UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's thesis

Mikkel Storgaard Knudsen

FShark

Futhark programming in FSharp

Advisor: Cosmin eller Troels

Handed in: July 24, 2018

Abstract

Here is a nice abstract
prut prut

Contents

1 Introduction 3

2 Background 10

3 The FShark subset 12

4 FSharkPrelude 18

4.1 Arrays in F# versus in Futhark 21

4.2 Converting jagged arrays to Futhark's flat arrays, and back again . . . 23

4.2.1 Analysis of FlattenArray 23

4.2.2 Analysis of UnflattenArray 24

5 The FShark Compiler and Wrapper 26

5.1 Design choices in writing the FShark Compiler 31

6 The Futhark C# backend 34

6.0.1 Entry functions 39

7 Benchmarks and evaluation 43

8 Current limitations 46

9 Related work 47

10 Conclusion and future work 48

11 FSharks interoperability between F#and Futhark (C#) 49

11.0.1 Pros and cons of the current design 50

11.1 The future of FShark interoperability 51

Chapter 1

Introduction

Developers worldwide are, and have always been, on the lookout for increased computing performance. Until recently, the increased performance could easily be achieved through advances within raw computing power, as CPU's had steadily been doubling their number of on-chip transistors, in rough accordance to Moore's Law (citér her).

However, the performance increases in CPU design has now slowed down significantly, due to physical limitations to CPU design.

Instead of adding more transistors or increasing the clock frequency in newer CPUs, the CPU manufacturers have instead opted to split their single-core CPUs up into multicore CPUs, which means that any program can now run several threads on the CPU's cores, simultaneously. The CPU's cores are specialized in advanced computations 's cores are

TODO: TODO: udfyld om hvordan GPU'en har mange flere kerner, og kan bruges til relativt simple operationer

In sectors like the financial sector and within the natural sciences, there is a need for handling large amounts of data in an effective manner. With algorithmic trading gaining ground within the trading sector, the trader who can analyze incoming buy- and sell-offers the fastest, usually has the advantage at the exchanges.

Likewise, faster computing can increase productivity for chemists and biologists who are analysing large datasets, physicists can run faster simulations, and so on. All of these activities are based on executing relatively simple computations on enormous datasets. The hundreds of simultaneous threads on the GPUs are, compared to the CPU, optimal for performing these calculations as fast as possible, which is why GPUs are increasingly being used for *General Purpose Computing on Graphics Processing Units*.

Parallel programming in Futhark

GPU programming is in principle easily available for everyone. As long as the user has access to a GPU and a reasonable PC for developing software, it just takes a bit of effort and reading to get started with CUDA, OpenCL or similar programming. Realistically however, it takes much more than just a little effort to start writing one's own GPU programs.

Take for instance the function $f(x) = ax + y$. In figure 1.1 we see the function implemented as a CUDA program. In this program, we are defining the kernel `saxpy` itself, and also manually copying data back and forth between the GPU. Compared to the same program written in Futhark (figure 1.2.)

Whereas the CUDA kernel needs to check whether the current thread is outside of the bounds of the input data, the equivalent kernel in Futhark is simply a declaration of it's function. Also, the Futhark version does not bother with getting array elements by the current thread ID.

In the main function itself, the initial lists are generated by functions, and the user doesn't have to allocate space on neither the computer *host*, or the GPU *device*. The hard work is done by a Futhark SOAC, which is eventually compiled into a kernel

Of course, Futhark is compiled into either C- or Python code that does indeed contain memalloc calls, GPU kernels with bounds checking, and so on, but that part is very well hidden from the Futhark programmers themselves. All in all, writing effective GPU programs becomes much more accessible when it's possible to do in a declarative manner, like Futhark, without also having to the minute details that comes with GPU computations.

```
  #include <stdio.h>

__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  cudaFree(d_x);
  cudaFree(d_y);
  free(x);
  free(y);
}
```

Figure 1.1: $ax + y$ in CUDA

```
let saxpy (a : f32) (x : f32) (y : f32) : f32 =
  a*x+y

let main =
  let N = 1<<20
  let a = 2f32
  let xs = replicate N 1f32
  let ys = replicate N 2f32
  let ys = map2 (saxpy a) xs ys
  in ys
```

Figure 1.2: $ax + y$ in Futhark

Motivation

FShark is intended to be a way of writing and utilizing Futhark, without actually having to write or interact with the Futhark language and compiler itself. Besides some tooling and an F# SOAC library, it primarily consists of the FShark compiler that compiles from F# source code to Futhark source code, and the Futhark C# generator, which compiles Futhark programs as either standalone C# programs or -libraries.

As much as most developers are happy to increase performance on big computations, it is not always an option to incorporate an extra langauge into an already existing programming language. At this moment, using Futhark in either a C#- or F# project is a contrived process that usually requires spawning a subprocess with a futhark-opencl C program from inside one of the .NET projects.

In order to use Futhark natively in .NET languages, it is therefore necessary to write a backend for Futhark in a .NET language. For FShark, I have chosen to implement this backend in C#, as the Futhark intermediate code ImpCode[1] is trivial to translate into imperative C# statements and expressions. Also, there are C# libraries available which supply OpenCL bindings, which are needed to implement the necessary OpenCL constructs from ImpCode.

It is my belief that exporting Futhark programs as .NET executables and -libraries will lower the barrier to Futhark usage in .NET projects significantly, hopefully increasing the all-round number of Futhark users, and in the long term, increasing utilization of GPU programming and making it more widely available.

However, one could do even more than just exporting Futhark to .NET, to increase accessibility:

As tens of thousands of programmers worldwide (CHECK NUMBER JEEEEZ) are already writing F# programs, and that most of F#s functional language features can be directly translated into equivalent Futhark features, it became worthwhile to investigate whether it was possible to design a way for users to both write and utilize Futhark in F# projects, without ever actually touching the Futhark language or compiler themselves. Instead, users can write their data parallel F# modules in FShark, and compile these modules into Futhark libraries automatically.

In this case, it would be possible to get Futhark speeds in F# programs, without doing much more than installing the Futhark compiler locally, and adding the required FShark libraries to the F# project.

It is my belief that being able to achieve Futhark performance in regular F# programs almost automatically, will make it significantly easier for people to adapt to Futhark programming.

(SOME MORE MORE SOME MORE)

[1] which stands for Imperative Code

The contributions of this thesis

The contributions of this thesis are as follows:

1. The FSharkPrelude:
   The FSharkPrelude is a subset of the Futhark SOACs, ported to F#. To write an FShark program, the user is directed to limit himself to the SOACs in the FSharkPrelude. This means exchanging Array.map for FSharkPrelude.Map, Array.foldBack for FSharkPrelude.foldr, and so on. However, the FSharkPrelude carries the guarantee, that all FSharkPrelude functions works equivalently to their Futhark SOAC namesakes. This prelude, together with the F# subset chosen for FShark, makes it possible to write F# programs which, when translated to Futhark code, are equivalent to their Futhark counterparts.

2. An F# subset translatable with FShark:
   As F# is not only a multi-paradigm language, but also has access to the entire standard .NET library, it was required to make FShark support only a subset of F#. This has been implemented by whitelisting only the F#-to-Futhark translatable types, constructs and expressions in the FShark compiler. Furthermore, no other module imports than FSharkPrelude are allowed. This subset is of course documented for users.

3. The FShark Compiler and Wrapper:
   The FShark Compiler and Wrapper takes a module written in FShark as input, converts the FShark module into a compiled Futhark C# module, and makes it available to the F# program, all at runtime. The pipeline is described in sec **??**

4. A C# backend for Futhark:
   To actually use Futhark in C# projects (and transitively F# projects), it was necessary to develop and add a C# backend to the Futhark compiler. This backend is equivalent in functionality to the C- and the Python backends that are already available.

Vocabulary

Unless otherwise specified, these are the terms used in the thesis:

For `FShark`

- The `FShark` *subset* is the subset of the `F#` language that is supported by the `FShark` compiler.
- The FShark Prelude is the library of `F#`-ported Futhark array functions and SOACs, and is included with `FShark`.
- `FShark` code is `F#` code which exclusively uses the `FShark` subset and FSharkPrelude.
- `FShark` modules are `F#` modules written entirely in `FShark` code.
- `FShark` projects are `F#` projects which uses `FShark` and `FShark` modules.

For Futhark

- Futhark code is code written in Futhark.
- Futhark C-, Python- or `C#` code refers to Futhark code that has been compiled into C-, Python or `C#` source code.

Roadmap

The main part of this thesis is split in four parts. blaaaah

Chapter 2

Background

In this chapter we will describe the languages that FShark depends on, including a short introduction to the OpenCL framework.

F#

F# is a relatively young .NET-based language, first released in 2003. It is a strongly-typed multiple-paradigm language, with a syntax that is primarily functional, resembling OCaml. Although F# is not as widely used as C#, Java and the like, it is currently experiencing increasing adaptation among developers[**?**]. Besides supporting multiple paradigms and a reasonable subset of functional langauge features (such as pattern matching), F#s primary strength is it's interoperability with the rest of the .NET ecosystem. Like C#, F# programs are compiled into Microsoft's Common Intermediate Langauge, and executed using Microsoft's Common Language Runtime.

Therefore, F# programs have full access to the standard .NET library, just as it can also readily import and use classes and methods from arbitrary C# libraries.

For FShark, F# has been selected as a source language for several reasons. First, most of F#s syntax is readily translatable into Futhark syntax, as long as the programmer stays away from using any of F#s non-functional constructs, like async or it's object oriented features. Second, as F# effortlessly interoperates with C# programs, and C# has plenty of OpenCL libraries available, we can write imperative OpenCL-powered programs in C#, for use in F# projects.

C#

C# C# C# C#

Futhark

Quoting from Futhark's own homepage,

> 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 optimising ahead-of-time compiler that presently generates GPU code via OpenCL, although the language itself is hardware-agnostic.

So far, plenty of handwritten GPU benchmark programs implemented in CUDA et al, has been ported to Futhark, with significant performance gains as a result. [**?**]. With these results in mind, it makes sense to start implementing other parallelizable algorithms and programs in Futhark. However, in the grand scheme of things, Futhark is still a relatively obscure programming language, and is almost solely used in academic settings.

With Futhark being a purely functional programming language, it has very few imperative language constructs available, and the few that it has, like in-place updates, are merely syntactic sugar for other existing library function calls.

As Futhark's main functionality is generating OpenCL kernels, it is in principle possible to compile Futhark programs for any language that are able to interface with the OpenCL API.

As a target language for `F#` translations, Futhark is ideal as we can identify and relatively easily translate a subset of the `F#` language to equivalent Futhark code, as the syntax itself is very similar. Even though `F#` also allows plenty of imperative and object oriented programming, `FShark` blocks the user from using these constructs, by failing at `FShark` compile time.

A primer on OpenCL

Chapter 3

The FShark subset

In the following chapter, we will describe the FShark subset. Although the subset is just a part of F#, we will describe it as if it was a language itself.

Figures 3.1 shows the statements available in FShark,

$$\begin{array}{lcl}
\mathit{prog} & ::= & \mathit{module}\ \mathit{prog} \\
 & | & \mathit{prog}'\ \mathit{prog} \\
 & | & \epsilon \\
\mathit{prog}' & ::= & \mathit{typealias} \\
 & | & \mathit{fun} \\
\mathit{progs}' & ::= & \mathit{prog}'\ \mathit{progs}' \\
 & | & \epsilon \\
 & & \\
\mathit{typealias} & ::= & \texttt{type}\ v\ = t \\
\mathit{module} & ::= & \texttt{module}\ v = \mathit{prog}'\ \mathit{progs}' \\
 & & \\
\mathit{fun} & ::= & \texttt{[<FSharkEntry>] let}\ \mathit{id}\ (v_1 : t_1)\ \ldots\ (v_n : t_n) : t = e \\
 & | & \texttt{let}\ v\ (v_1 : t_1)\ \ldots\ (v_n : t_n) : t' = e, \\
 & & \quad \textit{(for any } i \in 1..n,\ t_i \textit{ is not a tuple)}
\end{array}$$

Figure 3.1: `FShark` statements

$$\begin{array}{lcll}
e & ::= & (e) & \text{Expression in parens} \\
 & | & k & \text{Constant} \\
 & | & v & \text{Variable} \\
 & | & (e_0,\ \ldots,\ e_n) & \text{(Tuple expression)} \\
 & | & \{\texttt{id}_0 = e_0; \ldots ; \texttt{id}_n = e_n\} & \text{(Record expression)} \\
 & | & [|e_0; \ldots ; e_n|] & \text{(Array expression)} \\
 & | & v.[e_0] \ldots .[e_n] & \text{(Array indexing)} \\
 & | & v.\mathit{id} & \text{(Record indexing)} \\
 & | & v.\mathit{id} & \text{(Module indexing)} \\
 & | & e_1 \odot e_2 & \text{(Binary operator)} \\
 & | & -e & \text{(Prefix minus)} \\
 & | & \texttt{not}\ e & \text{(Logical negation)} \\
 & | & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \text{(Branching)} \\
 & | & \texttt{let}\ p = e_1\ \texttt{in}\ e_2 & \text{(Pattern binding)} \\
 & | & \texttt{fun}\ p_0\ \ldots\ p_n \rightarrow e & \text{(Anonymous function)} \\
 & | & e_0\ e_1 & \text{(Application)}
\end{array}$$

Figure 3.2: `FShark` expressions

$$\begin{array}{lcll}
p & ::= & \mathit{id} & \text{(Name pattern)} \\
 & | & (p_0, \ldots, p_n) & \text{(Tuple pattern)}
\end{array}$$

Figure 3.3: `FShark` patterns

$$\begin{array}{lcll}
t & ::= & \texttt{int8} \mid \texttt{int16} \mid \texttt{int} \mid \texttt{int64} & \text{(Integers)} \\
 & \mid & \texttt{uint8} \mid \texttt{uint16} \mid \texttt{uint} \mid \texttt{uint64} & \text{(Unsigned integers)} \\
 & \mid & \texttt{single} \mid \texttt{double} & \text{(Floats)} \\
 & \mid & \texttt{bool} & \text{(Booleans)} \\
 & \mid & (t_0 * \ldots * t_n) & \text{(Tuples)} \\
 & \mid & \{id_0 : t_0;\ \ldots;\ id_n : t_n\} & \text{(Records)} \\
 & \mid & t\ \texttt{array} & \text{(Arrays)}
\end{array}$$

Figure 3.4: `FShark` types

$$\begin{array}{lcll}
k & ::= & n\texttt{y} \mid n\texttt{s} \mid n \mid n\texttt{L} & \text{(8-, 16-, 32- and 64 bit signed integers)} \\
 & \mid & n\texttt{uy} \mid n\texttt{us} \mid n \mid n\texttt{UL} & \text{(8-, 16-, 32- and 64 bit unsigned integers)} \\
 & \mid & d\texttt{f} \mid d & \text{(Single and double precision floats)} \\
 & \mid & true \mid false & \text{(Boolean)} \\
 & \mid & (k_0,\ \ldots,\ k_n) & \text{(Tuple)} \\
 & \mid & \{id_0 = k_0;\ \ldots;\ id_n = k_n\} & \text{(Record)} \\
 & \mid & [|k_0; \ldots; k_n|] & \text{(Array)}
\end{array}$$

Figure 3.5: `FShark` literals

F# operators available in FShark

The F# subset chosen for FShark is described in this subsection. Note that all of these operators are overloaded and defined for all integer and floating point types in F#.

Arithmetic operators
The set of supported arithmetic operators is addition (+), binary subtraction and unary negation (-), multiplication (*), division (/) and modulus (%).

Boolean operators
FShark currently supports logical AND (&&), logical OR (||), less- and greater-than (<, >), less- and greater-or-equal (<=, >=), equality (=), inequality (<>) and logical negation (not).

Special operators
FShark also supports some of F#s syntactic sugar. These operators might not have direct Futhark counterparts, but their applications can be rewritten in Futhark for equivalent functionality. The supported operators are back- and forward pipes (<| and |>), and the range operator (e_0 .. e_1), which generates the sequence of numbers in the interval $[e_0, e_1]$. Note that in FShark, the range operator must be used inside an array as so [|e_0..e_1|] so we adhere to using arrays and not lists in our FShark programs.

Figure 3.6: FShark operators

F# standard library functions available in FShark

FShark supports a subset of the F# standard library. These are functions that are imported in F# modules by default.

Currently, bitwise operators like bitwise-AND and bitwise-OR are missing, but they should be relatively simple to add to the FShark subset, by adding them to the set of supported operators in the FShark compiler.

On the F# subset selected for FShark

For selecting the F# subset to support in FShark, I chose to look at what functions that were included in F#'s prelude. That is, the functions that are available in an F# program without having to open their containing module first. Fortunately, F# opens several modules by default of which I only needed to look in two different ones, to be able to support a reasonable amount of F# built-ins in FShark.

The primary module used in my supported F# subset is the module FSharp.Core.Operators. This module contained not only the standard arithmetic described in figure 3.6, but also most[1] of the functions shown in the figure 3.7. Except for unit type functions like failwith, exit and async, most of the functions and operators FSharp.Core.Operators have direct counterparts in Futhark's prelude, with equivalent functionality: All except for four of operators and functions chosen for FShark are in fact implemented in

[1] except for some convertion functions, found in FSharp.Core.ExtraTopLevelOperators

id
The identity function.

Common math function
The square root function (`sqrt`), the absolute value (`abs`), the natural exponential function (`exp`), the natural- and the decimal logarithm (`log` and `log10`).

Common trigonometric functions
Sine, cosine and tangent functions (both standard and hyperbolic): `sin`, `cos`, `tan`), `sinh`, `cosh` and `tanh`. Also one- and two-argument arctangent: `atan` and `atan2`.

Rounding functions
`FShark` supports all of `F#`s rounding functions: `floor`, `ceil`, `round` and `truncate`.

Number convertion functions
`FShark` supports all of `F#`s number convertion functions. For all the following functions t, $te = e', e : t_0, e' : t$, barring exceptions like trying to convert a too large 64-bit integer into a 32-bit integer.
The convertion functions available are `int8`, `int16`, `int`, `int64`, `uint8`, `uint16`, `uint`, `uin64`, `single`, `double`, `bool`.

Various common number functions
`min`, `max`, `sign` and `compare`.

Figure 3.7: `FShark` operators

Futhark's `math.fut` library. It was therefore aan obvious decision to support these functions and operators in `FShark`.

However, for the remaining four functions, that didn't have equivalents in Futhark's `math.fut`, their function calls are replaced with their identities instead. In example, whereas the `FShark` code

```
exp x
```

is written in Futhark as

```
exp x
```

because the exp function is also available in `math.fut`, the `FShark` code

```
cosh x
```

is rewritten as the full hyperbolic sine function instead, as so

```
((exp x) + (exp (-x))) / 2.0
```

These rewritings are not pretty to look at from a programmer's perspective, but `FSharks` Futhark code is not meant to be read by humans anyhow.

(MAYBE INVESTIGATE WHETHER INLINING THESE HAS PERFORMANCE PENALTIES)

The correctness of the `FShark` subset.

When transpiling code from one language to another, it is absolutely vital that the programmer can trust, that the resulting code in the target language is semantically equivalent to the source code. In `FShark`s case, it means that any program written using the `FShark` subset, must have the same result no matter whether it is run natively as `F#` code, or it is run as `FShark` compiled Futhark code.

I.e., one could imagine a programming language which had defined the function `log` not as the natural logarithm, but instead the binary logarithm. In such a case, the translation from that language to Futhark would still go without a hitch, and without any type errors to hint at the impending catastrophe. However, the native result with the Futhark result would be wildly different.

To ensure that every operator and function in the `FShark` subset has equivalent results, no matter whether the `FShark` code is run as native `F#` code, or compiled into Futhark, I have written a test suite with unit tests for each element in the `F#` subset.

(THESE ARE NOT ACTUALLY DONE YET) (Are unit tests enough?)

all convertion functions pass through i64. this might be a mistake, as real supports f32 to f64

thoughts on correctness of translations testing correctness of these translations

Chapter 4

FSharkPrelude

Besides defining an F# subset suitable for Futhark translation, it was also imperative to create a library of SOACs and array functions for FShark, to make it possible to write programs with parallel higher-order array functions.

Similarly to how the subset of math functions chosen from F# to include in the FShark was chosen, the SOACs and array function included in the FSharkPrelude has been picked directly from the Futhark libraries futlib/array.fut and futlib/-soacs.fut. The FSharkPrelude doesn't discriminate between array functions and SOACs, as maintaining and importing two different prelude files in FShark was needlessly complicated.

The FSharkPrelude consists of functions which are directly named after their Futhark counterparts, and have equivalent functionality. This prelude, together with the FShark subset, is what makes up the FShark language. When FShark developers are writing modules in FShark, they are guaranteed that their FShark programs has the same results, no matter whether their programs are executed like native F# code, or compiled and executed as Futhark.

The FSharkPrelude versions of Futhark functions are defined in three different ways.

1. Functions like the SOAC map and the array function length have direct F# equivalents, and are therefore implemented as calls to Array.map and Array.length respectively. For map for example, we have the following definition:

```
let Map f aa = Array.map f aa
```

2. Some Futhark SOACs, like reduce, takes a neutral element as one of the arguments in their function calls, whilst their F# counterparts (Array.reduce) does only take an operator and an array as arguments. To define the FShark SOAC so that it is equivalent to the Futhark version, it has been defined as so:

```
let Reduce (op: 'a -> 'a -> 'a) (neutral : 'a) (xs : 'a array) =
  let xs' = Array.append [|neutral|] xs
  in Array.reduce op xs'
```

Other functions, like the `map` functions which takes multiple arrays as arguments, require a bit of assembly first. For those `map` functions, we zip the arguments before using `Array.map` as usual:

```
let Map5 f aa bb cc dd ee =
  let curry f (a,b,c,d,e) = f a b c d e
  let xs = Zip5 aa bb cc dd ee
  in Array.map (curry f) xs
```

3. Lastly, some functions does not have `F#` counterparts. In example, we implement `scatter` using a for-loop:

```
let Scatter (dest : 'a array) (is : int array) (vs : 'a array) : 'a array =
  for (i,v) in Zip is vs do
    dest.[i] <- v
  dest
```

The complete list of available SOACs and array functions is available in appendix **??**.

Note that calls to `FSharkPrelude` functions are caught and exchanged for Futhark functions during the `FShark` compilation, as described in sec **??**.

Why is FSharkPrelude part of the `FShark` language?

Several of Futhark's SOACs, such as `map`, already has `F#` versions that are directly equivalent.

But there are several issues with just letting the `FShark` programmer use But many of these `F#` functions are contained in HER KOMMER DER MERE

1) it would be awkward to maintain a whitelist of accepted library functions, instead of simply handing the developer a gated library. 1½) uncomfortable to be told function is not supported at runtime

2) some Array functions have subtle differences compared to their futhark counterparts. In example, reduce doesn't take a neutral element which Futhark's reduce does.

3)

note; FSharkPrelude cannot detect bad operators in reduce. non-commutative ops in reduce goes bad when parallel, så deeeeeet

some implementations, like ZipN, are probably criminally ineffective.

How `FShark` SOACs differ from Futhark's ditto

On a surface level, `FShark` and Futhark SOACs are the same. After all, they have equivalent functionality. However, Futhark's SOACs gets special treatment in the

Futhark compiler, and are fused together where applicable. Take for instance the short code example in figure 4.1.

```
entry main : []f32 =
  let xs = iota 100
  let ys = map (f32.i32) xs
  let zs = map (+ 4.5f32) ys
  in zs
```

Figure 4.1: A short Futhark program consisting of just SOACs

For non-OpenCL programs, Futhark's compiler fuses all three expressions into one for-loop, as described in simplified Futhark C# code in figure 4.2. Similarly, in an OpenCL program, the short code example is translated into a single kernel, as shown in figure 4.3.

```
float[] mem = new float[100];
for (int i = 0 ; i < 100 ;i++)
{
  float res = int_to_float(i);
  res = res + 4.5f;
  mem[i] = res;
}
```

Figure 4.2: Figure 4.1 compiled as (simplified) non-OpenCL C# code.

In both of the compiled examples, we must first allocate a target array for our result, but note that although we obtain three different arrays in the original Futhark code, both of the compiled versions transform the iota expression into a for-loop instead, and inserts the operators from the two subsequent maps into the loop.

This is a concrete implementation Futhark fusion rules as defined in [**?**]; which states that $(map\ f) \circ (map\ g) \equiv map(f \circ g)$

However, executing FShark code as native F# code will execute the expressions as written, which means that we are allocating and writing to an array three times, once for each line in the program.

Futhark and nested maps

Futhark's compiler also specializes in parallelizing nested SOAC calls[**?**], which in example transforms nested map expressions into one single map expression. For Futhark programs like the one in figure 4.4, the resulting OpenCL program contains a single map kernel with $i * j$ active threads.

The F# compiler doesn't make any such transformations for FShark programs.

Arguing for Futhark-equivalent functionality

using a test suite with both positive and negative testing

```
__kernel void map_kernel(__global unsigned byte *mem)
{
   int global_thread_id = get_global_id();
   bool thread_active = global_thread_id < 100;

   float res;

   if (thread_active) {
       res = int_to_float(global_thread_id);
       res = res + 4.5f;
   }
   if (thread_active) {
       *(__global float *) &mem[global_thread_id * 4] = res;
   }
};
```

Figure 4.3: Figure 4.1, but compiled as a simplified OpenCL kernel.

```
let xss = map (\row ->
             map (fun col ->
               row * col
             ) <| iota j
         ) <| iota i
```

Figure 4.4: A nested FShark program

4.1 Arrays in F# versus in Futhark

As Futhark is an array language, designing the array handling for FShark was a non-inconsequential part of the design process. Whereas multidimensional arrays in Futhark are written as i.e. `[][]i32` for a two dimensional integer array, their actual representation in the compiled code is a flat array of bytes, and an array of integers denoting the lengths of the dimensions. Accessing the array at runtime can be done in $O(1)$, whether it's either at some constant or a variable index (i.e. `let second_x = xs[2]` or `let n = xs[i,j]`). The indexes are resolved during the Futhark compilation, either as scalars, or as a variable calculated from other variables.

RESEARCH WHETHER "RANDOM READS" ARE COALESCED IN A NICE WAY IN FUTHARK.

Functional languages like Haskell and F# mainly works with lists. In F#, lists are implemented as singly linked lists. Nodes in the list are dynamically allocated on the heap, and lookups take $O(n)$ time. We cannot make multidimensional lists, but we can make lists of lists: If we were to emulate a two dimensional list of integers in F#, we could use the type `int list list`. At runtime, the type would then be realized as a singly linked list of references to singly linked list of integers. For an `int list list` of $i \times j$ integers, we therefore have lookups in $O(i + j)$ time.

F# does also have arrays. The `System.Array` class itself is reference type. If we initialize an integer array in FSharp like so: `let arr = Array.create 10 0`, the type of `arr` is a reference to where it's corresponding array is located in memory. As the integers contained in the array are value types, the layout of the array referenced by arr is some initial array metadata, and then the ten integers stored in sequence.

We can access the array elements on $O(1)$ time, as indexing into the array is just done by accessing the array reference plus an index offset. If we want to emulate multidimensional arrays with these elements, we can create arrays of arrays (in .NET terms, these are called "jagged arrays"). In figure 4.5 we initialize a jagged array of integers.

```
let i = 8
let j = 5
let xss = Array.init i <| (Array.create j)

(* xss = [|
            [|0;0;0;0;0|];
            [|1;1;1;1;1|];
            [|2;2;2;2;2|];
            [|3;3;3;3;3|];
            [|4;4;4;4;4|];
            [|5;5;5;5;5|];
            [|6;6;6;6;6|];
            [|7;7;7;7;7|];
          |]
*)

let some_two = xss.[2].[3]
```

Figure 4.5: Initializing a jagged array of integers in FSharp

xss is an array of arrays, so xss is a reference to an array in memory, which itself contains references to other arrays. To retrieve the variable some_two, we first follow the reference to the array xss in memory. There we get the second element, which is a reference to another array in memory. In this array, we read the third element, which in this case is the 2 that we wanted.

The lookup takes $O(d)$ time[1], as we access arrays in $O(1)$ time, and have to follow d references to get to our element. If we just wanted a reference to the second array in xss, we would be chasing the first reference to arr, and then return one of the references stored within.

FSharp also offers actual multidimensional arrays. Instead of by initiabb DO MORE HERE

comparison between arrays: multidims are represented as single objects in memory, less cache misses. and so forth

WE HAVE NO WAY OF KNOWING WHERE THINGS ARE ALLOCATED

SHOULD WE LOOK AT PERFORMANCE OR LANGUAGE DESIGN FIRST?

[1] where d is the number of references we we are chasing to get our element.

4.2 Converting jagged arrays to Futhark's flat arrays, and back again

As mentioned in section 6.0.1, we cannot just pass jagged arrays as arguments to the Futhark `C#` entry functions. Instead, we must convert our jagged array into a flat array and an array of integers, and pass these two objects as arguments instead.

4.2.1 Analysis of FlattenArray

The simple algorithm for this flattening is described in pseudocode in figure 4.6. The implemented algorithm is slightly more complex, as it has perform various type castings, and also checks for invalid arrays such as irregular arrays. The implemented algorithm is available in the appendices.

When FlattenArray first is called with a jagged array as input, we don't know how many dimensions this array has. Therefore, we recursively call FlattenArray on the subarrays of the arrays, until these recursive calls reach a base case. The base case is the array that does not contain array references, but primitive values.

L2 : For a one dimensional jagged array, this branch is taken once. For a jagged array of d dimensions, it's taken $\prod_{n=1}^{d-1}(\text{subarrays at } d_n)$ times.

L3 is the base case, which takes $O(1)$ time. This is because we are just returning a tuple with the original array, and singleton array that holds the length of the array (creating the singleton array is also $O(1)$.)

L4 : For a jagged array of d dimensions, this branch is taken $\prod_{n=1}^{d-1}(\text{subarrays at } d_n)$ times.

L5 is the start of the recursive case. This line is called $O(d)$ times, d being the number of dimensions in the jagged array. The result of `map FlattenArray array` is an array of `a` array references and integer array references.

L6 MORE HERE

L7 simply retrieves a reference to the first array in the array of subarray lengths. This is $O(1)$.

L8 is by far the most costly line in the function. `F#`s `Array.concat` function takes a sequence of arrays, allocates a new array, and copies each element of the old arrays into the new array. Each of the n elements in the jagged array is copied to a new array a maximum of d times, which means we are performing $O(n * d)$ reads and writes.

L9 retrieves the length of an array, and is $O(1)$.

L10 appends a singleton array to the accumulated array of subarray dimensions, by first creating a singleton array, and then copy both the single element and the contents of the accumulated array to a third array of their collected length.

All in all, the upper bound on the `FlattenArray` algorithm is $O(n * d)$. This is a far cry from the performance of flattening in Futhark. Flattening is done in $O(1)$, as flattening merely calculates the product of the dimensions of the array, and returns the result as the new single dimension of the array.

```
FlattenArray (array : Array of a) : (Array of b * Array of int) =
  if a is not (Array of a):
    return (array, [len(array)])
  else:
    subarrays_and_lengths = map FlattenArray array
    (subarrays, subarrays_lengths) = unzip subarrays_and_lengths
    subarray_lengths = head(subarrays_lengths)
    concatenated_subarrays = concat subarrays
    this_length = len(array)
    lengths = [this_length] @ subarray_lengths
    return (contatenated_arrays, lengths)
```

Figure 4.6: Flattening jagged arrays, pseudocode

4.2.2 Analysis of UnflattenArray

The algorithm `UnflattenArray` in figure 4.7 restores the flat array from the Futhark `C#` program, to a jagged array in `F#`. Like in `FlattenArray`, the most expensive line in the function is the array-manipulating one. In `UnflattenArray`, it is line 7: For each dimension in the lengths array, we chunk our data array into multiple smaller arrays. Each of the n elements in the initial array is moved to a new and smaller array d times, which makes the complexity of this algorithm $O(n * d)$.

Why UnflattenArray hinders a specific tuple type

When an `FShark` function is invoked, it's arguments are prepared by an argument converter first. For scalar arguments, the argument is simply returned. But for array arguments, we must flatten the jagged array into a tuple that follows Futhark's array representation.

When the Futhark function returns, we then have to unflatten the Futhark arrays back into jagged arrays. To do this, we naively look at all the values returned by the Futhark function, and whenever we encounter a tuple of type `('a [] * int64 [])`, we assume that this is a flat array that needs to be unflattened. This procedure works fine, but has one side effect: `FShark` doesn't support entry functions that has `(('a [] * int64 []))` tuples in their return types, because this type is reserved.

To circumvent this, the user is instead encouraged to return the tuple as two separate values.

An alternative solution (FSharkArrays)

How they work

How they would alleviate the problem

```
UnflattenArray (lengths : Array of int) (data : Array of a) =
  if len(lengths) = 1:
    return data
  else:
    length = head(lengths)
    lengths' = tail(lengths)
    data' = chunk_array length data
    data'' = map (UnflattenArray lengths') data'
    return data''
```

Figure 4.7: Recreating a jagged array from flat array with dimensions

Why they weren't chosen anyhow (hint; needing to pepper FSharkArray all over code, would stand in way of idiomatic FSharp style)

Conclusion on arrays

Ultimately, choosing between jagged arrays, multidimensional arrays and FSharkArrays became a question of simplicity vs. performance. For `FShark`, I had the liberty to focus solely on simplicity, as `FShark` code is neither intended or even efficient when executed as native FSharp code. Therefore I could choose to let `FShark` use jagged arrays, instead of any of the other options.

The syntax for declaring a jagged array type closely resembles Futhark's multidimensional array syntax (take for instance FSharp's `int [] []` versus Futhark's `[][]i32` for declaring two-dimensional integer arrays). The close similarities between Futhark and `FShark` code means that `FShark` generated Futhark code is easier to read for debugging purposes, and likewise makes Futhark code easier to port to `FShark`.

Chapter 5

The `FShark` Compiler and Wrapper

Introduction

Parsing and building a regular F# program is trivial when using official build tools like `msbuild` or `fsharpc`. But in the case of `FShark`, we are not interested in the final result from the F# compiler, but merely its half-finished product.

As the F# Software Foundation offers the official F# Compiler as a freely available NuGet package for F# projects, we can use this package `FSharp.Compiler.Services` to parse the entire input `FShark` program and give us a Typed Abstract Syntax Tree of the FSharp expressions therein.

The F# Software Foundation actively encourages developers to create projects using the F# compiler library, they have published the collected F# compiler as a NuGet package, alongside a tutorial**??**on the usage of the various compiler parts.

For `FShark`, the Compiler Services package is used to compile a Typed Abstract Syntax Tree from a wellformed `FShark` source code file, which we then convert into- and print as a valid Futhark program. The Typed Abstract Syntax Tree is merely an AST that already has tagged all the contained expressions with their respective types.

We'll start with a detailed explanation of the `FShark` Compiler Pipeline.

The `FShark` Compiler Pipeline in practice

To examine the compiler pipeline in action, we'll go through the motions with the small example program displayed in figure 5.1.

We begin by constructing an instance of the `FShark`Wrapper. It has the following mandatory arguments:

`libName`

This is the library name for the `FShark` program. In the final Futhark `.cs`

```
module FSharkExample
open FShark.Main

[<EntryPoint>]
let main argv =
  let wrapper =
    new FSharkWrapper(
      libName="ExampleModule",
      tmpRoot="/home/mikkel/FShark",
      preludePath=
       ↪  "/home/mikkel/Documents/fshark/FSharkPrelude/bin/Debug/FSharkPrelude.dll",
      openCL=true,
      unsafe=true,
      debug=false
      )
  wrapper.AddSourceFile "../../srcs/ExampleModule.fs"
  wrapper.CompileAndLoad
  let xs = [|1;2;3;4|]
  let input = [|xs|] : obj array
  let xs' = wrapper.InvokeFunction "MapPlusTwo" input :?> int
   ↪  array
  printfn "Mapping (+2) over %A gives us %A" xs xs'
  0
```

Figure 5.1: An F# program using `FShark`

and `.dll` files, the main class will have the same name as the `libName`. This doesn't really matter if `FShark` is just used as a JIT compiler, but it's good to have a proper name if the user only wants to use the compiler parts of `FShark`.

`tmpRoot`

The `FShark` compiler works in its own temporary directory. This argument must point to a directory where F# can write files and execute subprocesses (Futhark- and C# compilers) which also has to write files.

`preludePath`

The `FShark` compiler needs the FShark prelude available to compile FShark programs.

`openCL`

Although Futhark (and therefore `FShark`) is most effective on OpenCL-enabled computers, the benchmarks in **??** still show a significant speed increase for non-OpenCL Futhark over native F# code. Therefore, `FShark` is also available for non-OpenCL users. Use this flag to tell `FShark` whether Futhark should compile C# with or without OpenCL.

`unsafe`

For some Futhark programs, the Futhark compiler itself is unable to tell whether certain array operations or SOAC usages are safe, and will stop the compilation, even though the code should (and does) indeed work. To enable these unsafe operations, pass a `true` flag to the compiler.

`debug`

> Passing the debug flag to the FShark compiler enables various runtime debugging features, for instance benchmarking the time it takes to run various parts of the compiler.

Now, we can pass a source file to the FShark wrapper, compile[1] it and load it into the FShark wrapper object.

To use the compiled FShark function, we must first wrap our designated input in an obj array. In this case, our chosen FShark function takes one argument, an int array. We define this array, and construct an argument array containing this single element. If the FShark function takes two arguments, we define an input obj array with two elements, and so forth. It is important to declare the input array as an obj array. Otherwise, F#s own type checker might very well faultily infer the input array as something else. In this particular case, input would've been inferred as being an int array array, until we declared its type specifically.

We then invoke the desired function through the wrapper. As all reflection-invoked functions return a value of type obj, we need to downcast this object manually. In this example, we use F#s downcast operator (:?>) to declare the return value as an int array. The actual return type is always the same as the return type declared in the source FShark file.

When FShark Wrapper Compiles

The general way to compile and load an FShark program into the FShark Wrapper, is by adding FShark source files to the wrapper object by calling the AddSourceFile method, and followingly calling the CompileAndLoad method. Although the FShark wrapper also offers other methods of loading and compilation, this is the primary one, as it initiates the entire FShark compilation pipeline.

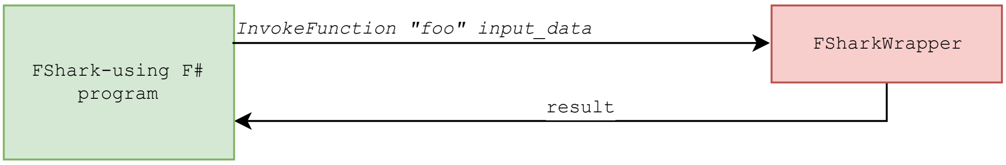


Figure 5.2: The FShark compilation pipeline

When calling CompileAndLoad, the supplied FShark source files are concatenated into one long source file, and written to a temporary location. An FSharpChecker is then initialized, so we can parse and type check the concatenated source code. The FSharpChecker is a class exported by the FSharp Compiler Services, and is a class that lets developers use part of the F# compilation pipeline at runtime.

We supply the FSharpChecker with the path to our precompiled FSharkPrelude assembly, and then call its ParseAndCheckProject method on to receive an assembly value, which contains the complete Typed Abstract Syntax Tree of our FShark program, in the form of an FSharpImplementationFileDeclaration.

[1] See subsection **??**

If the `FShark` developer followed the guidelines to write a well-formed FShark module, the main declaration of the program, the `FSharpImplementationFileDeclaration`, should contain a single `FSharpEntity`, which in turn contains all the remaining declarations in the program.

The declaration types within F#'s Typed AST

The `FSharpImplementationFileDeclaration` type has three union cases.

`InitAction of FSharpExpr`

`InitActions` are `FSharpExprs` that are executed at the initialization of the containing entity. These are not supported in `FShark`.

`Entity of FSharpEntity * FSharpImplementationFileDeclaration list`

An `Entity` is the declaration of a type or a module. In the case of `FShark`, three different kinds of entities are supported:

FSharpRecords are standard record types, and can be translated to Futhark records with ease. This entity has an empty `FSharpImplementationFileDeclaration list`.

FSharpAbbreviations are type abbreviations, and are easily translated into Futhark type aliases. This entity has an empty `FSharpImplementationFileDeclaration list`.

FSharpModules are named modules which contains subdeclarations. In this case, we retrieve the subdeclarations from the `FSharpImplementationFileDeclaration list`. The `FShark` compiler supports building FShark modules, but current limitations demands that modules are flattened when compiled to Futhark. This also means that function name prefixes in function calls are stripped when compiled to Futhark.

`MemberOrFunctionOrValue ofFSharpMemberOrFunctionOrValue * FSharpMemberOrFunctic`

F# doesn't differ between functions and values, which means that a function is merely a value with arguments. A pattern matched `MemberOrFunctionOrValue` value has the form `MemberOrFunctionOrValue (v, args, exp)`, where `v` contains the name and the type of the variable. If the `args` list is empty, `v` is simply a variable. If not, `v` is a function. `exp` contains the `FSharpExpr` that `v` is bound to. An `FSharpExpr` can be anything from a numeric constant to a very long function body.

In figure 5.3 we see a small but valid `FShark` program. It reads like a regular F# program, but contains the three vital parts that makes it usable as an `FShark` program.

- The module declaration on the first line declares that the following code is inside a module. In this case, we are declaring the module `ExampleModule`, although we could use any valid F# module name. As shown in figure 5.4, the top module declaration falls away during compilation, so only the top module contents are left.

```
module ExampleModule
open FSharkPrelude

module SomeValues =
  let Four : int = 4

  let SomePlus (x : int) (y : int) : int = x + y

[<FSharkEntry>]
let TimesTwo (x : int) : int =
  SomeValues.SomePlus x x

[<FSharkEntry>]
let MapPlusTwo (xs : int array) : int array =
  Map ((+)2) xs

let PlusSeven (x : int) : int =
  SomeValues.SomePlus x 7
```

Figure 5.3: A valid FShark program

- This open statement ensures that the F# Compiler Services has access to the FSharkPrelude during the compilation. It is possible to write an FShark program which doesn't use the FSharkPrelude, but this removes access to the SOACs that we use to write our data parallel programs.
- The [<FSharkEntry>] attributed function TimesTwo ensures that the resulting Futhark library from the FShark compiler has at least one entry point function. Without any entry point functions, we won't have any functions in the final compiled FShark program.

In figure 5.4 we see the resulting Futhark program. For now, we will ignore the transformations that have happened, except for two things: The Map function (called from FSharkPrelude) has been rewritten as the plain Futhark SOAC map in lowercase, and the module SomeValues has been flattened (see sec **??** for future plans.)

```
let Four : i32 = 4i32
let SomePlus (x : i32) (y : i32) : i32 =
  ((x i32.+ y))
entry TimesTwo (x : i32) : i32 =
  unsafe SomePlus(x) (x)
entry MapPlusTwo (xs : []i32) : []i32 =
  unsafe map (let x = 2i32 in
              (\(y : i32) -> ((x i32.+ y)))) (xs)
let PlusSeven (x : i32) : i32 =
  SomePlus(x) (7i32)
```

Figure 5.4: A valid FShark program, compiled to Futhark

This Futhark program is then stored in a temporary location in the user's file system, and compiled into as a library, using Futhark's C# compiler, either with or without OpenCL support. Finally after this compilation, we can invoke the resulting .dll file from within the `FShark`-using F# program.

Building **`FShark`** from the Typed AST

Only the supported FSharpExpr's has been listed here, but the full range of FSharpExpr's are available on [**?**].

FSharp-to-**`FShark`**IL rules

INTRODUCTION HERE

For these translations, we will disregard that all `FSharpExprs` are union cases of the F# data type `BasicPatterns`.

5.1 Design choices in writing the FShark Compiler

$$\begin{aligned}&\llbracket \mathit{Entity}(\texttt{IsFSharpRecord}, [(\mathit{field}_0 : \tau_0), \ldots, (\mathit{field}_n : \tau_n)]) \rrbracket\\ &= \texttt{FSharkRecord}(\texttt{[}(\mathit{field}_0 : \llbracket \tau_0 \rrbracket), \ldots, (\mathit{field}_n : \llbracket \tau_n \rrbracket)\texttt{]})\end{aligned}$$

$$\begin{aligned}&\llbracket \mathit{Entity}(\texttt{IsFSharpTypeAbbreviation}, \mathit{alias}, \tau) \rrbracket\\ &= \texttt{FSharkTypeAlias}(\mathit{alias}, \llbracket \tau \rrbracket)\end{aligned}$$

$$\begin{aligned}&\llbracket \mathit{Entity}(\texttt{IsFSharpModule}, [\mathit{decl}_0, \ldots, \mathit{decl}_n]) \rrbracket\\ &= [\llbracket \mathit{decl}_0 \rrbracket, \ldots, \llbracket \mathit{decl}_n \rrbracket]\end{aligned}$$

$$\begin{aligned}&\llbracket \mathit{MemberOrFunctionOrValue}((\mathit{name}, \tau^*, \mathit{IsEntryFunction}), [(\mathit{arg}_0 : \tau_0), \ldots, (\mathit{arg}_n : \tau_n)], e) \rrbracket\\ &= \mathit{FSharkVal}(\mathit{IsEntryFunction}, \texttt{FSharkFunction}([\llbracket \tau_0 \rrbracket, \ldots, \llbracket \tau_n \rrbracket], \llbracket \tau^* \rrbracket), \mathit{name}, [\mathit{arg}_0, .., \mathit{arg}_n], \llbracket e \rrbracket)\end{aligned}$$

Figure 5.5: Rules for translating FSharp declarations to FShark declarations

$$\begin{aligned}
\llbracket System.Int8 \rrbracket &= \texttt{FInt8}\\
\llbracket System.Int16 \rrbracket &= \texttt{FInt16}\\
\llbracket System.Int32 \rrbracket &= \texttt{FInt32}\\
\llbracket System.Int64 \rrbracket &= \texttt{FInt64}\\
\llbracket System.UInt8 \rrbracket &= \texttt{FUInt8}\\
\llbracket System.UInt16 \rrbracket &= \texttt{FUInt16}\\
\llbracket System.UInt32 \rrbracket &= \texttt{FUInt32}\\
\llbracket System.UInt64 \rrbracket &= \texttt{FUInt64}\\
\llbracket System.Single \rrbracket &= \texttt{FSingle}\\
\llbracket System.Double \rrbracket &= \texttt{FDouble}\\
\llbracket System.Boolean \rrbracket &= \texttt{Bool}\\
\llbracket System.Array\ \tau \rrbracket &= \texttt{FSharkArray}\ \llbracket \tau \rrbracket\\
\llbracket System.Tuple\ (\tau_0 \times \ldots \times \tau_n) \rrbracket &= \texttt{FSharkTuple}\ (\llbracket \tau_0 \rrbracket \times \ldots \times \llbracket \tau_n \rrbracket)
\end{aligned}$$

INSERT NOTE ON RULE FOR TUPLE ('a [] * long [])

Figure 5.6: Rules for translating .NET types to FSharkIL types

$$\begin{aligned}
\llbracket \texttt{FInt8} \rrbracket &= \texttt{i8}\\
\llbracket \texttt{FInt16} \rrbracket &= \texttt{i16}\\
\llbracket \texttt{FInt32} \rrbracket &= \texttt{i32}\\
\llbracket \texttt{FInt64} \rrbracket &= \texttt{i64}\\
\llbracket \texttt{FUInt8} \rrbracket &= \texttt{u8}\\
\llbracket \texttt{FUInt16} \rrbracket &= \texttt{u16}\\
\llbracket \texttt{FUInt32} \rrbracket &= \texttt{u32}\\
\llbracket \texttt{FUInt64} \rrbracket &= \texttt{u64}\\
\llbracket \texttt{FSingle} \rrbracket &= \texttt{f32}\\
\llbracket \texttt{FDouble} \rrbracket &= \texttt{f64}\\
\llbracket \texttt{Bool} \rrbracket &= \texttt{bool}\\
\llbracket \texttt{FSharkArray}\ \tau \rrbracket &= \texttt{[]}\llbracket \tau \rrbracket\\
\llbracket \texttt{FSharkTuple}\ (\tau_0 \times \ldots \times \tau_n) \rrbracket &= (\llbracket \tau_0 \rrbracket, \ldots, \llbracket \tau_n \rrbracket)
\end{aligned}$$

Figure 5.7: `FSharkIL` types to Futhark types

$$
\begin{array}{lcl}
\llbracket Const(obj, \tau)\rrbracket & = & \texttt{Const(}obj, \llbracket\tau\rrbracket\texttt{)} \\
\llbracket Value(v)\rrbracket & = & \texttt{Var(}v\texttt{)} \\
\llbracket AddressOf(v)\rrbracket & = & \llbracket v\rrbracket \\
\llbracket NewTuple(\_, [e_0, ..., e_1])\rrbracket & = & \texttt{Tuple([}\llbracket e_0\rrbracket, \ldots, \llbracket e_n\rrbracket\texttt{])} \\
\llbracket NewRecord((v_0 : \tau_0 * \ldots * v_n : \tau_n), [e_0, ..., e_1])\rrbracket & = & \texttt{Record([}(v_0, \llbracket e_0\rrbracket), \ldots, (v_n, \llbracket e_n\rrbracket)\texttt{])} \\
\llbracket NewArray(\tau, [e_0, ..., e_1])\rrbracket & = & \texttt{List(}\llbracket\tau\rrbracket, \texttt{[}\llbracket e_0\rrbracket, \ldots, \llbracket e_n\rrbracket\texttt{])} \\
\llbracket TupleGet(\_, i, e)\rrbracket & = & \texttt{TupleGet(}\llbracket e\rrbracket, i\texttt{)} \\
\llbracket FSharpFieldGet(e, \_, field)\rrbracket & = & \texttt{RecordGet(}field, \llbracket e\rrbracket\texttt{)} \\
\llbracket Call(\_, \texttt{GetArray}, \_, nil, [e_0, e_1])\rrbracket & = & \texttt{ArrayIndex(}\llbracket e_0\rrbracket, \llbracket e_1\rrbracket\texttt{)} \\
\llbracket Call(\_, name, \_, nil, [e_0, \ldots, e_n])\rrbracket & = & \texttt{Call(}name, [\llbracket e_0\rrbracket, \ldots, \llbracket e_n\rrbracket]\texttt{)} \\
\llbracket Call(\_, name, \_, \tau, [e_0, \ldots, e_n])\rrbracket & = & \texttt{TypedCall(}\llbracket\tau\rrbracket, name, [\llbracket e_0\rrbracket, \ldots, \llbracket e_n\rrbracket]\texttt{)} \\
\llbracket Call(\_, infixOp, \_, \tau, [e_0, e_1])\rrbracket & = & \texttt{InfixOp(}infixOp, \llbracket\tau\rrbracket, \llbracket e_0\rrbracket, \llbracket e_1\rrbracket\texttt{)} \\
\llbracket Call(\_, unaryOp, \_, \tau, [e_0])\rrbracket & = & \texttt{UnaryOp(}unaryOp, \llbracket\tau\rrbracket, \llbracket e_0\rrbracket\texttt{)} \\
\llbracket Let(v, e_0, e_1)\rrbracket & = & \texttt{LetIn(}v, \llbracket e_0\rrbracket, \llbracket e_1\rrbracket\texttt{)} \\
\llbracket IfThenElse(e_0, e_1, e_2)\rrbracket & = & \texttt{If(}\llbracket e_0\rrbracket, \llbracket e_1\rrbracket, \llbracket e_2\rrbracket\texttt{)} \\
\llbracket Lambda((v : \tau), e)\rrbracket & = & \texttt{Lambda(}v, \llbracket\tau\rrbracket, \llbracket e\rrbracket\texttt{)} \\
\llbracket Application(func, \_, [e_0, \ldots, e_n])\rrbracket & = & \texttt{Application(}\llbracket func\rrbracket, \texttt{[}\llbracket e_0\rrbracket, \ldots, \llbracket e_n\rrbracket\texttt{])} \\
\llbracket TypeLambda(e)\rrbracket & = & \llbracket e\rrbracket \\
\llbracket DecisionTree(\_, \_)\rrbracket & = & \texttt{Pass} \\
\llbracket DecisionTreeSuccess(\_, \_)\rrbracket & = & \texttt{Pass}
\end{array}
$$

Figure 5.8: Translation rules for FSharp expressions to FSharkIL expressions

$$
\begin{array}{lcl}
\llbracket Const(obj, \tau)\rrbracket & = & obj\llbracket\tau\rrbracket \\
\llbracket Var(v)\rrbracket & = & v \\
\llbracket Tuple([e_0, \ldots, e_n])\rrbracket & = & (\llbracket e_0\rrbracket, \ldots, \llbracket e_n\rrbracket) \\
\llbracket Record([(v_0, e_0), \ldots, (v_n, e_n)])\rrbracket & = & \{v_0 = \llbracket e_0\rrbracket,\ \ldots,\ v_n = \llbracket e_n\rrbracket\} \\
\llbracket List(\rrbracket\llbracket\tau\rrbracket, \texttt{[}\llbracket e_0\rrbracket, \ldots, \llbracket e_n\rrbracket\texttt{])} & = & [\llbracket e_0\rrbracket,\ \ldots,\ \llbracket e_n\rrbracket] \\
\llbracket TupleGet(\rrbracket\llbracket e\rrbracket, i) & = & \llbracket e\rrbracket.i \\
\llbracket RecordGet(field, e)\rrbracket & = & \llbracket e\rrbracket.field \\
\llbracket ArrayIndex(e_{arr}, [e_0, \ldots, e_n])\rrbracket & = & \llbracket e_{arr}\rrbracket\texttt{[}\llbracket e_0\rrbracket, \ldots, \llbracket e_n\rrbracket\texttt{]} \\
\llbracket Call(name, [e_0, \ldots, e_n]\texttt{)}\,\rrbracket & = & name\ (\llbracket e_0\rrbracket)\ \ldots\ (\llbracket e_n\rrbracket) \\
\llbracket TypedCall(\rrbracket\llbracket\tau\rrbracket, name, [\llbracket e_0\rrbracket, \ldots, \llbracket e_n\rrbracket]\texttt{)} & = & \llbracket\tau\rrbracket.name\ (\llbracket e_0\rrbracket)\ \ldots\ (\llbracket e_n\rrbracket) \\
\llbracket InfixOp(\rrbracket infixOp, \llbracket\tau\rrbracket, \llbracket e_0\rrbracket, \llbracket e_1\rrbracket\texttt{)} & = & (\llbracket e_0\rrbracket)\ \llbracket\tau\rrbracket.infixOp\ (\llbracket e_1\rrbracket) \\
\llbracket UnaryOp(\rrbracket unaryOp, \llbracket\tau\rrbracket, \llbracket e_0\rrbracket\texttt{)} & = & \llbracket\tau\rrbracket.unaryOp\ (\llbracket e_0\rrbracket) \\
\llbracket LetIn(\rrbracket v, \llbracket e_0\rrbracket, \llbracket e_1\rrbracket\texttt{)} & = & \texttt{let}\ v\ =\ \llbracket e_0\rrbracket\ \texttt{in}\ \llbracket e_1\rrbracket \\
\llbracket If(\rrbracket\llbracket e_0\rrbracket, \llbracket e_1\rrbracket, \llbracket e_2\rrbracket\texttt{)} & = & \texttt{if}\ \llbracket e_0\rrbracket\ \texttt{then}\ \llbracket e_1\rrbracket\ \texttt{else}\ \llbracket e_2\rrbracket \\
\llbracket Lambda(v, \llbracket\tau\rrbracket, \llbracket e\rrbracket\texttt{)}\,\rrbracket & = & \backslash(v : \llbracket\tau\rrbracket) \rightarrow \llbracket e\rrbracket \\
\llbracket Application(\llbracket func\rrbracket, \texttt{[}\llbracket e_0\rrbracket, \ldots, \llbracket e_n\rrbracket\texttt{])}\,\rrbracket & = & (\llbracket func\rrbracket)\ (\llbracket e_0\rrbracket)\ \ldots\ (\llbracket e_n\rrbracket) \\
\llbracket Pass\rrbracket & = & \epsilon
\end{array}
$$

Figure 5.9: `FSharkIL` expressions to Futhark

Chapter 6

The Futhark C# backend

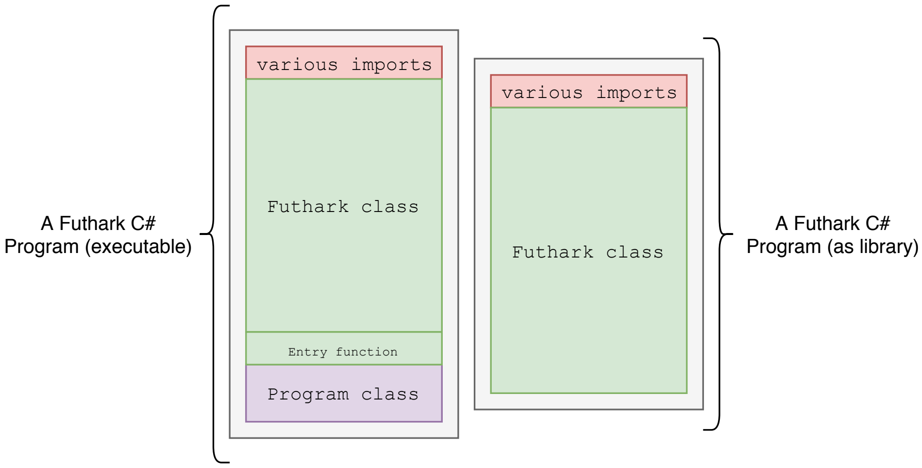


Figure 6.1: The two possible types Futhark C# programs

To be able to use Futhark with F# programs, it was necessary to compile Futhark programs to a language that F# could work with. Although the difference between running a compiled Futhark C- and C# executable from the command line is negligable, a Futhark C# backend would allow .NET projects to use Futhark libraries natively, instead of running their Futhark calculations through seperate C or Python modules.

Because F# has almost frictionless interoperability with C#, and C#s imperative constructs are very close to the intermediate code that Futhark generates for it's code generation, it was an easy decision to implement a C# generating backend for Futhark, to accompany the already existing C- and Python backends.

A Futhark backend must be able to do two different programs from a given Futhark program:

First, it must be able to generate standalone executables which can take input data from the `stdin` stream, and send the results to the `stdout` stream. Although a Futhark C,

-Python or C# executable should have equivalent functionality, their performance may vary, and the users may alter between the versions depending on which platforms that are available on their systems.

Second, and more interesting, it must be able to generate single file libraries which can then be imported and used in other C, Python or C# projects, in the same manner as any other library.

```
let main (xs : []i32) : []i32 = map (+2) xs
```

Figure 6.2: A very small Futhark program map2.fut

In example, if we compile the Futhark program in figure 6.2 as a Python library, we will be able to use it in a Python program, as showed in figure 6.3. Likewise, we would like to be able to do the same thing in a C# or an F# context.

```
import numpy as np
from map2 import map2

xs = np.array([1,2,3])
map2object = map2()
xs_res = map2object.map2(xs)
print xs_res # prints [3,4,5]
```

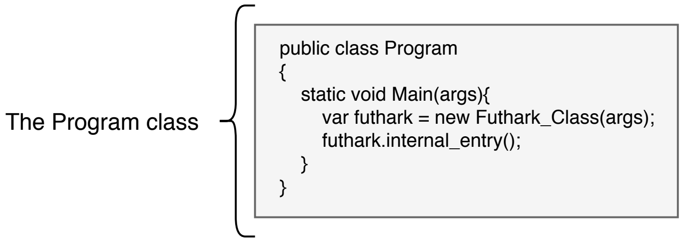
Figure 6.3: A very small Python program

The anatomy of a Futhark C# program

In figure 6.1, we see the two different ways we can compile a Futhark program to C#. They're largely the same, except for that the executable Futhark program must have a Program class with a Main method defined, so that there is an entrypoint defined for the compiled executable. Furthermore, the Futhark class in the executable version contains an entry function which chooses what Futhark function to run (in cases where the Futhark program has more than one entry function defined.)

The Program class itself (as seen in figure 6.4) is not especially interesting, and does only contain a main method which initialises the Futhark class, and calls the entry function inside the Futhark class. For both Futhark programs, the top consists of the various imports needed for the program.

This leaves us with the Futhark class itself. Figure 6.5 shows the different parts that make up the generated Futhark C# class. In the following sections we will walk through the individual parts.



```
public class Program
{
    static void Main(args){
        var futhark = new Futhark_Class(args);
        futhark.internal_entry();
    }
}
```

Figure 6.4: The FShark compilation pipeline

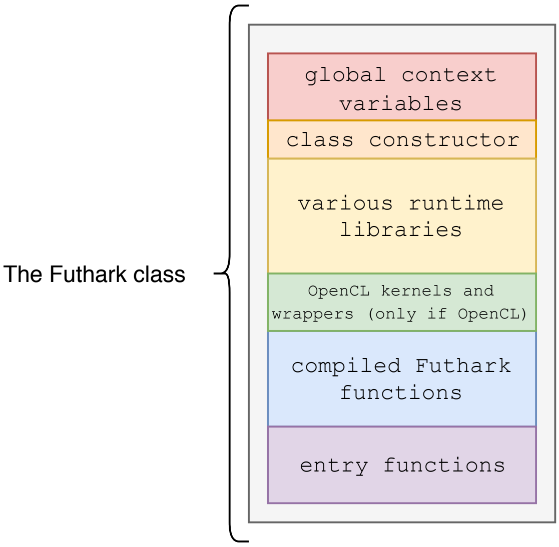

The Futhark class
global context variables
class constructor
various runtime libraries
OpenCL kernels and wrappers (only if OpenCL)
compiled Futhark functions
entry functions

Figure 6.5: The layout of the C# Futhark class

Global context variables

Compiled Futhark programs need to keep track of several variables. Both normal and OpenCL-enabled Futhark C# programs can take several options when they're launched from the command line. In example, num_runs tells the Futhark runtime how many times the chosen entry function should be executed, and the variable runtime_file tells the Futhark runtime where it should write timing information to, for example for benchmarking purposes.

Instead of passing an argument array along throughout all the functions in the Futhark class, like we usually would if we were writing purely functional programs, we instead set these arguments as class variables at class initialization, so we can refer to them everywhere throughout the rest of the class.

For non-OpenCL programs, the variables are exclusively for benchmarking and debugging purposes. For OpenCL programs however, the global variables are vital for the program's execution. In an OpenCL program, the Futhark class must keep track of two extra variables.

The struct futhark_context ctx is the struct that contains the global state of the current program's execution. Contained in the global state there is the current list of unused but allocated OpenCL buffers on the device, kernel handles for all the OpenCL kernels used in the Futhark program, and a counter for the total running time of the program. There is even another context contained in the `futhark_context`, namely the `opencl_context`, which contains the current state of the device, and also information about it's platform, it's queue and so forth.

The struct `futhark_context_config cfg` is similar to the `futhark_context`, but is only used for constructing the actual `futhark_context`.

The class constructor

The class constructor is necessary to setup the global variables needed throughout the Futhark class. When the Futhark program is compiled as an executable, the command line arguments are passed to the class constructor by the `Program` class. If the Futhark program is compiled as a library, the programmer can pass a string array of arguments to this constructor manually.

Besides setting class variables, OpenCL-enabled versions will initialize (and set) first the `futhark_context_config cfg` variable, and afterwards the `futhark_context` itself.

The various runtime libraries

The runtime libraries are a set of seperate `C#` files that are written and distributed through the Futhark compiler. When a Futhark program is compiled, these library files are concatenated and embedded directly into the rest of the generated code. They contain functionality which the generated Futhark programs depend on. The runtime libraries are the following:

`memory.cs`

As Futhark's stores all array values (no matter the dimensionality) as a flat one-dimensional byte arrays (with an accompanying array of 64-integers which denote the dimensions of the flat array), it was necessary to define a set of functions to interact with these byte arrays. I.e., `memory.cs` contains the `writeScalarArray` functions, which writes a scalar value to a byte array. The function is overloaded so it works with scalars of any integer or floating point primitive. See figure 6.6 for an example:

`scalar.cs`

This library contains all the scalar functions necessary for Futhark `C#` programs. In Futhark, arithmetic operators are defined for integers and floats of all sizes, and bitwise operators are defined for all integers. However, this is not the case in `C#`, where many arithmetic operators are only defined for 32- and 64 bit integers.

If these operators are used with 8- or 16 bit operands, the operands are implicitly casted to 32 bit integers at compile time, which also means that the final result of the operation is a 32 bit integer, which doesn't has the right type.

Therefore, wrapper functions must be defined for even the simplest arithmetic functions. I.e., integer addition in `C#` Futhark is actually four different functions:

```
void writeScalarArray(byte[] dest, int offset, double value)
{
    unsafe
    {
        fixed (byte* dest_ptr = &dest[offset])
        {
            *(double*) dest_ptr = value;
        }
    }
}
```

Figure 6.6: `writeScalarArray` writes a value at the specified offset in some byte array.

```
static sbyte add8(sbyte x, sbyte y){ return Convert.ToSByte(x + y); }
static short add16(short x, short y){ return Convert.ToInt16(x + y); }
static int add32(int x, int y){ return x + y; }
static long add64(long x, long y){ return x + y; }
```

Besides, `scalar.cs` also contains the `C#` definitions for the various mathematical functions from Futhark's `math.fut`library, such as `exp`, `sin` and `cos`.

`reader.cs`

The reader contains the entire functionality for recieving function parameters through `stdin`. The reader reads scalars of any of the Futhark-supported primitives, and also arrays and multidimensional arrays of scalars. The reader also supports reading streams of binary data. It is only necessary for Futhark executables.

`opencl.cs`

MAYBE WRITE ALL OF THIS ALSO? ALRIGHT THANKS

The compiled Futhark functions

The compiled Futhark functions are the Futhark Intermediate Code functions, expressed in the target language, and corresponds to the entry functions found in the entry functions-section of the Futhark class. Only the Futhark `entry` functions are compiled to individual functions, and remaining helper functions are inlined here.

In OpenCL programs, all array functions and SOAC calls are compiled as individual (or fused) OpenCL kernels. Therefore, the compiled Futhark functions in these programs consists of mainly some scalar operations and memory allocations, and calls to Futhark-generated kernel wrapper functions.

In non-OpenCL programs, the array functions and SOAC calls are not stored in seperate wrapper functions, but inlined in the Futhark functions.

OpenCL kernels and wrappers

If the Futhark program is compiled for OpenCL, all array handling function- and SOAC calls are compiled as OpenCL kernels. This part of the Futhark class has two parts:

1. The string (actually a single string in an array) `opencl_prog`, which contains the entire Futhark-generated OpenCL source code for the Futhark program in question. This source code contains all the OpenCL kernels for the program, and is passed to the OpenCL device, compiled and loaded, when the Futhark class is initialized. Handles to the individual kernels are then stored in the `futhark_context`.
2. For each kernel in the `opencl_prog`, the Futhark compiler generates a kernel wrapper function. These wrapper functions takes the kernel arguments (such as scalar values, array values and indexes) as input, and performs all the OpenCL specific work necessary for the actual kernel launch; in example setting the kernel arguments on the device, and copying data back and forth between host and device buffers.

6.0.1 Entry functions

Futhark's internal representation of array values are one dimensional byte arrays (which can represent arrays of any type and dimensionality), and an accompanying list of integers denoting the lengths of the array's dimensions. However, Futhark does not expect it's users to pass this form of arrays as function arguments, which is why each Futhark `entry` function has a corresponding entry function in the final compiled code.

To discern between Futhark functions and entry functions, the Futhark function's name is prefixed with "`futhark_`", as in for example "`futhark_foo`". Depending on whether the Futhark program is compiled as an executable or a library, the entry function itself is then named "`entry_foo`" or just "`foo`".

For executables, "`entry_foo`" is a function that doesn't take any arguments. Instead, it uses the reader functions from `reader.cs` to parse the arguments for "`foo`" from `stdin`, and passes them to the Futhark function. For all array values in the arguments, the array values are converted into Futhark representations of them. When the Futhark function returns the result, the result is then printed to `stdout`.

For libraries, the "`entry_`" prefix is dropped, and the function just takes care of converting array arguments into and back-from their Futhark representations.

MENTION THE INPUT ARGUMENTS SPECIFICALLY ARRAYS THAT HAVE TO BE FLATTENED FIRST

The C# backend, compared to the C- and Python counterparts

THE PYTHON BACKEND HAS MUCH FUNCTIONALITY ENCAPSULATED IN PYOPENCL, AND DOESN'T NEED TO DECLARE VARIABLES BEFORE SETTING THEM LESS COMPLEX GENERATOR NEEDED AS VARIOUS OPENCL STATEMENTS ARE HANDLED AUTOMATICALLY BY LIBRARY

C BACKEND MUST BE AWARE OF ALL SIZES AND EVERYTHING AT COMPILE TIME, WHICH MEANS STATES MUST BE ALLOCATED THROUGH COMPLEX STRUCTS AT COMPILE TIME, AND STRUCTS MUST BE DEFINED AT COMPILE TIME AS WELL

C ALLOWS NULL POINTERS, CS DOES NOT WHICH MEANS WE NEED PLACEHOLDER VARIABLES

CSHARP GENERATOR IS SOMEWHERE INBETWEEN AS IT IS CAN HANDLE OBJECTS WHICH CAN CARRY STATE, FURTHERMORE DYNAMIC MEMORY ALLOCATION

Memory management in Futhark C#

As Futhark stores array values around in byte arrays, it is relevant to compare the difference between how the array handling differs between Futhark's C backend, and this C# backend. For OpenCL programs, the memory management of C# and C is largely the same, as the OpenCL side of these programs are the same. C# does after all just use C bindings for it's OpenCL interactions.

However, for non-OpenCL C# programs, we have to take C#'s memory model into consideration

C implicitly allows unsafe programming. In this case, it means interacting with system memory by reading and writing arbitrary values from/to arbitrary locations, designating the values and destinations as whatever type we want. In figure 6.7, we see a for-loop that performs a summing scan on an array of integers. On line 6, reading from right to left, we are first creating reference to a location in the byte array xs_mem_4223. However, as the reference is a pointer to a byte in the array, we must recast it as an int32_t pointer. After we do this, we can finally derefer the pointer to retrieve a four byte integer from the byte array.

We add the retrieved integer to our accumulating variable scanacc_4187, before we cast a reference in our destination byte array as an integer pointer, and store the result there.

WHY IS THIS NOT ALLOWED?

DESCRIBE TWO DIFFERENT WAYS OF DOING IT ANYHOW 1) MARSHAL 2) unsafe and fixed

what was chosen and why

```
memblock mem_4226;
memblock_alloc(&mem_4226, bytes_4224);
int32_t scanacc_4187 = 0;

for (int32_t i_4189 = 0; i_4189 < sizze_4135; i_4189++) {
    int32_t x_4147 = *(int32_t *) &xs_mem_4223[i_4189 * 4];

    scanacc_4187 += x_4147;

    *(int32_t *) &mem_4226[i_4189 * 4] = scanacc_4187;
}
```

Figure 6.7: A short snippet from a Futhark C program

Selection an OpenCL interface for C#

OpenCL interaction is not a part of the .NET standard library, but several libraries do exist for .NET/OpenCL interactions. For this thesis, I researched a selection of these libraries, to determine which one that would fit the best for my purposes. As Futhark depends on being able to interface with the OpenCL platform directly, it was necessary to find an OpenCL library for .NET which had direct bindings to the OpenCL developer library.

The .NET libraries I took into consideration was NOpenCL, OpenCL.NET and Cloo. All three libraries have been designed to aide OpenCL usage in C# programs, by simplifying OpenCL calls behind methods GØR BEDRE.

NOpenCL

NOpenCL was the first candidate for the C# backend, and had several advantages to the other two: As per February 2018, it had been updated within the last year, and was therefore the least deprecated library. Second, the NOpenCL repository on Github contains both unit tests and example programs.

However, NOpenCL is also tailored for Windows use, and therefore not a good fit for Futhark, as Futhark is available on both Windows, Linux and Mac OS. Furthermore, the library is not available through the NuGet package manager, and the OpenCL API calls are needlessly complex to work with through the library.

OpenCL.NET

OpenCL.NET also has a test suite, is available through NuGet, and is used as the backend for other libraries, such as the F# GPU library Brahma.FSharp**??**.

However, this library hardcoded to work on a in a Windows context, and has not been updated for more than five years.

Cloo

Cloo is usable on all three platforms, and it is available on NuGet. Furthermore, as opposed to the other two libraries, the Cloo library contains a class with static functions that does nothing but passing arguments on to the OpenCL library, using C#s DllImport attribute. It is immediately possible to skip most of Cloos features, and just use the library for it's OpenCL bindings.

Even then, Cloo has not been updated within the last five years, and probably won't be in the future either.

Given these three candidates, I chose to work with Cloo: It was the only one that had the necessary OpenCL bindings readily available, and the only one that was platform agnostic.

Writing a custom OpenCL bindings library

Though Cloo is a good fit for Futhark C#, it is also slightly risky to depend on a five year old unmaintained library in a modern project. Therefore, it could be a good idea to write a smaller library similar to Cloo, specifically for Futhark - or maybe even just include it with Futhark as one of the C# runtime libraries.

Chapter 7

Benchmarks and evaluation

`FShark` generated Futhark compared to original Futhark code

Appendices show

The `LocVolCalib` benchmark

small.in: FShark (openCL) took 211882 microseconds. Average invokation (fshark non openCL) time was 81194767 microseconds. Native took 438 929 311 microseconds.

medium.in: (Fshark opencl)invokation time was 310833 microseconds Fshark nonopencl Average invokation time was 154 141 321 ms Native took 900 643005 microseconds.

large.in:

fshark with opencl Memory Allocation Error fshark sans opencl 2450 637 053 microseconds Native took 24757 874 577 microseconds.

for all three datasets

The `nbody` benchmark

for all three datasets

Specifications for benchmark

We have run the benchmarks on a system with these attributes:

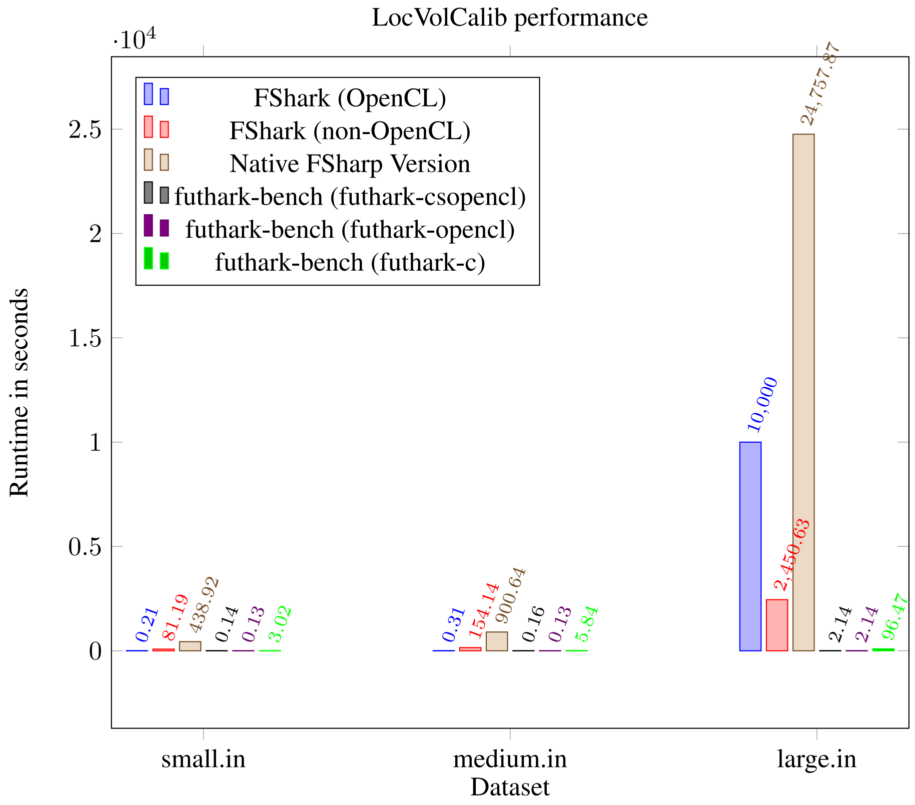


Figure 7.1: Comparison between Python and Futhark performance for simple model

- CPU: 4 cores of Intel Core i5-6500 at 3.20GHz
  - L1 cache: 128 KiB
  - L2 cache: 1024 KiB
  - L3 cache: 6144 KiB
- GPU: GeForce GTX 970

Introduction for the two benchmarks LocVolCalib and nbody

why are they faster in general

Chapter 8

Current limitations

Chapter 9

Related work

Chapter 10

Conclusion and future work

Chapter 11

FSharks interoperability between F#and Futhark (C#)

FShark stands on three legs: The FShark compiler, the Futhark C# code generator, and the FSharkWrapper. The compiler is responsible for compiling FShark source code into Futhark source code, and the C# code generator takes the result Futhark source code, and compiles OpenCL powered C# libraries, which can be imported directly back into F#.

It is of course possible to use the compiler and the code generator as individual modules, but for this project, the FSharkWrapper has been designed to let users use FShark without having to understand any of the underlying pipeline.

To illustrate this; take a look at figure **??**. In the first line, the user initializes the FSharkWrapper with the arguments necessary to use the wrapper itself. In the second line, the user adds a source file to the wrapper by it's path. In the third line, the user tells the wrapper to run the compilation pipeline. Assuming that the compilation goes well, the user can then invoke some function from the FShark program in line four.

Here, calling the `CompileAndLoad()` function triggers the entire `FShark` pipeline as described in **??**, and does then have a function available for the user to call afterwards.

However, as this is the default way of using `FShark`, we are currently calling `CompileAndLoad()` every time we use the `FShark` program. This is happening even though we only need the final compiled C# assembly to load back into `F#` at runtime.

Everytime we run the FShark compiler pipeline, we are therefore also

1. parsing, typechecking and generating a TAST from the `FShark` code, using FSharp's compiler.
2. generating Futhark source code from the FSharp TAST
3. Writing the Futhark source code to disk
4. running the Futhark compiler and C# code generator on the Futhark source code

5. running the mono C# compiler on the resulting C# source code

For two selected benchmarks we have the following times

11.0.1 Pros and cons of the current design

As there are demonstratively great performance gains to be won by only using the compiler part of the FShark pipeline, it is worth discussing whether the rest of the FShark pipeline should remain.

Besides eliminating the entire compilation operation at every `FShark` execution, a compiler-only approach to the `FShark` compiler would give us the following advantages:

- **Standalone-modules first:** As the compiler is now only used once, the resulting Futhark assembly is readily importable in any .NET project, as long as the required Mono libraries are also available. This goes not only for the user who just compiled the assembly, but also for any other user who has acquired the necessary Mono libraries. This means that the `FShark` developer can use and share the `FShark` assemblies with colleagues and coworkers like any other sharable .NET library, as this is indeed what a compiled Futhark `C#` library amounts to.

  Corollarily; this `FShark` design would make `FShark` is useful for generating high-performing .NET libraries. (Although one could write such libraries in Futhark instead of `FShark`.)

- **Static typing of `FShark` module:** The current runtime-only approach means that the user must rely on reflection to call `FShark` functions. In this situation, all modern IDE comforts like autocompletion, and especially static type checking and inference falls away. For the following example , the current design demands that we first wrap our arguments in an `obj array`, before invoking the function `foo`. Furthermore, we must also downcast the result using `F#`s downcasting operator `:?>`. Because we are upcasting our arguments to an `obj array`, we can actually pass any (correctly casted) array as an argument to our reflection-invoked function, without triggering any type errors at compile time. The same goes for the downcasted result from the function. We can cast the result as whatever type we like, and not run into any trouble until we finally run the compiled program. However, if we use `FShark` to generate assemblies instead, we now have all the type information available at compile time. Our compiler will block us from compiling the program by giving us useful type errors. Last but not least, we can remove all the casting operations that are littering the program.

- **Getting rid of, or at least trimming down, the FSharkWrapper:**

However, the current design of `FShark` also has some advantages that follows automatically from the design.

- **Rapid `FShark` code development:** Currently, it is recommended that any `FShark` code for a project is also built as part of the project. By including the `FShark` file in the original project's source list, we can call the `FShark` module natively, without running the `FShark` compiler, to prototype and debug the `FShark` code directly in our IDE, before we switch to using the compiled version of the `FShark` code.

- **En mere**

11.1 The future of `FShark` interoperability

With these considerations in mind, my future work on `FShark`s interoperability consists of reducing FSharkWrapper in size, so it only takes an `FShark` source path and a .NET assembly outpath as inputs, and does nothing more than orchestrating the `FShark`-, the Futhark- and the `C#`compiler. The current design is too complex, largely from supporting too many superflous features like concatenating multiple sources, and so on.

I will also be researching the optimal way to keeping the `FShark` module development as close to the rest of the `FShark`-using project as possible, without

The current design enables direct prototyping, which must of course be kept in later versions of `FShark`.