Contributing to Futhark for your Bachelors Project

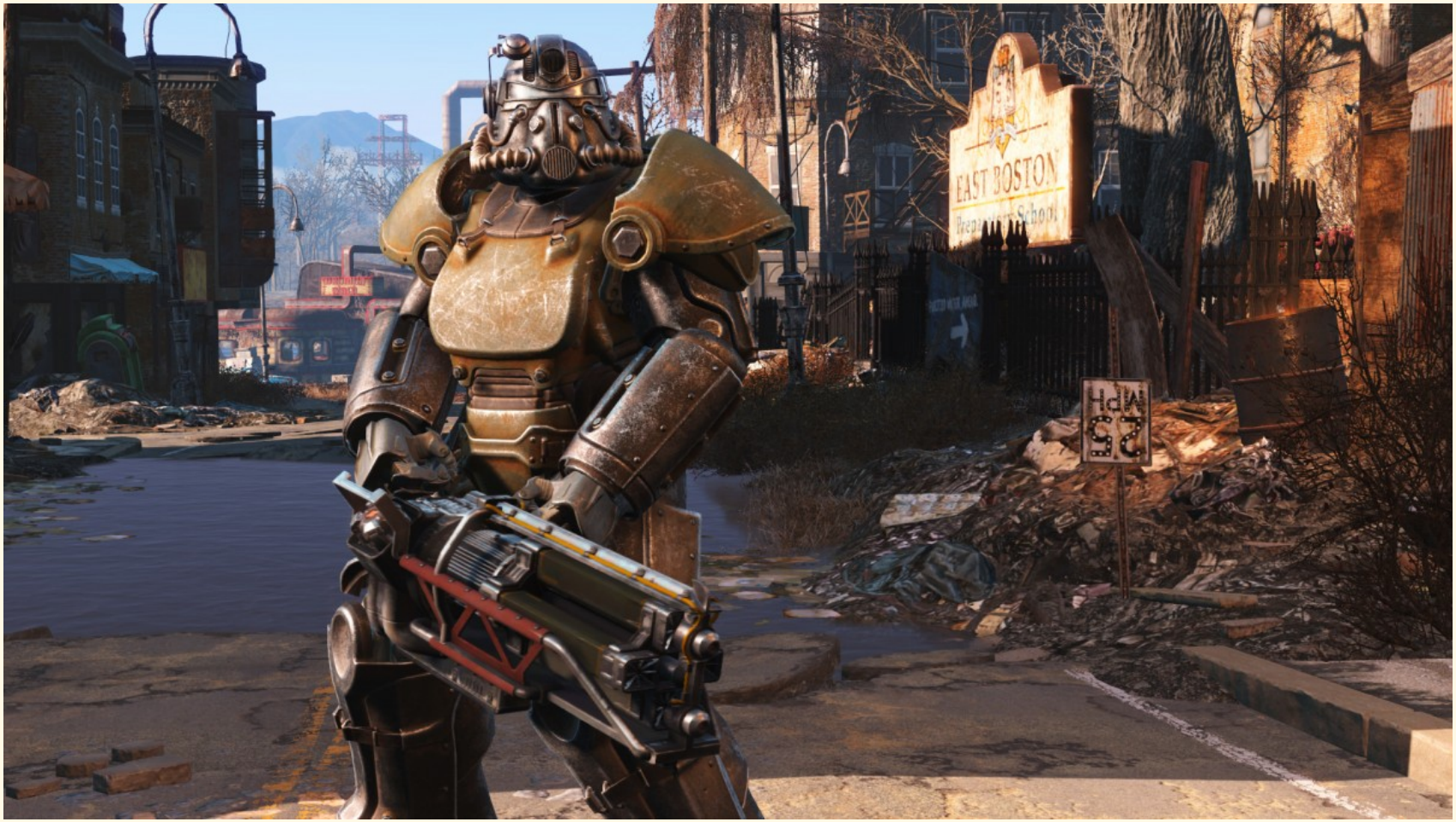
Troels Henriksen (athas@sigkill.dk)

Computer Science
University of Copenhagen

5th of February 2018

I recently bought an expensive new graphics card

Not to look at this:



But at this:

```
clGetPlatformIDs(1, &platform, &platforms);
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,
               &device, &devices);
cl_context_properties properties[]= {0};
cl_context context =
  clCreateContext(properties, 1, &device,
                  NULL, NULL, &error);
cl_command_queue cq =
  clCreateCommandQueue(context, device,
                       0, &error);

cl_program prog =
  clCreateProgramWithSource(context, 1,
                            &transpose_cl,
                            NULL, &error);
clBuildProgram(prog, 0, NULL, "", NULL, NULL);
```

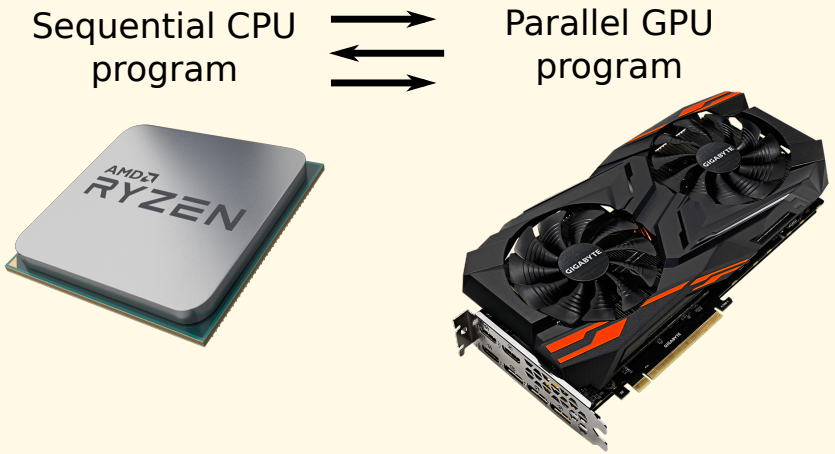
Context

- Graphics cards (*GPUs*) provide massive amounts of computational power, but they are hard to program. It gets even harder (impossible) if you want both reusable and fast code.
- Still, GPGPU (using GPUs for non-graphics workloads) has been increasing in popularity for the past ten years, as the potential performance improvements are significant.
- GPUs are extremely parallel processors capable of keeping tens of thousands of threads in flight. But these threads have to access memory in certain patterns, and do roughly the same, or they will run very slowly.
- This calls for a language that makes it easier to deal with.

GPU programming

GPUs function as extremely fast data-parallel *co-processors*. They are not independent, but are controlled by an ordinary CPU process that sends code and data.

- For strange historical reasons, GPU programs are often called *kernels*. The CPU code is called *host code*.
- At runtime, a CPU program will send a GPU program (often specified in a dialect of C) to the device driver for the GPU, and get back some GPU-specific machine code.
- This GPU-code is then sent to the GPU along with data in order to perform computation.

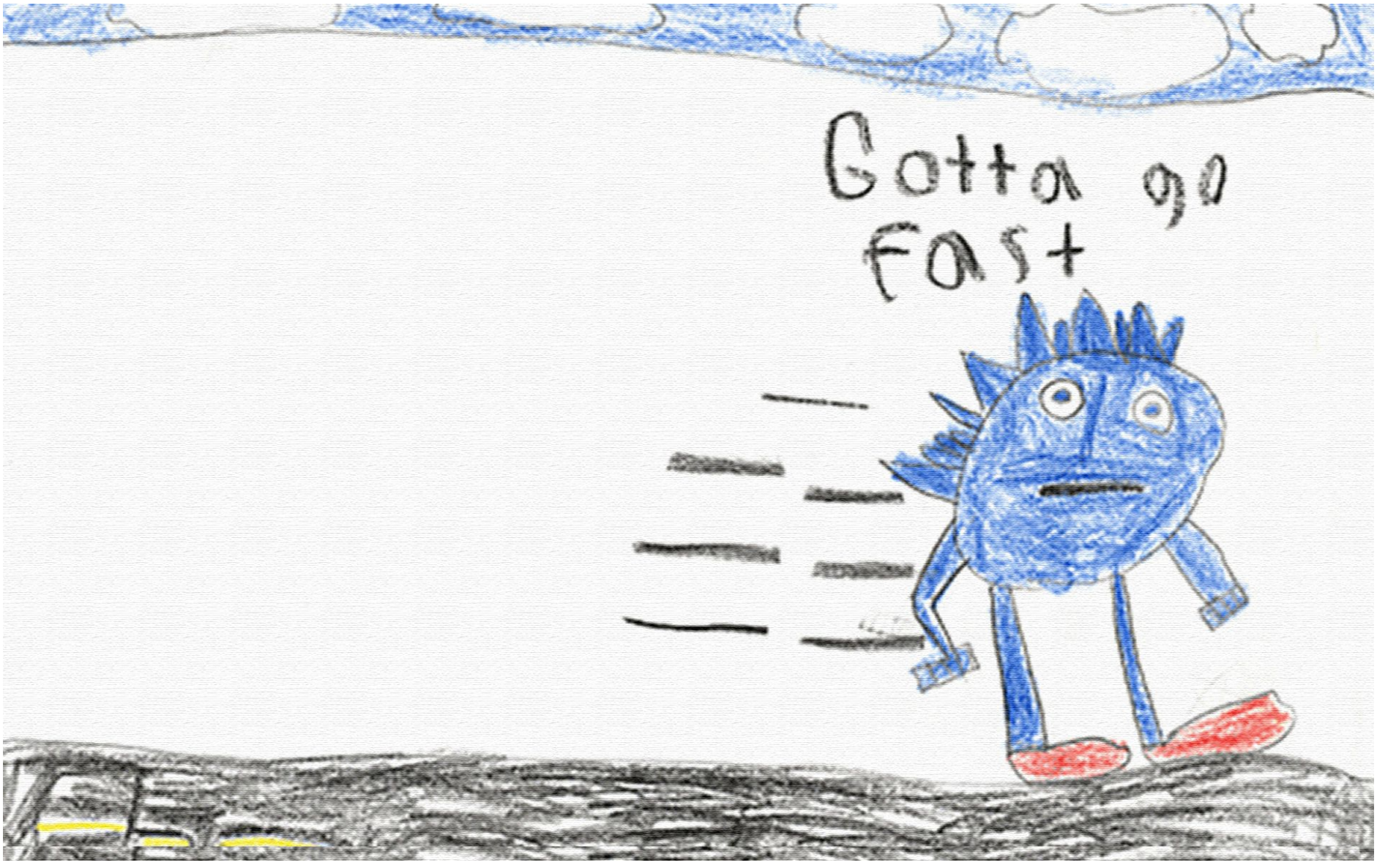


Futhark overview

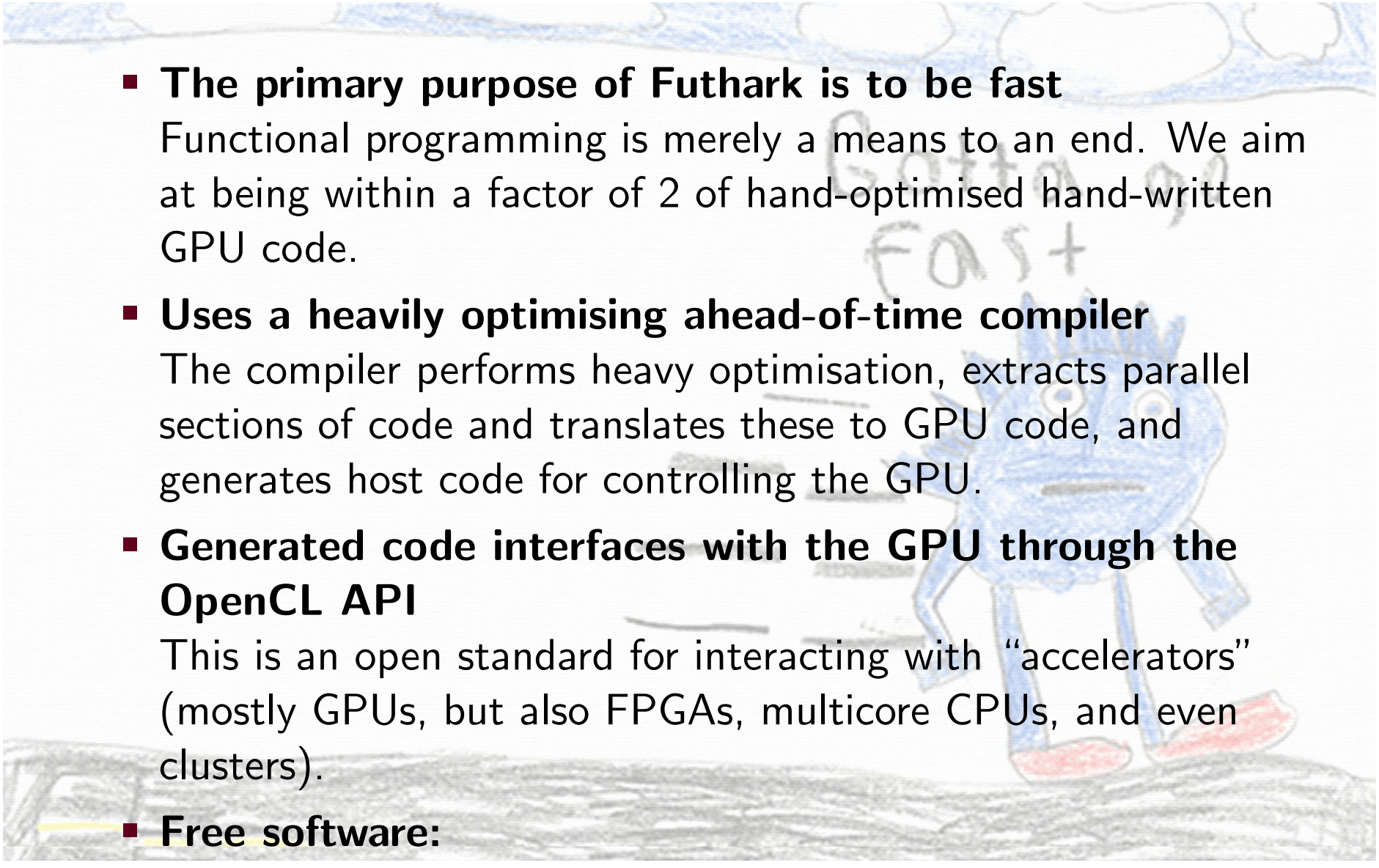
We have created a programming language, *Futhark*, intended for writing parallel programs that can be compiled to run on GPUs and other parallel machines.
Futhark is a data-parallel purely functional array language. Looks a bit like a simplified combination of SML and Haskell.

```
let vector_add [n] (a: [n]i32) (b: [n]b): [n]i32 =
  map (+) a b

let product [n] (a: [n]i32): i32 =
  reduce (*) 1 a

let dot_product [n] (a: [n]i32) (b: [n]i32): i32
  product (vector_add a b)
```

So what do we want to do with this language?


Gotta go
fast

Futhark Details



- **The primary purpose of Futhark is to be fast**
  Functional programming is merely a means to an end. We aim at being within a factor of 2 of hand-optimised hand-written GPU code.
- **Uses a heavily optimising ahead-of-time compiler**
  The compiler performs heavy optimisation, extracts parallel sections of code and translates these to GPU code, and generates host code for controlling the GPU.
- **Generated code interfaces with the GPU through the OpenCL API**
  This is an open standard for interacting with "accelerators" (mostly GPUs, but also FPGAs, multicore CPUs, and even clusters).
- **Free software:**
  `https://github.com/diku-dk/futhark.git`

Roughly Three Kinds of Projects

(Light) Compiler Hacking:

The Futhark compiler is written in Haskell. It is well written, but not small (45k SLOC). Only fairly limited work possible for a bachelors project.

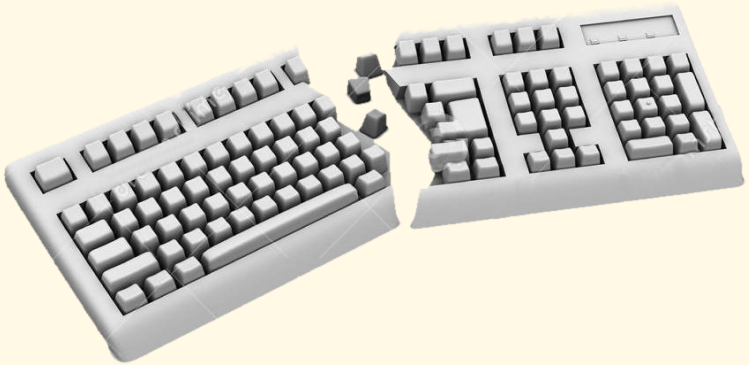
Parallel Programming in Futhark:

We port benchmarks, libraries, and example applications to determine areas in which Futhark should be improved.

Work on Infrastructure and Tooling:

Futhark is run as a sound software engineering project, which means tooling to run tests, analyse benchmark results, etc. These are very "open" projects, in that you are free to choose language and methodology yourself.

The Unit of Difficulty



- We rate the difficulty of project proposals from one to five broken keyboards.
- Not simply a measure of workload, but based on whether we believe we understand every issue related to implementation of the project.
- Higher difficulty implies more “unexpected” problems that we cannot immediately give you an answer to.

Project: Java Backend

Idea The host code could be generated in any language with OpenCL bindings. Since most of the runtime is ideally spent in the GPU code, performance of the host language should not matter much. We already have code generators for Python and C, and soon C# - we would like to add Java to this list.

Challenges Mapping Futhark IR constructs and types to Java; generating a nice API. However, you will have to write code in Haskell.

Difficulty

Prior generations of bachelors students have already found the landmines for you - but it's still fun and rewarding work.

Project: Java Backend (details)

In principle, we could have a backend code generator for every language out there. A fully functioning code generator is not particularly large (less than 2000 SLOC for the Python backend). However, they are still a maintenance burden, so we would like to keep the number limited. JVM and .NET are, however, interesting, because by supporting these, Futhark would immediately be accessible to all other languages running on these platforms (Java, C#, Kotlin, F#, Scala, etc).
One thing that eased the development of the Python backend (which was itself a bachelors project in 2016) was the presence of a mature library for calling OpenCL. These also exist for .NET (either NOpenCL or OpenCL.Net), and the JVM (JOCL).
This project will require you to write a nontrivial amount of code in Haskell, although the existing backends provide a pretty good template to follow. For example, see the following files:

- `https://github.com/diku-dk/futhark/blob/master/src/Futhark/CodeGen/Backends/GenericPython/AST.hs`
- `https://github.com/diku-dk/futhark/blob/master/src/Futhark/CodeGen/Backends/GenericPython.hs`
- `https://github.com/diku-dk/futhark/blob/master/src/Futhark/CodeGen/Backends/PyOpenCL.hs`

The Python backend is pretty full-featured. We don't expect all the bells and whistles from a bachelors project.

Project: Benchmark Tooling

Idea Every time a change is made to the Futhark compiler, our benchmark suite is automatically run on various machines, and the results stuffed away. We now have over 5000 JSON files with data, but very little tooling that allows us to analyse how compiler performance has varied over time. We would like to have such tools.

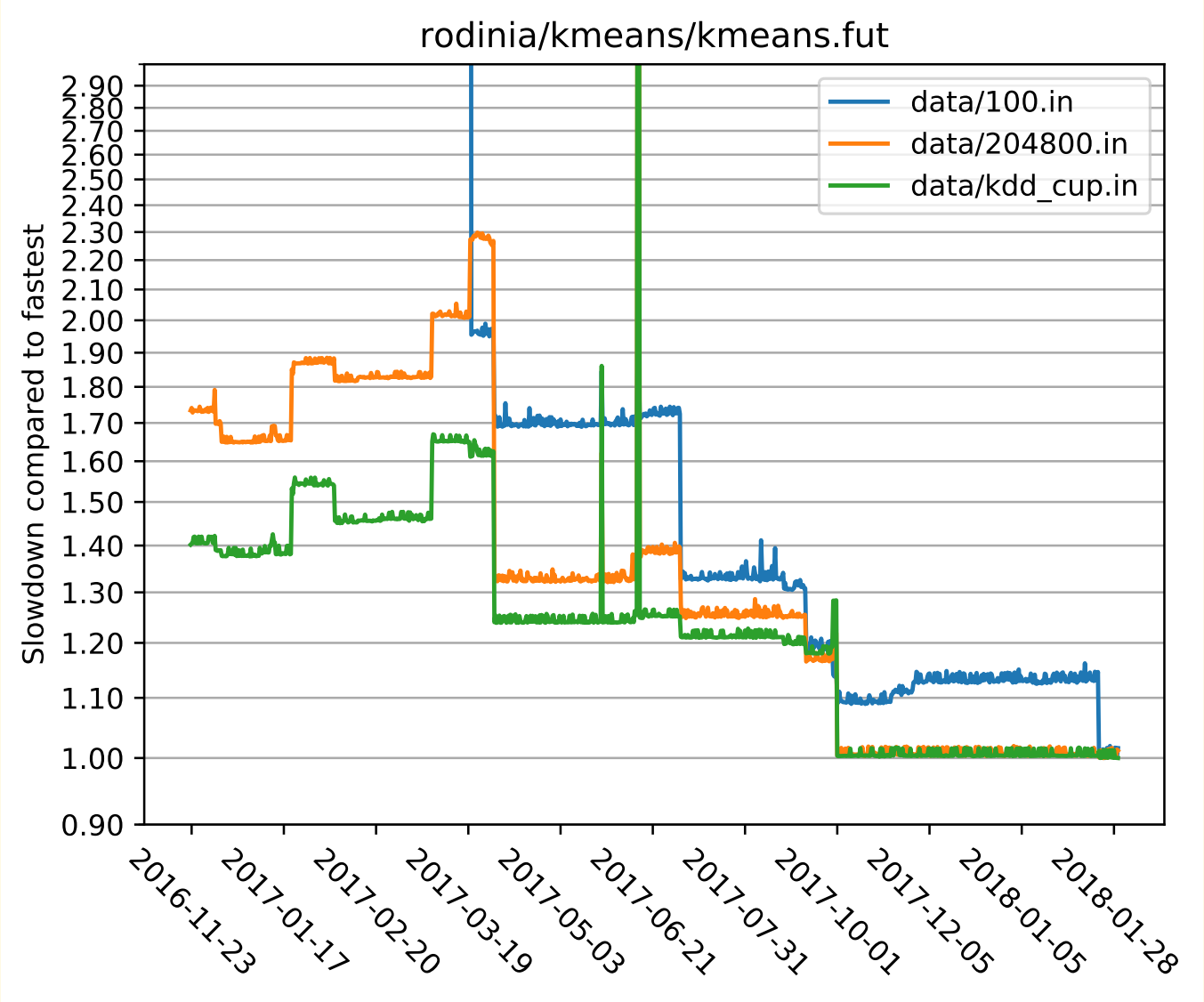
Challenges Involves a bit of independent problem analysis to figure out what we need, as well as some UI design and visualisation work. You can probably look at established compiler projects for inspiration.

Difficulty

Feasible even for a single student - and you can pick whichever language, infrastructure and tools you want.

Project: Benchmarking Tooling (details)

We have a little bit of tooling, which for a specific machine and benchmark can give us a graph of how the runtime has varied over time, proportional to the fastest the benchmark has ever been.



Project: Benchmark Tooling (details)

The graph on the previous slide is generated by a script hacked up in Python, and has multiple problems:

Not automatic: Whenever benchmark results are produced, we would like such graphs (or similar) to be produced for all machine/benchmarks configurations and made publicly available within a web page somewhere on `https://futhark-lang.org`

Lacks information: E.g. exactly which change caused the large performance degradation in the beginning of summer 2017, and which one fixed it? All benchmark results are keyed to Git commits, so the answer exists, it is just hard to get to. An interactive Javascript-based graph with zooming and extra information (such as absolute runtime) on hover could help here.

No ability to query: I can't ask "when was this benchmark fastest, and what was the first change that caused a (significant) slowdown?" I would also like to be able to to ask "given two commits IDs, give me a summary of the performance changes across *all* benchmarks".

We have some suggestions on what a solution should look like, but are pretty open to anything that can help us make better use of our present and future data. If you want to play with web-based visualisations and graphs, then we're fine with that. If you think doing query languages would be fun, that would be great too. Anything is better than what we have right now.

Project: CUDA Backend

Idea Currently, the Futhark compiler generates code using the OpenCL library, but some interesting platforms support only NVIDIAs CUDA. It would be useful to add a CUDA backend to Futhark.

Challenges There are three different CUDA layers that could be targed by the code generator: language, runtime, and driver level. I have no idea which one is best. Will possibly require some minor modification of kernel code generator (the CUDA and OpenCL kernel languages are similar but not identical).

Difficulty

Project: CUDA Backend (details)

This project involves a significant amount of independent work on studying the CUDA documentation and determining how to use its low-level facilities correctly.
In order to ease maintenance, the new CUDA backend should re-use most of the concepts used by the existing OpenCL backend. Hence, you will probably have to write some glue code to make CUDA look a bit more like OpenCL. I do not expect this to be conceptually difficult.
You will need to write a fairly nontrivial amount of Haskell for this one. However, you will be working at the very tail end of the compiler, so you will not have to spend a lot of time studying an existing large code base. In a perfect world, you might be able to get away with simply writing some runtime system code (in C/C++) and a CUDA-fied version of this module—

`https://github.com/diku-dk/futhark/blob/master/src/Futhark/CodeGen/Backends/COpenCL.hs`

—but we do not live in a perfect world.
One very interesting question is whether code generated by the CUDA backend will actually run faster than the equivalent code generated by the OpenCL backend, and if so, why.

Project: A Statistics Library

Idea We need to improve the capabilities of Futhark's standard library. Not just so we can avoid writing the same primitives over and over again, but also to test that the library and compiler is flexible enough. A library containing various statistical methods and primitives would be useful.

Challenges Programming with an experimental compiler always carries a little risk. Also, data-parallel functional programming is likely not a paradigm that any of you have prior experience with. Of course, that's also what makes it *fun*.

Difficulty

Project: A Statistics Library (details)

The best avenue of attack is probably to port an existing statistics library. Doubly so if you are not already experienced with statistics, and have an idea of what such a library should contain. By basing the work on an existing library, you get test cases almost for free, which saves you from having to understand exactly *what* the statistical methods compute. Instead, you just have to make sure you get roughly the same results.

This is generally the approach you have to take when porting benchmarks and libraries to a new language, and there is no way you can be an expert in all of computational fluid dynamics, ray-tracing, heat dynamics, heart-wall simulations, molecular dynamics, image processing, and fractals (to name just a few of the topics covered by the Futhark benchmark suite).

We are not picky as to which existing library should form the basis for the work, but this Javascript library looks pretty simple:

`https://github.com/compute-io/compute.io`

The work would involve translating statistical primitives, of course, but also redesigning the library to take advantage of Futhark's unique features (such as the module system) and restrictions.

If you would rather port a useful non-statistics library (maybe some deep learning!), that would also be welcome. In particular, machine learning or linear algebra primitives would be useful.

Project: Implement Language Server Protocol (LSP) for Futhark

Idea *"The Language Server protocol is used between a tool (the client) and a language smartness provider (the server) to integrate features like auto complete, go to definition, find all references and alike into the tool"*. Implement an LSP server for Futhark to get these features in various code editors that support LSP.

Challenges The LSP is a real industrial spec, and probably too large to implement completely in a single bachelor's project. You will have to identify a realistic yet practically useful subset.

Difficulty

Project: Implement Language Server Protocol (LSP) for Futhark (details)

Futhark already has an Emacs major mode that supports syntax highlighting, (some) automatic indentation, and such. We could work on adding support for autocompletion and *go to definition* functionality in an ad-hoc way, but not only is it a lot of work, it would also not be accessible to those unfortunate people who do not use Emacs. By implementing an LSP server instead, this functionality would also be available to people using such editors as VSCode, Sublime Text, Vim, etc.

The Futhark compiler frontend can be used to analyse a Futhark program to obtain identifiers and the locations of various definitions. Realistically, in order for this information to be accessible, the LSP server will have to be written in Haskell. Alternatively, I could extend the Futhark compiler to dump all this information in some standard format (JSON) which you could read in an LSP server implemented in another programming language.

While I have not ever implemented an LSP server myself, it is my impression that it is not overly complicated. However, this is definitely a coding-heavy project.

The LSP spec `https://microsoft.github.io/language-server-protocol/specification`

Project: SequenceL to Futhark compiler

Idea Texas Multicore produces a data-parallel functional language called SequenceL. It uses quite a different way to express parallelism, based on implicit vectorisation. It would be interesting to write a compiler that translates SequenceL to Futhark.

Challenges I don't know SequenceL in detail, and there is no SequenceL implementation with freely available source code. (You can write the first one!)

Difficulty

Project: SequenceL to Futhark compiler (details)

This is a matrix multiplication written in SequenceL:

```
matmul(A(2), B(2)) [i,j] := sum( A[i,all] * B[all,j] );
```

And this is in Futhark:

```
let matmul [n][m] (A: [n][m]i32, B: [m][p]i32): [n][p]i32 =
  map (\A_row ->
         map (\B_col -> reduce (+) 0 (map (*) A_row B_col))
             (transpose B))
      A
```

There are two main differences that enable the SequenceL implementation to be more succint:

- SequenceL has syntax that directly indicates that the function produces a two-dimensional array, with the function definition giving the result for a given `[i,j]` index.
- SequenceL overloads functions and operators (here, `*`) to operate on arrays and not just scalars. They use a transformation called *normalise-transpose* to do this in a systematic manner.

In Futhark's view of the world, both of these correspond to inserting appropriate uses `maps` at appropriate locations, so we believe there is no deep different in the expressivity of the two languages. Since the Futhark compiler generates code that is, as far as we can determine, significantly better than that which is produced by the SequenceL compiler, there is value in being able to translate SequenceL to Futhark.
This project involves creating a SequenceL compiler *from scratch*, using whichever technology (language, tooling, whatever) you prefer. You will have to implement all the usual bits of a compiler (lexer, parser, type checker), except that the code generator will produce high-level Futhark instead of low-level assembly, as you may have done before. If you want to try your hand at writing your own compiler, then this is the project.

SequenceL on Rosetta Code: `https://rosettacode.org/wiki/Category:SequenceL`

SequenceL documentation: `https://texasmulticore.com/documentation/3.0/index.html`

Other Projects

If you have an idea of your own, come by and we can have a chat about it.
Inspiration:

- A "Try Futhark" web application like `https://tryhaskell.org/`.
- Port a benchmark from a suite like Rodinia or Parboil to Futhark.
- An automatic code formatter like `gofmt`.
- Fixing/improving/rewriting the automatic indentation of the `futhark-mode` for Emacs.

If you have an ≤ 15 ECTS idea for something that would contribute to the project, we can probably supervise it.
I can also supervise other projects related to programming languages, parallel programming, or high-performance computing, but the more distant the project is from my core research, the less specific help I can provide.

Compiler hacking is hard work, but rewarding

"Futhark: the harder the battle the sweeter the victory!"
-Rasmus Wriedt Larsen (bachelor and masters thesis)

"Hvis du laver et Futhark-projekt, kommer du aldrig til at sove."
-Niels G. W. Serup (bachelor and masters thesis)

"Et ambitiøst projekt med en hårdt arbejdende kerne af folk! Hvad mere kan man ønske sig?" *-Hjalte Abelskov (bachelor thesis)*

"Efter at havde arbejdet med Futhark føles alt andet i mit liv meningsløst." *-Daniel Gavin (bachelor thesis)*

"Futhark: Så lille et sprog, der kan så meget for så mange!"
-Kasper Abildtrup Hansen (bachelor thesis)

Contact

If you think any of this is interesting, come talk to Troels...

- ...in the office shared with Cosmin: 01-0-017 at HCØ.
- ...via email: `athas@sigkill.dk`
- ...or on IRC: #diku on Freenode (`http://ucph.dk/#tabs1-chat`)

Or talk to Martin Elsman or Cosmin Oancea.

Also check out the website at `https://futhark-lang.org`