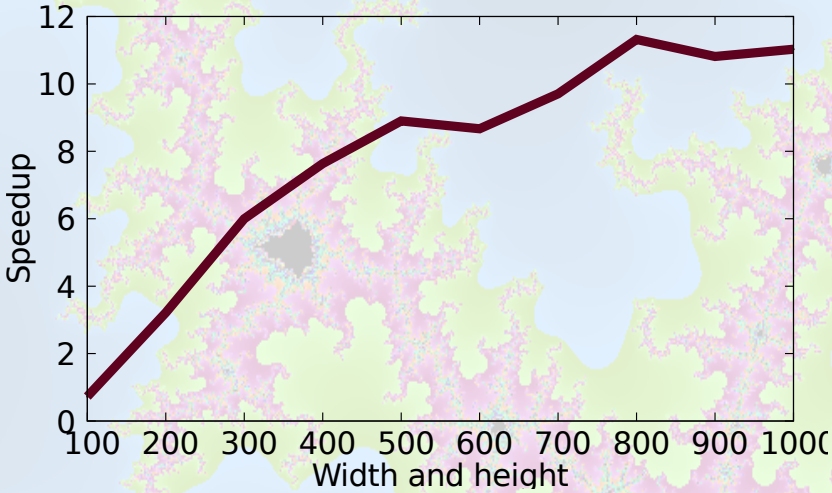
[1] https://www.ibm.com/developerworks/community/blogs/jfp/entry/How_To_Compute_Mandelbrodt_Set_Quickly

Mandelbrot in Futhark

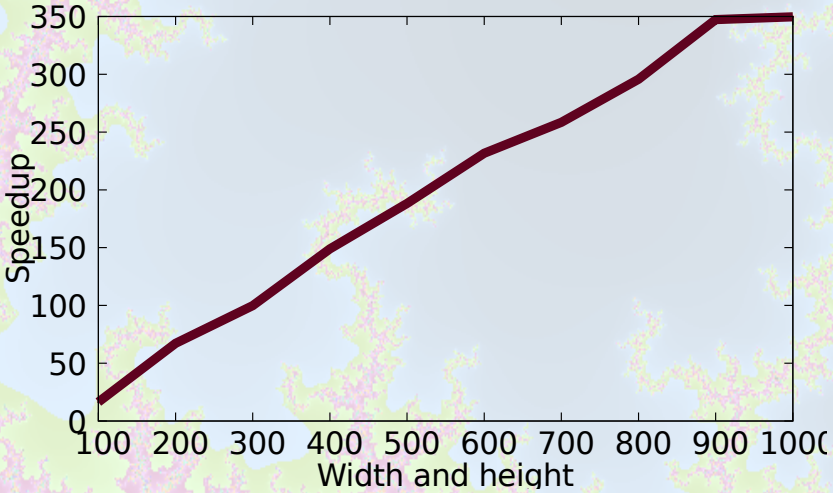
```
fun divergence (c: complex) (d: i32): i32 =
  loop ((z, i) = (c, 0)) = while i < d &&
                                dot(z) < 4.0 do
    (addComplex(c, multComplex(z, z)),
     i + 1)
  in i

fun mandelbrot(css: [n][m]complex) (d: i32): [n][m]i32 =
  map (\cs ->
         map (\c -> divergence c d)
             cs)
      css
```

- Only one array written, at the end.
- `while` loop terminates when the element diverges.

Mandelbrot speedup on GPU compared to sequential implementation in C



Numpy-style



Futhark-style

Moral: The vectorised style can sacrifice a lot of potential performance.

Running a Futhark Program

Define contrived entry point

```
fun main (n: i32) (m: i32) (d: i32): i32 =
  let css = make_complex_numbers n m
  let escapes = mandelbrot css d
  in reduce (+) 0 (reshape (n*m) escapes)
```

- Creates some arbitrary complex numbers, computes their divergence, and sums the results.
- Futhark is a pure language and cannot read input or write results itself.
- **When launching a Futhark program, we must indicate an entry point and input data.**

Compile to sequential code

```
$ futhark-c mandelbrot.fut -o mandelbrot-c
$ echo 10000 10000 100 | \
  ./mandelbrot-c -t /dev/stdout
611240
999901i32
```

Compile to sequential code

```
$ futhark-c mandelbrot.fut -o mandelbrot-c
$ echo 10000 10000 100 | \
  ./mandelbrot-c -t /dev/stdout
611240
999901i32
```

Compile to parallel (GPU) code

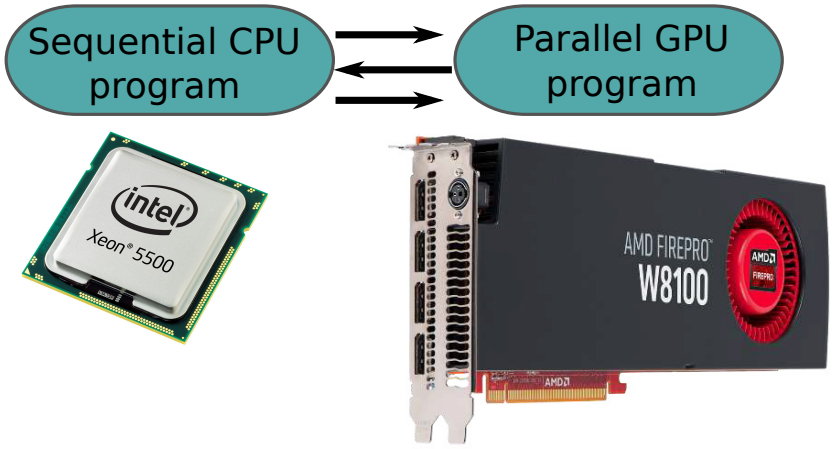
```
$ futhark-opencl mandelbrot.fut -o mandelbrot-opencl
$ echo 10000 10000 100 | \
  ./mandelbrot-opencl -t /dev/stdout
7550
999901i32
```

Advantage

80× speedup of parallel over sequential execution.

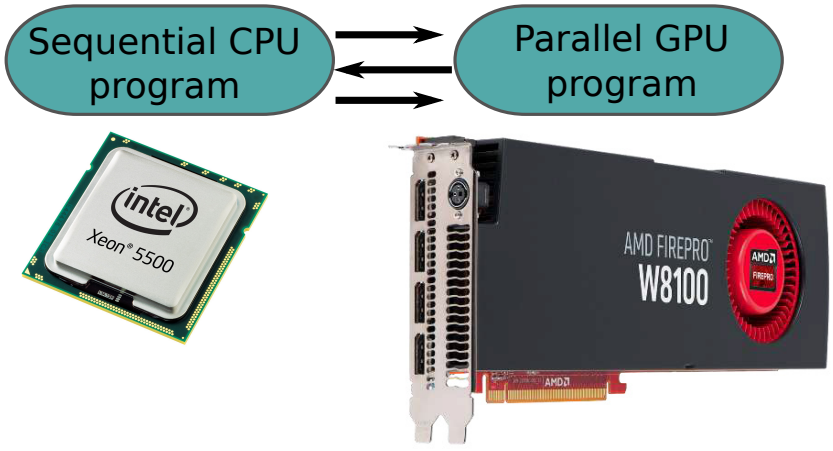
How OpenCL Works

- The CPU uploads code and data to the GPU, queues execution, and copies back results.
- **Observation:** the CPU code is all management and bookkeeping and does not need to be particularly fast.



How OpenCL Works

- The CPU uploads code and data to the GPU, queues execution, and copies back results.
- **Observation:** the CPU code is all management and bookkeeping and does not need to be particularly fast.



How Futhark Becomes Useful

We can generate the CPU code in whichever language the rest of the user's application is written in. This presents a convenient and conventional API, hiding the fact that GPU calls are happening underneath.

Compiling Futhark to Python+PyOpenCL

```
$ futhark-pyopencl --library mandelbrot.fut
```

This creates a Python module `mandelbrot.py` which we can use as follows:

```
$ python
>>> import mandelbrot
>>> m = mandelbrot.mandelbrot()
>>> m.main(100, 100, 255)
25246
>>> m.main(1000, 1000, 300)
299701
```

Good for all your mandelbrot summing needs.

Compiling Futhark to Python+PyOpenCL

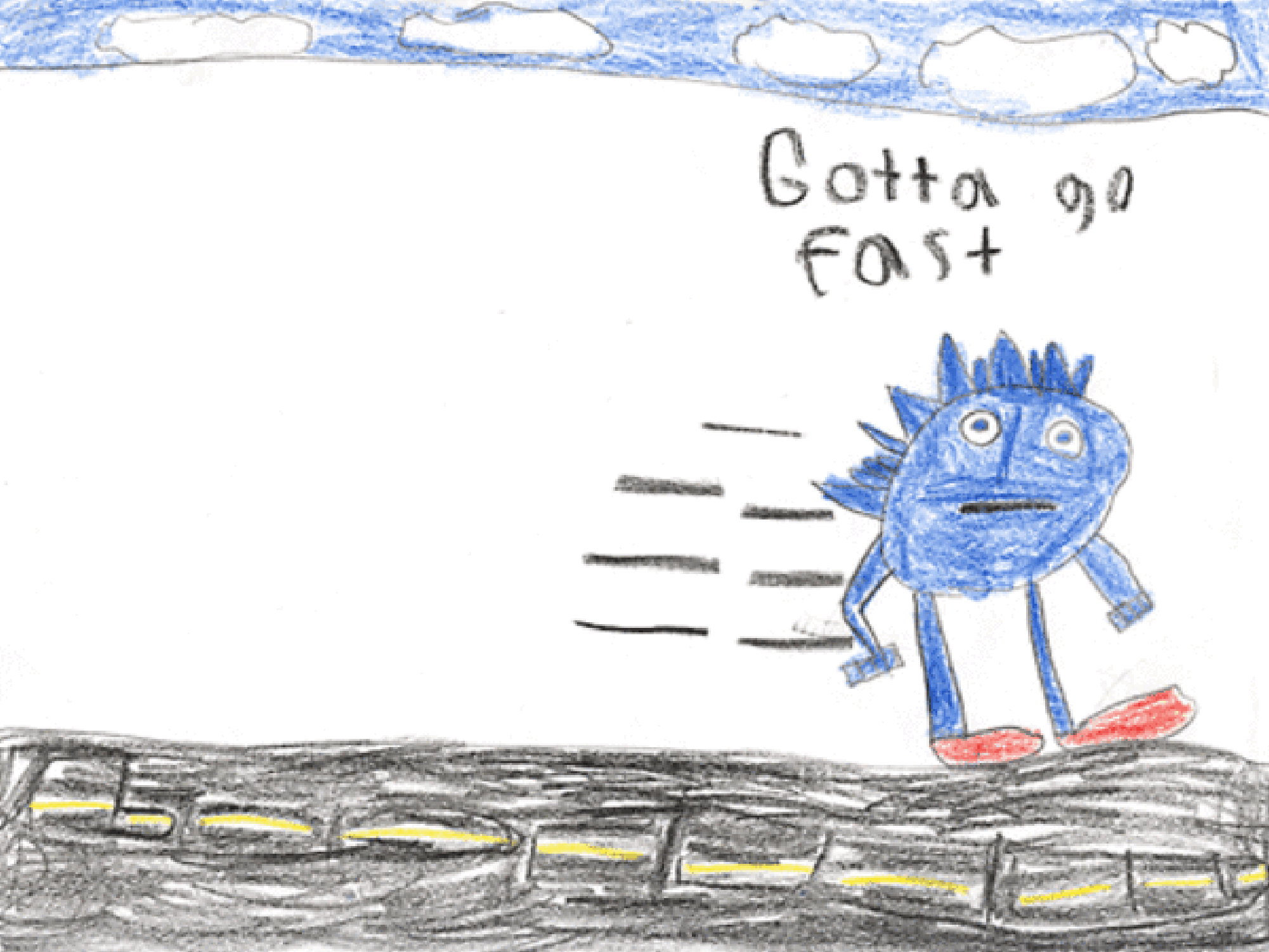
```
$ futhark-pyopencl --library mandelbrot.fut
```

This creates a Python module `mandelbrot.py` which we can use as follows:

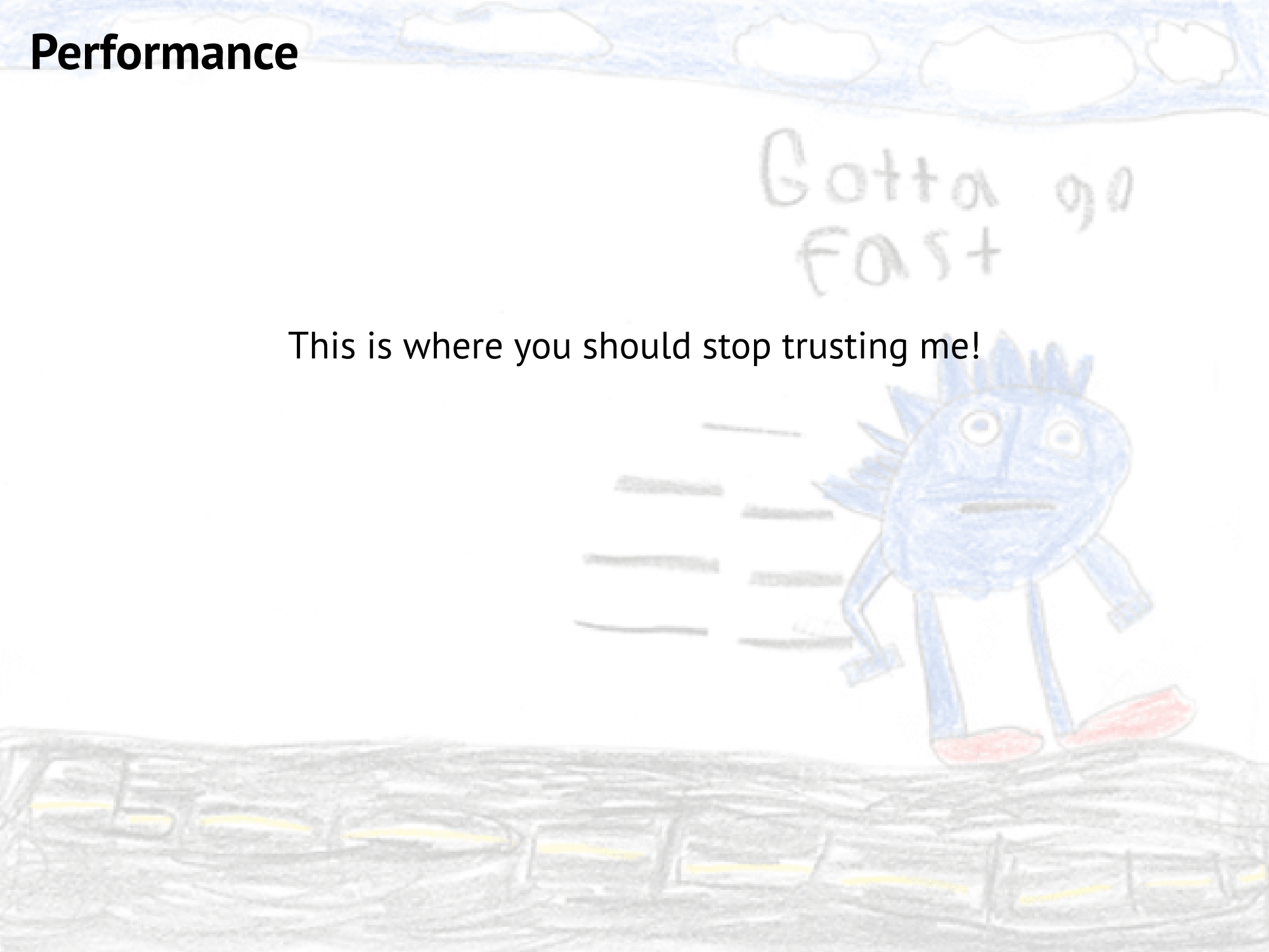
```
$ python
>>> import mandelbrot
>>> m = mandelbrot.mandelbrot()
>>> m.main(100, 100, 255)
25246
>>> m.main(1000, 1000, 300)
299701
```

Good for all your mandelbrot summing needs.
Or, we could have our Futhark program return an array containing pixel colour values, and use Pygame to blit it to the screen...

Gotta go fast

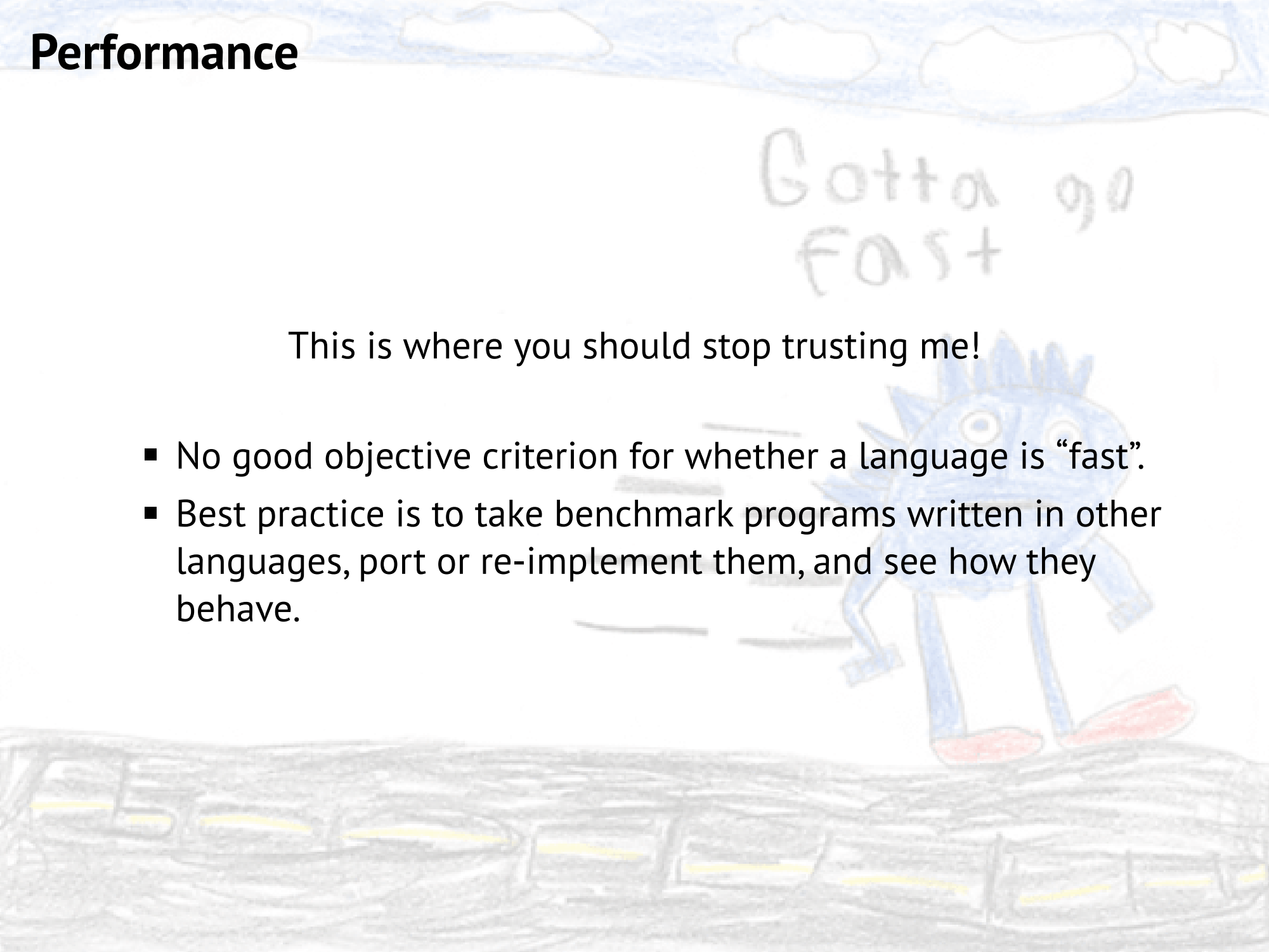


Performance



This is where you should stop trusting me!

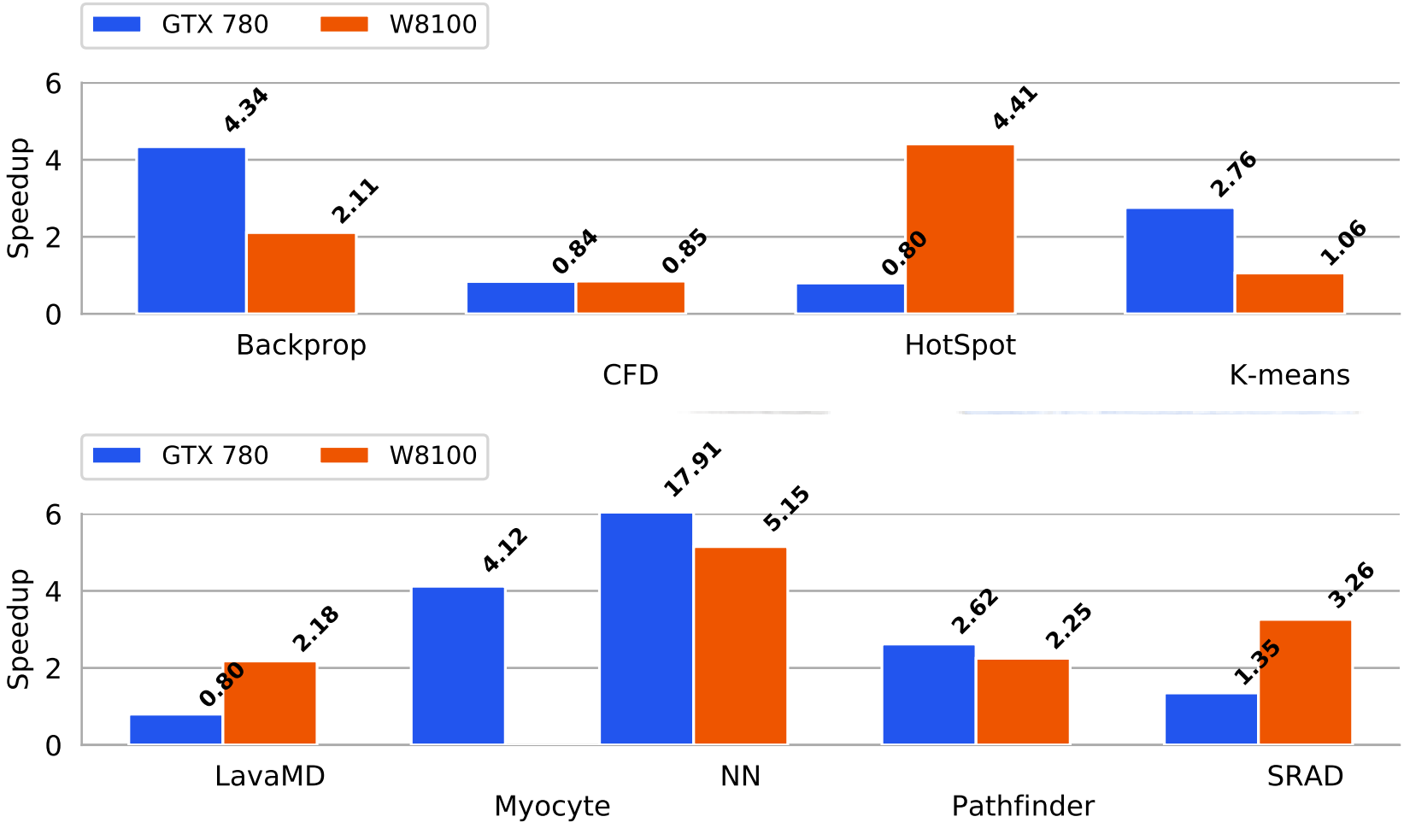
Performance



This is where you should stop trusting me!

- No good objective criterion for whether a language is “fast”.
- Best practice is to take benchmark programs written in other languages, port or re-implement them, and see how they behave.

Speedup Over Hand-Written Rodinia OpenCL Code



Summary

- Futhark is a small high-level functional data-parallel language with a GPU-targeting optimising compiler.
- Can be integrated with existing languages and applications.
- Performance is okay.

Questions?

Website `https://futhark-lang.org`

Code `https://github.com/HIPERFIT/futhark`

Benchmarks `https://github.com/HIPERFIT/futhark-benchmarks`