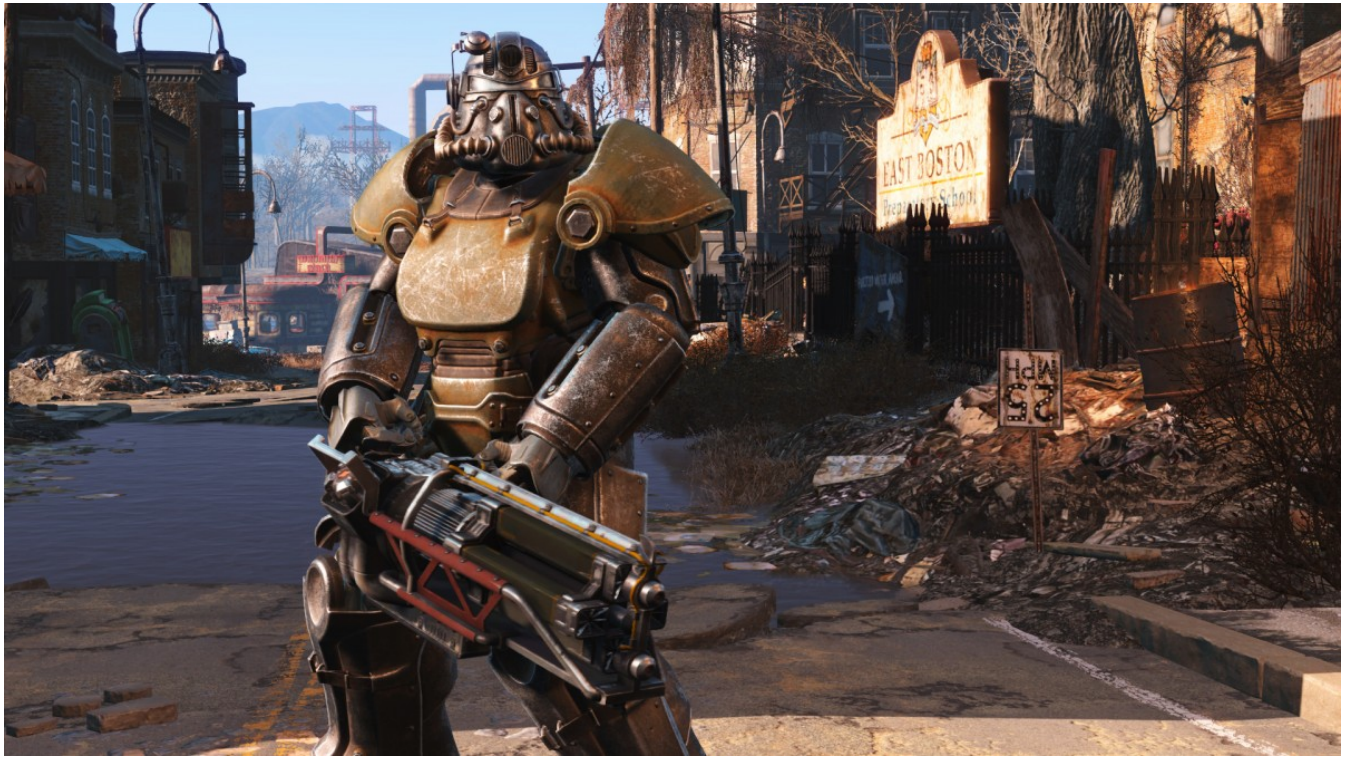
Contributing to Futhark for your Bachelors Project

Martin Elsman (mael@di.ku.dk)
Troels Henriksen (athas@sigkill.dk)

Computer Science
University of Copenhagen

4th of September 2017

A Computing Scientist is a person who gets his hands on a nice 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);
```

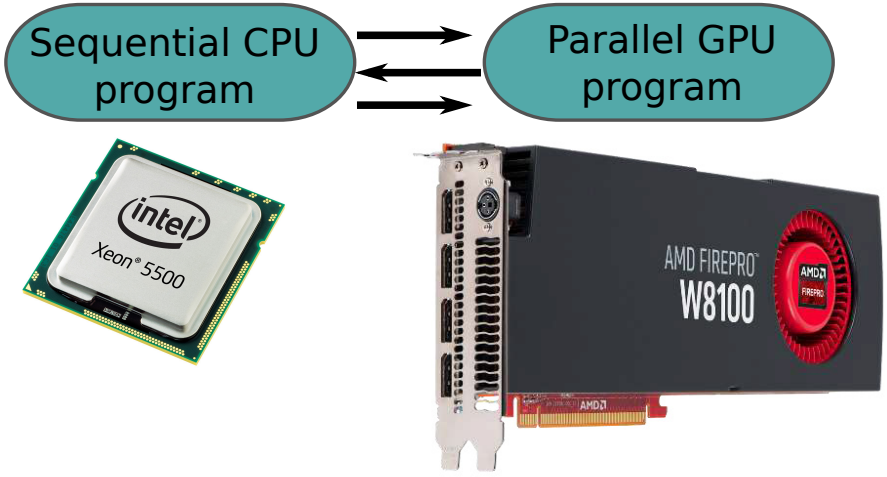
Context

- 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.

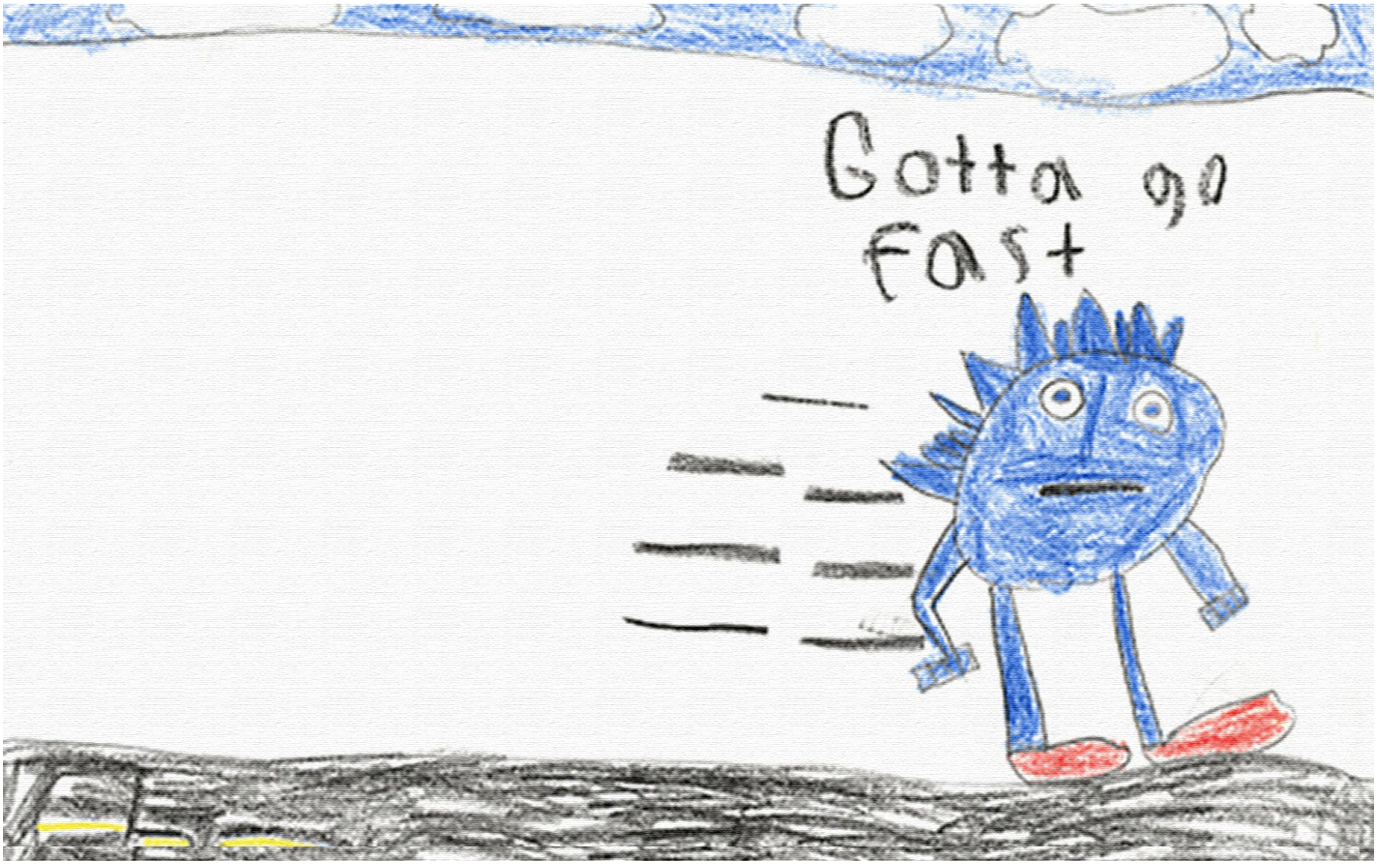


Futhark overview

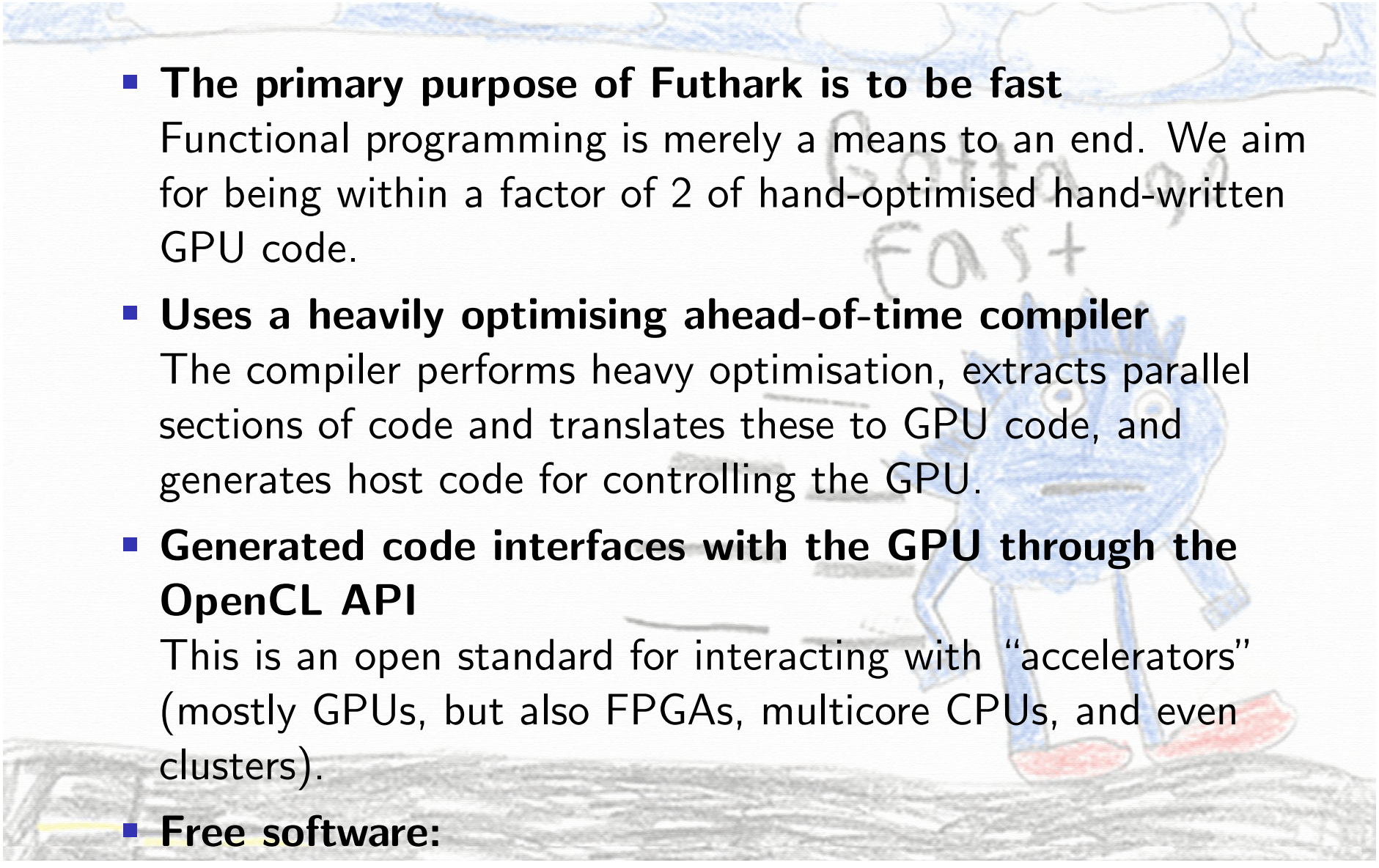
We have created a programming language, *Futhark*, intended for writing parallel programs that can be compiled to run on GPUs. Futhark is a data-parallel purely functional array language. Looks a bit like a simplified combination of SML and Haskell.

```
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 for 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 large (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: Porting Benchmarks from 3Shape

Idea 3Shape is a Danish company that produces 3D scanners and CAD/CAM software for the dental and audio industries. To try out more productive ways of writing parallel code, they have suggested two small programs, presently written in C#, that students can port to Futhark.

Challenges It can require some creativity to figure out where parallelism exists in code written with sequential execution in mind.

Difficulty

3shape

Project: Porting Benchmarks from 3Shape (details)

The two suggested projects, as described by Peter Dahl from 3Shape, are:

Removal of projected light patterns in TRIOS color images.

Our TRIOS scanner is able to generate 1 mega pixel color images, which is used for our HD Photo feature and intraoral camera. In these pictures it is possible to see the projected dot pattern, which degrade the image quality. Our own approach is to filter the image by removing certain frequencies in the frequency domain. This is done by transforming the image with an FFT, removing the desired frequencies, and then applying the inverse FFT. Our current implementation is not real-time and not well parallelized, so it would be interesting to explore using Futhark for this.

Estimating start guess transformation of RGB-D images.

In the TRIOS Software RGB-D images are received real-time from the scanner, and it is the responsibility of the scanning software to combine these into one coherent model. For registration we use the well-established ICP algorithm, where the first step is to find an start guess of how to align the incoming image to the current model. In TRIOS we find the start guess by doing an FFT based image correlation, where the image from the latest registered image is multiplied with incoming image in the frequency domain. Our current implementation is only partly parallelized, so it would be interesting to use Futhark to make a better performing implementation.

3shape

Project: Improving the C Backend

Idea The main compiler backend can generate very fast standalone programs via C, but we would like for the generated code to be callable by other programs through a C API. We already tested this approach in Python, where it works very well.

Challenges Designing an API that can represent both transparent and opaque Futhark values at the C level, and coming up with a useful scheme for error handling.

Difficulty

Designing C APIs requires good taste and judgment. Good error handling may be tricky, since C does not natively support exceptions.

Project: Improving the C Backend (details)

At the end of the Futhark compilation pipeline, the program is converted into an imperative intermediate form with explicit invocations of OpenCL kernels. This imperative form is then translated to a concrete programming language—currently either Python or C. The Python code generator supports a mode where it generates a reusable Python module that can be imported by ordinary Python code, and we would like the same for C.
Concretely, we want the capability to take a program `foo.fut`, which defines various Futhark functions as entry points, and compile it into a pair of files `foo.h` and `foo.c`. A C program can then link with `foo.c` and use the interface in `foo.h` to call the entry points.
Mapping types is a problem. For example, one entry point may be a function that takes a two-dimensional array:

```
entry mat_scale [n][m] (xss: [n][m]i32) (a: i32): [n][m]i32 = ...
```

Unlike Python, C does not define a multidimensional array type, so how should this look in the generated API? One way would be to define a C type

```
struct array_2d_i32 { ... };
```

and a function used to create such arrays from flat C data

```
struct array_2d_i32 mk_array_2d_i32(int, int, int32_t*);
```

and then declare the entry point as

```
struct array_2d_i32 mat_scale(futhark_context_t*, struct array_2d_i32, int);
```

Where the `futhark_context_t*` argument functions as an opaque handle for GPU state and such. C types and packing/unpacking functions would have to be defined for every Futhark type used in entry points.

While some hacking on the compiler will be needed, the main challenge is the API design.

Project: Java or .NET Backends

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. We already have code generators for Python and C - we would also like Java or C#/F#.

Challenges Mapping Futhark IR constructs and types to the target language; 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: JVM or .NET Backends (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.

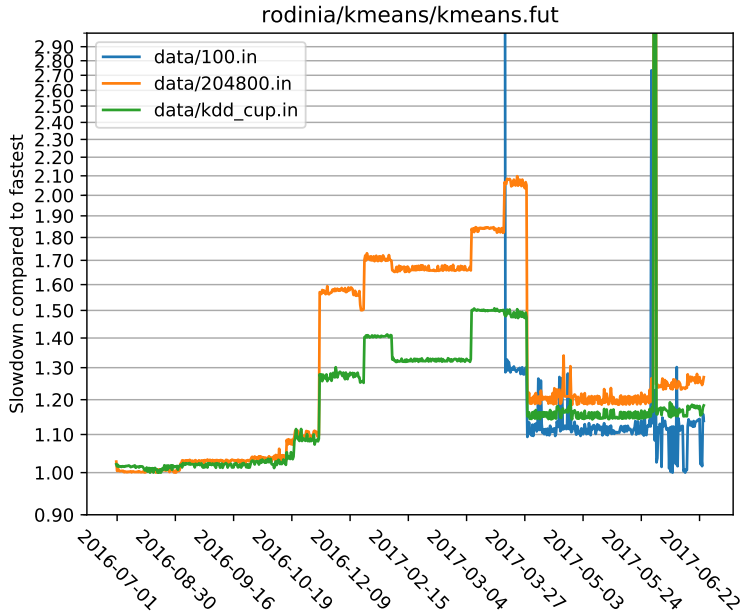
Challenges Involves a bit of independent problem analysis to figure out what we need, as well as some UI design and visualisation work. You can probably look at established compiler projects for inspiration.

Difficulty

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

Project: Benchmark Tooling (defails)

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.



This graph 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, 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?"

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 possible some minor modification of kernel code generator (the CUDA and OpenCL kernel languages are similar but not identical).

Difficulty

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, that would also be welcome. In particular, machine learning or linear algebra primitives would be useful.

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/`.
- An automatic code formatter like `gofmt`.

If you have an ≤ 15 ECTS idea for something that would contribute to the project, we can probably supervise it.

Compiler hacking is hard work, but rewarding

"Futhark: the harder the battle the sweeter the victory!"
-Rasmus Wriedt Larsen

"Hvis du laver et Futhark-projekt, kommer du aldrig til at sove"
-Niels G. W. Serup

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

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

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.