UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's thesis

Mikkel Storgaard Knudsen

FShark

Futhark programming in FSharp

Advisor: Cosmin eller Troels

Handed in: July 31, 2018

Abstract

Here is a nice abstract
prut prut

Contents

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).

As performance increases in single-CPU design has stalled due to the power wall[**?**] (among other things), developers are turning to multi-core processors instead. As the number of cores increases, so does the number of active threads available for parallel data processing.

Modern mainstream GPUs can run tens of thousands of threads in parallel. Modern mainstream CPUs, like the current Ryzen series by AMD, usually support between 10 and 20 simultaneous threads. This makes GPUs the optimal target for data-parallel programming.

GPU programming is complicated: GPU-targeting developers must not only write the computational kernels for the GPUs, but also often manually handle the memory allocations and -transfers between the main program and the GPU device. Such difficulties in GPU development, compared to normal (sequential) CPU development, severely hinders the adaption of GPU programming in general.

Even though most programming languages support GPU programming through various libraries, there are very solutions that offers GPU programming through high level programming - the users still have to write their own kernels in some form, and likewise declare their own buffers.

Two mainstream languages which lack high level GPU programming solutions are `C#` and `F#`.

It is safe to say that there exists plenty of `C#` and `F#` projects in the real world, which could greatly benefit from parallelizing parts of their algorithms, but current solutions would then demand that those parts in particular should be rewritten at least partly as GPU code, depending on the libraries used. Depending on someone to have non-mainstream GPU coding skills on a conventional developer team is not feasible, so the benefits from parallelizing are often left alone in favor of maintaining a more accessible code base.

Currently, there does exist plenty of high level solutions to this problem. In particular, numerous domain specific languages exists that allows programmers to solve their domain specific problems in a high level language, and compile it to standalone GPU accelerated libraries or programs.

Of such DSLs we have for instance:

- Forma
- Ebb
- one more

However, DSLs such as these are always either embedded in some host language (such as Accelerate), or compiled to standalone executable programs. Their compilers are designed to optimize performance, but rarely to support interoperability to any significant degree.

Projects like APLtail[**?**] have shown new ways to obtain GPU-accelerated executables and libraries from source code written in high level language. APLtail parses and compiles APL[1] code into redistributable C or Python libraries.

In summary, the hardware for massively parallel programming is widely available. Furthermore, solutions exists for writing efficient GPU programs in high level languages, but these have weak interoperability support with mainstream languages.

1.1 What `FShark` sets out to do

This thesis takes inspiration from APLtail[**?**], and creates a solution that lets users compile efficient GPU programs from a high level programming language, whilst at the same time supporting a high level of interoperability with a mainstream language. Whereas APLtail allows integration of GPU Futhark-written computation kernels in C- and Python programs (by means of code generation), we would like to use code generation to make Futhark-written kernels available for use in `C#` and `F#` programs.

To show that this is feasible, we first design a `C#` code generator for the Futhark compiler. This code generator must be able to generate `C#` source files, that can be compiled and used either as standalone executables, or as importable libraries in any other `C#` or `F#` program. There were several notable challenges in this process, namely 1) designing `C#` programs that could encapsulate entire Futhark programs in a single class, and 2) designing helper libraries to include in the generated code, and 3) designing a way to write sequential (non-GPU) Futhark code as pure `C#`, in cases where GPU devices are unavailable.

The code generator alone should not convince anyone that we are creating GPU kernels from a high level language, which is why we also design and implement a compiler which takes source code written in a mainstream language, compiles it as efficient GPU kernels, and (together with the code generator) makes the resulting GPU program immediately operable from the mainstream language itself. The main challenges for this was 1) identifying which parts of the `F#` language that were suitable for Futhark translation, 2) implementing a standard library for Futhark targeted `F#` programs, and 3)

[1] more accurately a subset of the APL language

designing and implementing a compiler pipeline that would let users program and use GPU kernels in `F#`, without manually using Futhark compilers or importing external libraries.

Empirical evaluation demonstrates that this approach is feasible. We both show that that unit tests written in a high level language can be compiled and executed correctly as computational kernels on the GPU, just as we also take complex benchmark programs written in a mainstream language, compile them into computational kernels for the GPU, and use them directly in the mainstream language afterwards.

The contributions of this thesis

The contributions of this thesis are as follows:

1. A `C#` code generator for the Futhark language compiler, which generates GPU accelerated libraries that can integrate seamlessly in `C#` and `F#` code bases.
2. A select subset of the `F#` langauge which can be translated directly to Futhark source code of equivalent functionality. This includes a library which implements Futhark SOACs[**?**] in `F#`, allowing people to write `F#` code which can be ported automatically to Futhark.
3. A compiler and wrapper pipeline which allows users to compile individual `F#` modules in their projects to GPU accelerated libraries, and load and execute code from these modules in the rest of the `F#` project.
4. A set of benchmarks and unit tests that shows that this approach is indeed feasible.

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 first show two languages for GPU programming, namely CUDA and Futhark. Then we will show `C#` and it's interoperability with Futhark. Finally, we will take a look at `F#`, and how we can expect Futhark/`F#` interoperability.

2.1 CUDA

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.

2.1.1 A simple CUDA program

Take for instance the function $f(x, y) = ax + y$. In figure 2.1 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.

The CUDA kernel

Line 3's `__global__` signifies that the following function is a CUDA kernel. Line 4 to 10 contains the computational kernel for the GPU. Line 4 contains the kernels name and arguments. The kernel takes as arguments *a* is a scalar constant, the pointers *x* and *y* are references to floating point arrays in the GPU device memory, and *n* denotes the length of the arrays *x* and *y*.
Line 6 computes the current computational thread's global id i. If we compare a parallel computational kernel with a sequential for-loop, this global id is the iterator variable. Line 8 performs the actual $f(x, y)$ calculation, and stores the result in the y array, but only if the if-clause in line 7 is true. As we have tens of thousands of threads running simultaneously on the CUDA device, we only want to perform any array operations if

we know that our current global id is within the length of the array.

The CUDA main function

We also need a main function to run the kernel:
Line 14 sets N to $1 << 20$ (or 2^{20}).
Line 16 and 17 allocates memory for two arrays x and y in system memory.
Line 19 and 20 allocates memory for two arrays x and y on the CUDA device (the GPU.)
Line 22 to 25 initializes the arrays x and y with scalar values 1.0 and 2.0.
Line 27 and 28 copies our arrays from system memory to the corresponding arrays on th CUDA device.
Line 31 executes the `saxpy` kernel.
Line 33 copies the result from CUDA device back to the y array in the system memory.
The remaining lines frees the allocated memory, first from the CUDA device and then from the system memory.

```
  #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 2.1: $ax + y$ in CUDA

2.2 Futhark

Whereas the CUDA program and kernel contained large amounts of memory handling and bounds checking, a similar program written in Futhark spares us for a lot of the manual labor above. Figure 2.2 contains a Futhark program that is semantically equivalent to the CUDA program, in regards to the computational result.

```
1  let saxpy (a : f32) (x : f32) (y : f32) : f32 =
2    a*x+y
3
4  entry main =
5    let N = 1<<20
6    let a = 2f32
7    let xs = replicate N 1f32
8    let ys = replicate N 2f32
9    let ys' = map2 (saxpy a) xs ys
10   in ys'
11
```

Figure 2.2: $ax + y$ in Futhark

Line 1 to 2 defines a function that takes three floats (a, x and y) and returns $ax + y$.

Line 4 to 10 defines our main function.
In line 4 we use `entry` instead of `let` to tell the compiler that `main` is an entry point in the compiled program. This means we can call this function when we import the compiled program as a library, as opposed to the function `saxpy`, which cannot be accessed as a library function.
Line 5 sets N to $1 << 20$ (or 2^{20}).
Line 6 sets a to 2.0.
Line 7 uses the built-in function `replicate`[1] to generate an N element array of 1.0
Line 8 uses the built-in function `replicate` to generate an N element array of 2.0
Line 9 uses the built-in function `map2` to apply the curried function (`saxpy a`) to the arrays xs and ys.

`map f xs` has the type `((a -> b) -> []a -> []b)`, and returns the array of f applied to each element of xs.

`map2 f xs ys` is very similar, but has the type `((a -> b -> c) -> []a -> []b -> []c)`, and applies f to the elements of xs and ys pairwise.
In this case, we are calling `map2` with the function `(saxpy a)`, which is just `saxpy` with the first argument a already defined.

When we compare the program in figure 2.1 to the same program written in Futhark (figure 2.2), we quickly see how Futhark's high level declarative approach is simpler and less verbose than CUDA's. The Futhark compiler does the heavy lifting, by parsing Fuhark source code and generating OpenCL code and wrapping them in standalone C- or Python programs.

2.3 F#

`F#` is a high level multi-paradigm programming language in the .NET family. `F#`s syntax follows a classical functional programming style. For instance, this means we

[1] replicate has the type `int -> a -> []a`

can take (some programs) written in Futhark, and port them to F# in a way that very closely resembles the original Futhark code.

Figure 2.3 shows the Futhark program from **??** written in F#.

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

let main =
  let n = 1<<<20
  let a = 2.0f
  let xs = array.replicate n 1.0f
  let ys = array.replicate n 2.0f
  let ys' = array.map2 (saxpy a) xs ys
  in ys'
```

Figure 2.3: $ax + y$ in Futhark

F# also supports object oriented programming, and has seamless interoperability with the rest of the .NET language family. We can therefore readily use C# libraries and classes in F#, and vice versa.

C#

C# C# C# C#

Chapter 3

Architecture and use cases

3.1 Generating GPU accelerated libraries

In this thesis, we are interested in making Futhark-generated GPU kernels available in `C#` programs. Currently, Futhark source code can be compiled to C and Python code. In example, the Futhark C compiler follows the basic architecture shown in figure 3.1. When we call the Futhark C compiler `futhark-c` on a Futhark source file, the compiler reads the source code file (and possibly imports) into the compiler. The compiler optimizes the input program, and expresses it in a compiler-internal intermediate language. This intermediate language code is then passed to a code generator, which is expresses the intermediate language code in C, and writes the result to a C source file. This file can then be used in any other C program.

For Python compilation, exchange "C" for "Python".

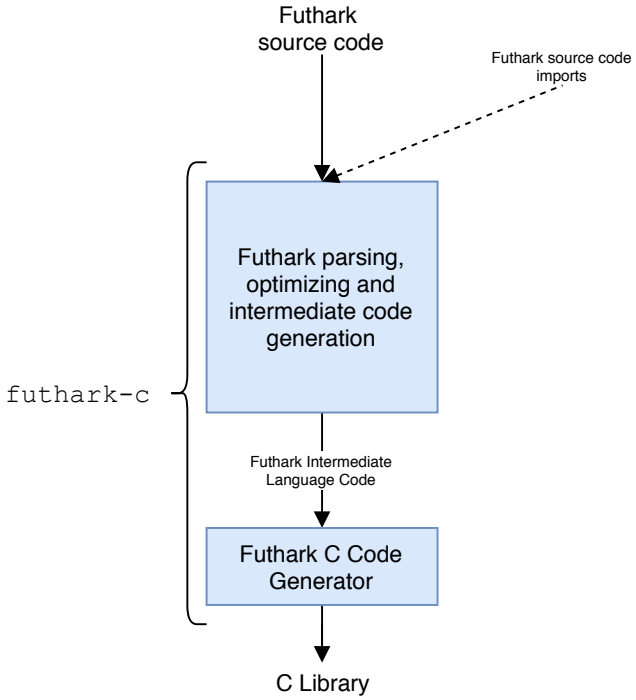


Figure 3.1: The Futhark-to-C compilation pipeline

Using Futhark in Python

We will now describe a use case for the Futhark-to-Python compiler:

1. We write a short Futhark program, which has a single entry function available. This program takes an array of integers, and adds 2 to each element in the array. The program is shown in figure 3.2.
2. We then compile the Futhark program into a library file, by calling the Futhark compiler from the command line, like shown in figure 3.3. This compiles `mapPlus2.fut` to a Python file called MapPlus2.py
3. Finally, we write a short Python program in which we want to integrate the mapPlus2 function in our program. Such a program is shown in figure 3.4. In this program, we are importing the Futhark library on line 1 and constructing an instance of the contained Futhark class on line 5.
   On line 4, we generate an array of integers from 0 to 1000000, and finally on line 6, we use the exposed Futhark function mapPlus2 to add 2 to every element in our array.

```
entry mapPlus2 (xs : []i32) : []i32 =
map (+2) xs
```

Figure 3.2: A short Futhark program called mapPlus2.fut

```
$ futhark-py --library -o MapPlus2.py mapPlus2.fut
```

Figure 3.3: We call the Futhark-to-Python compiler `futhark-py` on mapPlus2.fut

```
1  from MapPlus2 import MapPlus2
2
3  def main():
4    xs = range(1,1000000)
5    mapPlus2Class = MapPlus2()
6    xs2 = mapPlus2Class.mapPlus2(xs)
```

Figure 3.4: We use the compiled Futhark program as any other library.

How a Futhark-to-C# would be used

We will now describe a use case for the Futhark-to-C# compiler:

1. We write a short Futhark program, which has a single entry function available. This program takes an array of integers, and adds 2 to each element in the array. The program is shown in figure 3.5.
2. We then compile the Futhark program into a library file, by calling the Futhark compiler from the command line, like shown in figure 3.6. This compiles `mapPlus2.fut` to a C# file called MapPlus2.cs
3. Finally, we write a short C# program in which we want to integrate the mapPlus2 function in our program. Such a program is shown in figure 3.7. In this program, we are importing the Futhark library on line 2 and constructing an instance of the contained Futhark class on line 8.
   On line 9, we generate an array of integers from 0 to 1000000, and finally on line 10, we use the exposed Futhark function mapPlus2 to add 2 to every element in our array.

```
entry mapPlus2 (xs : []i32) : []i32 =
  map (+2) xs
```

Figure 3.5: A short Futhark program called mapPlus2.fut

```
$ futhark-cs --library -o MapPlus2.cs mapPlus2.fut
```

Figure 3.6: We call the Futhark-to-C# compiler `futhark-cs` on mapPlus2.fut

```
1  using System.Linq;
2  using MapPlus2;
3
4  public class Program
5  {
6      public static int Main(string[] args)
7      {
8          var mapplus2Class = new MapPlus2();
9          var xs = Enumerable.Range(0, 1000000).ToArray();
10         var xs_result = mapplus2Class.mapPlus2(xs)
11     }
12 }
```

Figure 3.7: We use the compiled Futhark program as any other library.

But to be able to achieve this, we must design and implement a Futhark C# code generator.

3.2 Obtaining and integrating GPU kernels from- and in high level languages

Our second goal of this thesis is to create an architecture which lets us obtain GPU kernels from high level language code, and afterwards integrate these kernels back into the high level language program. Specifically, we want to obtain GPU kernels from F# source code, and use these kernels in F# programs afterwards.

In figure **??**, we show such an architecture for the F# language.

3.2.1 A use case

1. source code
2. send for translation
3. use function immediately

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

[<FSharkEntry>]
let run_saxpy (a : int) (xs : int array) (ys : int array): int array =
  let res = Map2 (saxpy a) xs ys
  in res
```

Figure 3.8: A short FShark module called MapPlus2.fs

```
[<EntryPoint>]
let main =
  let fshark = new FShark()
  fshark.addSourceFile("MapPlus2.fs")
  fshark.CompileAndLoad()

  let a = 5
  let xs = Iota 10000
  let ys = Replicate 10000 1
  let res = fshark.InvokeFunction("run_saxpy", a, xs, ys)
```

Figure 3.9: Compiling and using MapPlus2.fs from within an F# program.

Chapter 4

The Futhark C# backend

In this chapter we COOL INTRODUCTION mention that the generator is implemented by me, but all the design is inspired by the already existing Python and C backends.

4.1 Recap on using Futhark C# libraries

For a given Futhark program (such as figure 4.1), we want to be able to compile the program to a `C#` library from the command line like shown in figure 4.2. This results in a compiled `C#` dynamically linked library[1]. Then, we can include this library in a `C#` project like any other external library, and use its functions as expected, like shown in figure 4.3.

[1] A `.dll` file

```
entry mapPlus2 (xs : []i32) : []i32 =
  map (+2) xs
```

Figure 4.1: A short Futhark program called mapPlus2.fut

```
$ futhark-cs --library -o MapPlus2.cs mapPlus2.fut
```

Figure 4.2: We call the Futhark-to-C# compiler `futhark-cs` on mapPlus2.fut

```
1  using System.Linq;
2  using MapPlus2;
3
4  public class Program
5  {
6      public static int Main(string[] args)
7      {
8          var mapplus2Class = new MapPlus2();
9          var xs = Enumerable.Range(0, 1000000).ToArray();
10         var xs_result = mapplus2Class.mapPlus2(xs)
11     }
12 }
```

Figure 4.3: We use the compiled Futhark program as any other library.

4.1.1 Compiling and using Futhark C# executables

Not all users are interested in using Futhark programs as parts in other code projects. Instead, these users can opt to compile Futhark programs to executables. Recalling the Futhark example in Figures 4.1 to 4.3, we instead opt to compile the Futhark program as an executable program.

Keeping the Futhark source file `mapPlus2.fut` from 4.1, we use futhark-cs to compile the program into an executable from the command line, shown in figure 4.4. Here, the compiler outputs an executable called `MapPlus2` next to the original source file.

We can now execute this program in one of two ways. Either, we write our arguments in a string on the command line, and `echo` them through a pipe into the executable. This method is shown in figure 4.5. Here, the one argument we are using is an integer array. We pass the array to the executable, and it prints the result to `stdout` after it has finished. The `i32`'s tells us that the integers are 32 bit signed integers. For multi-argument entry functions, we separate the arguments with whitespace.

For the second method, we store our arguments in a dataset file. For example, we can store our integer array in a plain text file, shown in figure 4.6.

We can then use the command line to redirect the contents of the dataset file into our command, as shown in figure 4.7.

If we want to, we can redirect the output from stdout to a file of our own choice, shown in figure 4.8. Here, we redirect Futhark's output to `result.txt`, and print it to `stdout` using `cat` to confirm that we indeed have our correct result.

```
$ futhark-cs -o MapPlus2 mapPlus2.fut
```

Figure 4.4: We call the Futhark-to-C# compiler `futhark-cs` on mapPlus2.fut

```
$ echo "[1,2,3,4,5,6,7]" | ./MapPlus2
[1i32, 2i32, 3i32, 4i32, 5i32, 6i32, 7i32]
```

Figure 4.5: Piping arguments wrapped in a string into our Futhark executable

```
[1,2,3,4,5,6,7]
```

Figure 4.6: The contents of `array.in`

```
$ ./MapPlus2 < array.in
[1i32, 2i32, 3i32, 4i32, 5i32, 6i32, 7i32]
```

Figure 4.7: Redirecting arguments stored in a dataset file into our Futhark executable

```
$ ./MapPlus2 < array.in > result.txt
$ cat result.txt
[1i32, 2i32, 3i32, 4i32, 5i32, 6i32, 7i32]
```

Figure 4.8: Redirecting Futhark output into file, and printing said file to `stdout` afterwards.

4.2 The Futhark C# compiler architecture

In figure **??** we showed a rough sketch of the Futhark compiler's architecture. To sufficiently explore the contribution of this thesis, we will however first need a more detailed view of the architecture we need to implement to accomplish our goal. This architecture is depicted in figure 4.9.

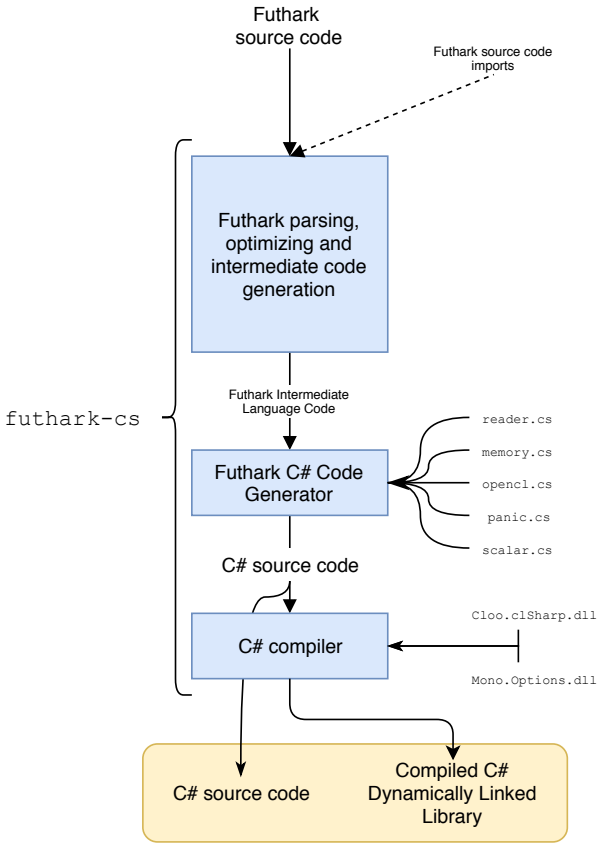


Figure 4.9: The Futhark C# architecture, including necessary imports.

We will now describe its three main steps:

Step 1:

A Futhark source file is passed to the Futhark compiler (like in figure 4.2). Although only the main source file is passed as an argument to the compiler, the compiler also includes any imports in the main source file, if there should be any.

This first part of the Futhark compiler is responsible for parsing the passed Futhark program, including imports, and performs all type checks, SOAC optimizations, fusions and so on**??**. The result of this process is a Futhark program expressed in Futhark's internal intermediate language called `ImpCode`. The `ImpCode` grammar is included in the appendix. The `ImpCode` language contains everything from memory operations like allocating and deallocation memory (both on system memory and on the GPU), interfacing with the OpenCL

device (like copying buffers back and forth between the system and the GPU, setting kernel arguments and launching computation kernels), and also basic expressions like addition and multiplication.

Step 2:

The C# code generator takes the Futhark program written in ImpCode, and expresses it as C# source code. In example, we can take the simple ImpCode expression in figure 4.10, and rewrite it as C# code, shown in figure 4.11.

```
SetScalar "x" (
    BinOpExp Add
      (ValueExp (IntValue (Int32Value 4)))
      (ValueExp (IntValue (Int32Value 5)))
    )
```

Figure 4.10: Setting int x to 4+5 with simplified ImpCode

```
int x = 4 + 5;
```

Figure 4.11: Setting int x to 4+5 in C#

Besides taking an ImpCode program as input, it also embeds a set of prewritten C# libraries[2] into the generated C# code. These libraries are ncessary for the finished C# program, and are described in sec **??**.
The finished C# source code is passed to a C# compiler, but also written to disk so it is available for the developer.

Step 3:

To use the C# code, we need to compile it using the command shown in figure 4.12. We tell the compiler that we have external libraries stored at the location stored at $MONO_PATH[3], and we tell the compiler to reference two extra external libraries Mono.Options.dll and Cloo.clSharp.dll, as we need these libraries in the Futhark C# programs.

We also add the library flag so csc compiles to a .dll file instead of generating an executable. Finally we add the /unsafe flag so the compiler allows us to use unsafe statements in the C# program.

```
$ csc -lib:$MONO_PATH \
      -r:Mono.Options.dll -r:Cloo.clSharp.dll /unsafe mapPlus2.cs
```

Figure 4.12: We call the C# compiler in mapPlus2.cs

However, the last step in futhark-cs does this for the user automatically, as long as the user has the set the required $MONO_PATH variable, and that the directory that $MONO_PATH points to, contains the required libraries.

[2] reader.cs et al

[3] An environment variable that should be set to a directory containing external runtime libraries for Mono runtime usage.

This thesis leverages the software that are used in step 1 and 3, but has not contributed to them. Instead, this thesis implements the code generator described in step 2.

4.2.1 Designing the Futhark C# generator

In the grand scheme of things, the Futhark C# generator is not interesting by itself. The entire contribution to the Futhark compiler is around 4500 SLOC, split 50/50 between C# and Haskell code. We will instead focus on the design of the code *generated* by the code generator.

4.3 The `futhark-cs`-generated C# program

The Futhark C# compiler can compile a Futhark source file in two different ways. As in the examples that have been shown so far, Futhark can compile to a library that is usable in both C# and F# programs. Furthermore, Futhark can compile Futhark programs to standalone executables, which takes argument inputs from the `stdin` stream, and prints the results to `stdout`.

As this thesis focuses on interoperability, we will primarily concentrate on the design of the C# code generated for Futhark libraries, and second mention design differences in cases, where the Futhark executables differs from the libraries. In figure 4.13 we show a high level representation of the generated C# classes generated to their respective specifications.

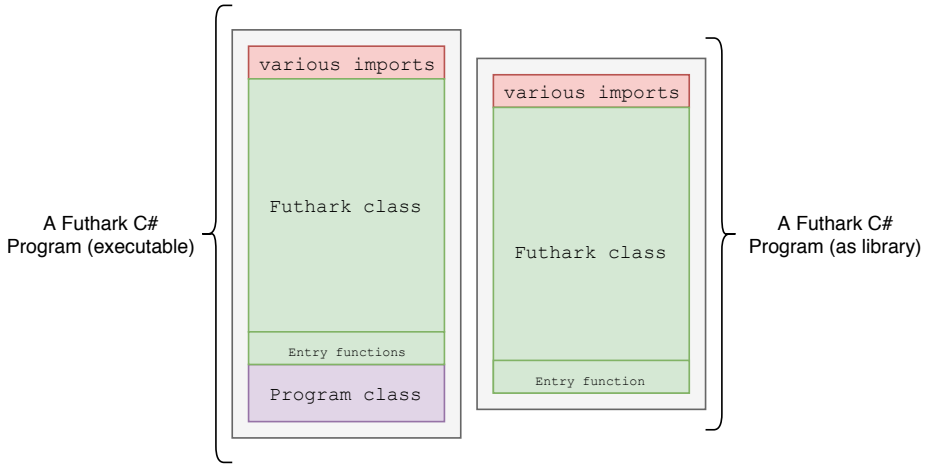


Figure 4.13: The two possible types Futhark C# programs

Here follows a short explanation of the different sections of the Futhark programs.

`various imports`
: This part is the various `using`-statements which imports the necessary library into C# program.

`Futhark class`
: The Futhark class is a singleton class that encapsulates the entire Futhark runtime program. The Futhark class is discussed in subsec 4.3.1.

`Entry functions`
: The entry functions are functions that take human readable input and passes them to the internal representations of Futhark functions.

`Program class`
: In the case of Futhark libraries, the entire C# program consists of the imports and the Futhark class. Only for executables do we add the entry functions to the Futhark class, and the Program class to the C# source file itself.

  The Program class contains a Main method, which is necessary for the C# program to be compiled as an executable. This design is discussed in **??**

4.3.1 The Futhark class design

The Futhark class is the single class defined in the compiled Futhark library. It is depicted in figure 4.14 The following subsections explains the various parts of the

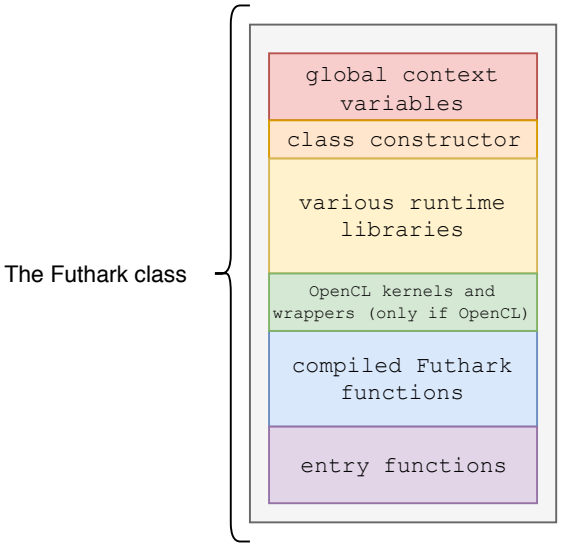


Figure 4.14: An overview of the Futhark class

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 4.15 for an example:

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

Figure 4.15: `writeScalarArray` writes a value at the specified offset in some byte array.

`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:

```
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`, as shown in the example in subsec 4.1.1. 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. This enables Futhark to parse datasets that are stored as pure byte representation, instead of string representations. It is only necessary for Futhark executables.

`opencl.cs`

`opencl.cs` contains wrapper functions for OpenCL's memory related functions. In example, instead of calling `clCreateBuffer` directly for allocating an OpenCL buffer, we call `OpenCLAlloc` from `opencl.cs`. By using a wrapper function instead of calling `clCreateBuffer`, we encapsulate functionality and employ better error handling. The wrapper functions in `opencl.cs` also employs a `free list` for OpenCL memory allocations. This list is stored in the `futhark_context`, and has the following functionality:
1) When the Futhark program frees an OpenCL buffer, it is not actually freed, but is instead added to the `free list`.
2) When the Futhark program later allocates an OpenCL buffer, it first goes through the `free list` to see, whether it can use one of the already existing allocations instead.

The compiled Futhark functions

The compiled Futhark functions are the Futhark functions as written in `ImpCode`[4], expressed in the target language, in this case C#.

[4] See figure 4.10

The compiled Futhark functions 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. There are also mixes, in example `for`-loops that call OpenCL kernels.

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.

4.3.2 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`.

4.3.3 Entry functions in executables

Consider again our small Futhark program mapPlus2 (figure 4.16).

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

Figure 4.16: A short Futhark program called mapPlus2.fut

If we compile this program as an executable, we get Futhark/entry function pair shown in figure 4.17. The example is very simplified but does resembles the actual implementation in functionality.

By calling `entry_main()`, we first call `ReadStrArray<int>` to parse an integers array from stdin. We then read the number of elements in the array into a variable, and then convert the integer array to a byte array, as Futhark functions use byte arrays for internal value array representation.

We then call the internal Futhark function, which returns a byte array, the length of the byte array and the number of elements that the byte array represents. We reform the byte array into an integer array, and print the result to `stdout`.

```
(int, byte[], int) futhark_main(int byte_array_length, byte[]
↪  byte_array, int byte_array_elms)
{

    // ...
    // futhark stuff happens
    // ...

    return (out_array_length, out_array, out_array_elms);
}

void entry_main()
{
    var (int_array, lengths) = ReadStrArray<int>(1, "i32");
    var int_array_elms = lengths[0];
    var byte_array = convertToByteArray<int>(int_array);
    var byte_array_length = byte_array.Length;

    var (res_array_memsize, res_array, res_array_elms) =
        futhark_entry(byte_array_length, byte_array,
        ↪  int_array_elms);

    var res_array_as_ints = reform_array<int>(res_array,
    ↪  res_array_elms);
    printArray(res_array_as_ints);
    exit(0);
```

Figure 4.17: A simplified Futhark/entry function pair from the mapPlus2 executable

4.3.4 Entry functions in libraries

If we compile the program in fig 4.16 program as a library, we get Futhark/entry function pair shown in figure 4.18. The example is very simplified but does resembles the actual implementation in functionality.

The difference between this example and the example in subsec 4.3.3 is that function parameters are given as function arguments, and that the result is returned from the function as opposed to just writing them to `stdout`.

```
(int, byte[], int) futhark_main(int byte_array_length,
↪  byte[] byte_array, int byte_array_elms)
{

    // ...
    // futhark stuff happens
    // ...

    return (out_array_length, out_array, out_array_elms);
}

(int[], int[]) entry_main(int[] int_array, int[]
↪  int_array_lengths)
{
    var int_array_elms = int_array_lengths[0];
    var byte_array = convertToByteArray<int>(int_array);
    var byte_array_length = byte_array.Length;

    var (res_array_memsize, res_array, res_array_elms) =
        futhark_entry(byte_array_length, byte_array,
        ↪  int_array_elms);

    var res_array_as_ints = reform_array<int>(res_array,
    ↪  res_array_elms);
    var res_lengths = new int[] { res_array_as_ints.Length
    ↪  };
    return (res_array_as_ints, res_lengths);
```

Figure 4.18: A simplified Futhark/entry function pair from the mapPlus2 library

Type signature for entry functions

Currently, library functions aren't callable with jagged arrays[5], but must instead be called with a flat element array and an array of lengths denoting the dimensionality of the flat element array. This is explained in sec **??**.

The `FShark` implementation offers helper functions that can flatten jagged arrays into flat arrays/dimension array pairs, and back again. In the future, this might be added to the Futhark `C#` backend for usability purposes.

[5] See sec **??**

The Program class design

As shown in figure 4.13, we only add the Program class to the Futhark program so we have an entrypoint for the executable. The entire Program class is depicted in figure 4.19, with the contained source code and all.

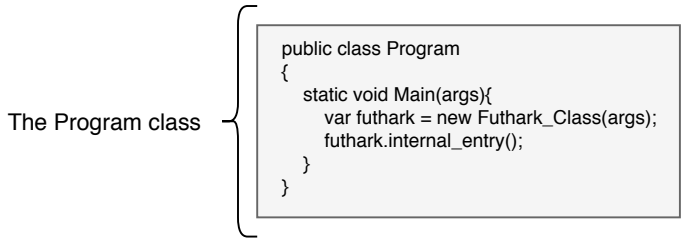


Figure 4.19: The complete functionality of the Program class for Futhark executables.

4.4 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 4.20, 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.

In `C#`, arrays and lists are accessed by indexing, i.e. `var x = myArray[10];`. These arrays are managed by .NET's CLR[6], and access violations, such as indexing out of bounds, makes the CLR raise a suitable exception, which can be handled in the `C#` program by catching it accordingly.

However, for performance reasons4.4.1, we are interested in writing to our `C#` arrays directly. To do this, we must use both `unsafe` blocks and `fixed` blocks.

[6]Common Language Runtime

```
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 4.20: A short snippet from a Futhark C program

The `unsafe` block

In `C#`, we cannot just write arbitrary values to arbitrary locations, as this opens the program for memory access violations by trying to access memory outside of `C#`s memory space. Such violations triggers segmentation faults which halts the entire program, instead of throwing an exception.

Therefore we encapsulate our unsafe pointer-using code in an `unsafe` block.

The `fixed` block

`C#`s CLR manages memory locations for allocated buffers, which means that it also moves these memory allocations around in memory during program execution when necessary. To be able to read and write to buffers referenced by pointers, we must therefore fix these buffers in memory, before we are able to use them directly.

An example of using the unsafe- and the fixed block is shown in figure 4.21. We start the function by starting an `unsafe` block. After we have started the `unsafe` block, we fix the destination buffer in memory and get a pointer to the exact location that we are interested in. Finally, we use a cast to treat the destination pointer as a `double` pointer so we can store a `double` at that location.

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

Figure 4.21: `writeScalarArray` writes a value at the specified offset in some byte array.

4.4.1 Performance

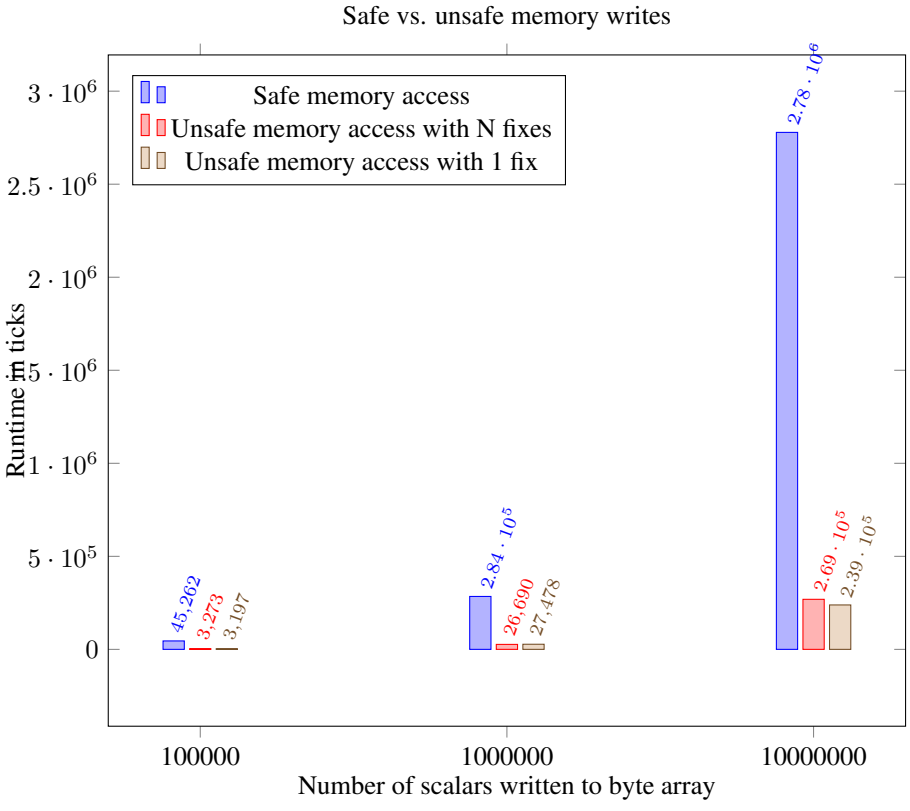
Although C# does offer safe methods to store values directly in byte arrays, we have chosen to avoid these functions as their implementations carry huge overhead compared to doing things the unsafe way. For this benchmark, we are writing N integers to a byte array, using three methods shown in figure 4.22.

```
static void UsingBuffer()
{
    byte[] target = new byte[TEST_SIZE*sizeof(int)];
    for (int i = 0; i < TEST_SIZE; i++)
    {
        var intAsBytes = BitConverter.GetBytes(i);
        Buffer.BlockCopy(intAsBytes, 0, target, i * sizeof(int), sizeof(int));
    }
}

static void UsingUnsafe1()
{
    byte[] target = new byte[TEST_SIZE*sizeof(int)];
    for (int i = 0; i < TEST_SIZE; i++)
    {
        unsafe
        {
            fixed (byte* ptr = &target[i * sizeof(int)])
            {
                *(int*) ptr = i;
            }
        }
    }
}

static void UsingUnsafe2()
{
    byte[] target = new byte[TEST_SIZE*sizeof(int)];
    unsafe
    {
        fixed (byte* ptr = &target[0])
        {
            for (int i = 0; i < TEST_SIZE; i++)
            {
                *(int*) (ptr+i*sizeof(int)) = i;
            }
        }
    }
}
```

Figure 4.22: Three methods of writing integers to an array.

The full program is available in listing 10 in the appendix, to compare safe and unsafe methods of writing values to byte arrays, and the results (as shown in fig 4.23) tells us that there are definite performance gains to retreive by going `unsafe`.

The obvious reason for the performance difference is, that for the safe method, we allocate perform N byte array allocations by using the BitConverter, whereas the unsafe methods perform none. For the difference between the second and third method, we have the small overhead that comes from fixing the target buffer in memory.



H

Figure 4.23: Comparison between Python and Futhark performance for simple model

4.5 Selecting 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 `Cloo`s 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.

4.5.1 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 5

The `FShark` language

The second contribution of this thesis is a high level programming language, which can be compiled automatically to GPU kernels that are readily usable in a mainstream programming language.

In this chapter we present the `FShark` language. The `FShark` language is the sum of two parts:
1) The `FShark` subset, which is a defined subset of the `F#` language and the `F#` standard library.
2) An accompanying standard library, which adds SOACs and array functions that can be used in programs written in the `FShark` subset.

The `FShark` language can be compiled into standalone GPU kernels using the `FShark` compiler **??**. These kernels can then be integrated directly in `F#` programs.

The program in figure 5.1 is written in `FShark`, and can be used as any other `F#` code in an `F#` project. However, as it is written in `FShark`, we can pass it through the `FShark` compiler and end up with the result Futhark code shown in figure 5.2 At this moment we we will skip explaining the meaning of the code, and simply point out that the `FShark` source code and the resulting source code has a strong resemblance.

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

let getArrayPair (a : int) : (int array * int array) =
  let xs = Iota a
  let n = Length xs
  let ys = Rotate (a / n) xs
  in (xs, ys)

[<FSharkEntry>]
let entry (a : int) : int array =
  let (xs, ys) = getArrayPair a
  let res = Map2 (saxpy a) xs ys
  in res
```

Figure 5.1: A short `FShark` program

```
let saxpy (a : i32) (x : i32) (y : i32) : i32 =
  ((((a * x)) + y))
let getArrayPair (a : i32) : ([]i32, []i32) =
  let xs = iota (a)  in
  let n = length (xs)  in
  let ys = rotate (((a / n))) (xs) in
  (xs, ys)
entry entry (a : i32) : []i32 =
  unsafe let patternInput = getArrayPair(a) in
         let ys = patternInput.2 in
         let xs = patternInput.1 in
         let res = map2 ((\(x : i32) -> (\(y : i32) ->
    saxpy(a) (x) (y)))) (xs) (ys) in
         res
```

Figure 5.2: The source code in figure 5.1 compiled to Futhark source code by the `FShark` compiler.

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

5.1 `FShark` syntax

Figures 5.3 to 5.7 shows the complete `FShark` syntax.

$$
\begin{array}{lcl}
prog & ::= & module\ prog \\
 & | & prog'\ prog \\
 & | & \epsilon \\
prog' & ::= & typealias \\
 & | & fun \\
progs' & ::= & prog'\ progs' \\
 & | & \epsilon
\end{array}
$$

$$
\begin{array}{lcl}
typealias & ::= & \texttt{type}\ v\ = t \\
module & ::= & \texttt{module}\ v = prog'\ progs' \quad \textit{(See subsec 5.2.2 on FShark modules)}
\end{array}
$$

$$
\begin{array}{lcll}
fun & ::= & \texttt{[<FSharkEntry>]}\ \texttt{let}\ id\ (v_1 : t_1)\ \ldots\ (v_n : t_n) : t = e & \\
 & | & \texttt{let}\ v\ (v_1 : t_1)\ \ldots\ (v_n : t_n) : t' = e, & \textit{See subsec 5.2.1}
\end{array}
$$

Figure 5.3: `FShark` statements

e	::=	(e)	(Expression in parenthesis)
	|	k	(Constant)
	|	v	(Variable)
	|	$(e_0,\ \ldots,\ e_n)$	(Tuple expression)
	|	$\{\texttt{id}_0 = e_0; \ldots; \texttt{id}_n = e_n\}$	(Record expression)
	|	$[\|e_0; \ldots; e_n\|]$	(Array expression)
	|	$v.[e_0] \ldots .[e_n]$	(Array indexing)
	|	$v.id$	(Record indexing)
	|	$v.id$	(Module indexing (*See subsec 5.2.2*))
	|	$e_1 \odot e_2$	(Binary operator)
	|	$-e$	(Prefix minus)
	|	$\texttt{not}\ e$	(Logical negation)
	|	$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	(Branching)
	|	$\texttt{let}\ p = e_1\ \texttt{in}\ e_2$	(Pattern binding)
	|	$\texttt{fun}\ p_0\ \ldots\ p_n \rightarrow e$	(Anonymous function)
	|	$e_0\ e_1$	(Application)

Figure 5.4: `FShark` expressions

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

Figure 5.5: `FShark` patterns

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

Figure 5.6: `FShark` types

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

Figure 5.7: `FShark` literals

5.2 Notes to the FShark grammar

5.2.1 Limits to function argument types

There are several limits to the F# compiler, which limits the types available in function definitions.

1. Tuples in entry functions, whether they are used in the arguments or in the return types, are allowed to contain a maximum of seven elements. This is because the CLR runtime uses the type `System.Tuple` for these tuples. Incidentally, `System.Tuple` is only defined for tuples up to seven elements.

   This limitation can be circumvented by using more tuples. In example, rewriting a function so it returns
   `((int * int * int * int) * (int * int * int * int))`
   instead of
   `(int * int * int * int * int * int * int * int).`

2. Non-entry functions cannot have tuple type arguments. For functions that take tuple arguments, the F# compiler parses these tuple type arguments correctly. However, the F# compiler rewrites calls to these functions in a curried way. In figures Figure 5.8 and **??** we see two F# functions, and a simplified representation of how they are represented in the F# intermediate language.

   When translating from FShark to Futhark, the Foo function is correctly written in Futhark as a function with a tuple argument, but as the call expression now treats `foo` as having a different type than first defined, the corresponding Futhark translation will trigger a type error at compile time.

```
let foo ((x , y) : (int * int)) : int = x + y

let bar = foo (4,5)
```

Figure 5.8: A function calling another function in F#

```
Function foo ((x , y) : (int * int)) : int = x + y

Function bar () : int = Call foo 4 5
```

Figure 5.9: F# curries tuple arguments when calling tuple functions

3. Entry functions should not use record type arguments, as these have not been investigated fully for FShark use yet.

5.2.2 FShark modules

The modules supported in FShark are not higher-order modules as in ML, but instead just a nested namespace in the containing module.

5.3 F# operators available in FShark

The F# subset chosen for FShark is described in 5.10. Note that all of these operators are overloaded and defined for all integer and floating point types in F#, except for modulus which in Futhark is only defined for integers.

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 the range operator must be used inside an array (as so $[|e_0..e_1|]$), so the expression generates an array instead of a list.

Figure 5.10: FShark operators

5.4 F# standard library functions available in FShark

FShark supports a subset of the F# standard library. These are readily available in all F# programs, without having to open other modules. The standard library subset is shown in figure 5.11.

5.4.1 On selection the F# subset to include in 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 5.10, but also most[1] of the functions shown in the figure 5.11. 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 Futhark's math.fut library. It was therefore aan obvious decision to support these functions and operators in FShark.

[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 5.11: `FShark` operators

However, for the remaining four functions[2] that didn't have equivalents in Futhark's `math.fut`, their function calls are replaced with their identities instead. In example, the `FShark` code in figure 5.12 can be expressed in Futhark directly (figure 5.13). However, the hyperbolic cosine function is available in `F#` (figure 5.14), but not in Futhark.

Therefore, the compiled `FShark` Futhark code is just the hyperbolic function inlined (figure 5.15). When the `FShark` compiler encounters the `cosh` function like figure 5.14, the `cosh` call is replaced with `cosh`'s definition in the `FShark` generated Futhark code.

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

5.4.2 Missing arithmetic operators in `FShark`

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.

[2] `compare`, `cosh`, `sinh`, `tanh`

```
exp x
```

Figure 5.12: e^x in F#

```
exp x
```

Figure 5.13: e^x in Futhark

```
cosh x
```

Figure 5.14: $cosh\ x$ in F#

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

Figure 5.15: $cosh\ x$ in Futhark

5.5 The FShark standard library

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

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 whether they are executed as native F# code, or compiled and executed as Futhark. The guarantee is on the condition that we know that the individual parts of the FShark language and standard library, are correctly translated to Futhark counterparts. We can check that this condition still holds by running the FShark test suite. See sec **??** for an elaboration on these tests.

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

1. Functions like map and array functions like length have direct F# equivalents. The FSharkPrelude versions therefore simply pass their arguments on to the existing functions. In example, FSharkPrelude.Map is shown in figure 5.16. 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:
2. Some Futhark SOACs have F# counterparts that are very close to their original definition. I.e., Futhark's reduce takes a neutral element[3] 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. In such cases, the FSharkPrelude version changes the input slightly before passing it on to the existing function. See figure **??**.
3. Lastly, some functions does not have F# counterparts at all, such as scatter. In these cases, we manually implement an equivalent function in F#. Note that we are not limited to the FShark subset in the FSharkPrelude, as the prelude functions are not translated by the FShark compiler, but detected caught and exchanged for Futhark functions during the FShark compilation (see **??**). FSharkPrelude.Scatter is shown in figure 5.18.

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

Note that calls to FSharkPrelude functions

[3] For parallelization purposes

```
let Map (f : 'a -> 'b) (xs : 'a array) : 'b array =
  Array.map f xs
```

Figure 5.16: `FSharkPrelude.Map`

```
let Reduce (op: 'a -> 'a -> 'a) (neutral : 'a) (xs : 'a array) =
  if null then neutral
  else Array.reduce op xs
```

Figure 5.17: `FSharkPrelude.Reduce`

```
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
```

Figure 5.18: `FSharkPrelude.Scatter`

Why is FSharkPrelude part of the `FShark` language?

Although plenty of functions in the Futhark library already has `F#` counterparts, we have chosen not to allow these `F#` counterparts to be used directly in `FShark` programs. Besides basic differences like different naming in `F#` and Futhark for equivalent functions[4] like, there are multiple other reasons.

1) From a user experience point of view, it is awkward to maintain a whitelist of accepted functions from certain classes. In example, `Array.map` is exchangable with Futhark's `map`, but there are no immediate Futhark version of `F#`s `Array.sortInPlace`. Therefore, the `FShark` compiler would succesfully exchange a call to `Array.map` with a call to `map`, but it would have to halt with an error message, if the user tried to use `Array.sortInPlace`.

2) Some Array functions have subtle differences compared to their Futhark counterparts. As shown in figure 5.17, `Array.reduce` is slightly different in `F#`.

We are slightly hypocritical, as we DO let users use a subset of `F#`s standard library functions. However, there is a whitelist available for this subset in the `FShark` language specification, and the standard library functions are not visibly called as a method from another module.

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 5.19.

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

Figure 5.19: 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 5.20. Similarly, in an OpenCL program, the short code example is translated into a single kernel, as shown in figure 5.21.

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 `map`s into the loop.

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

[4] like `fold` and `foldBack` vs. `foldl` and `foldr`

```
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 5.20: Figure 5.19 compiled as (simplified) non-OpenCL C# code.

```
__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 5.21: Figure 5.19, but compiled as a simplified OpenCL kernel.

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 5.22, the resulting OpenCL program contains a single map kernel with $i * j$ active threads.

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

Figure 5.22: A nested FShark program

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

5.6 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 array types 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.

Functional languages like Haskell mainly works with lists. Although F# is a multi-paradigm language and not exclusively functional, we primarily work with lists when writing functional code in F#.

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, where n is the length of the list. 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 $n \times m$ integers, we therefore have lookups in $O(n + m)$ time.

F# does also have arrays. The `System.Array` class itself is reference type.
If we initialize an integer array in F# like in figure 5.23, the result is that the variable `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.

```
// Array.create : int -> 'a -> 'a array
let arr = Array.create 10 0
// arr = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0;|]
```

Figure 5.23: Initializing an array in F#

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 5.25 we initialize a jagged array of integers. To see how the jagged array is stored in memory, see figure 5.26.

`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 $O(r)$ time[5] as we access arrays in $O(1)$ time, and have to follow r references to get to our element. If we just wanted a reference to the second array

[5] where r is the rank of the jagged array

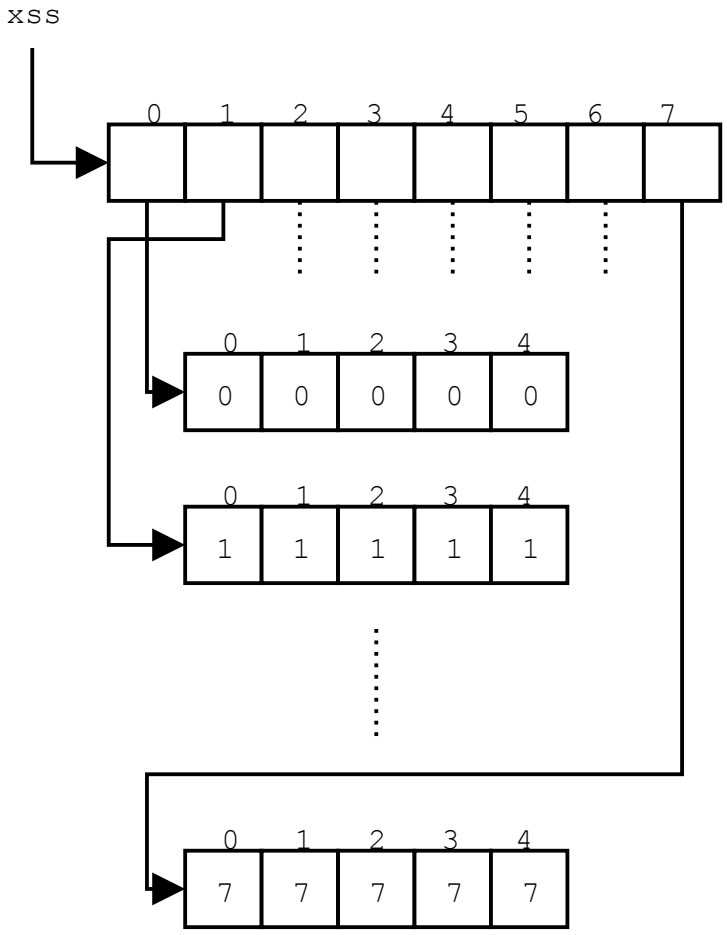


Figure 5.24: The memory representation of an 8 by 5 jagged array in C#.

```
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 5.25: Initializing a jagged array of integers in FSharp

in `xss`, we would be chasing the first reference to `arr`, and then return one of the references stored within.

`F#` also offers actual multidimensional arrays, of up to 32 dimensions. As opposed to jagged arrays, the elements of these multidimensional arrays are stored contiguously in memory, and the entire array can therefore be accessed at once, instead of chasing references like with the jagged array. Chasing references may introduce cache misses which carries cache penalties and therefore a slower performance.

However, using multidimensional arrays in `FShark` would make it much harder to implement `FSharks` SOACs to the standard library. When we apply functions from `F#s Array` module to a jagged array, we treat the jagged array as an array of elements. In example, this means that applying `Array.map f` to a two-dimensional jagged array `xss` will apply `f` to each array referred to by `xss`.

If on the other hand, we used `Array2D.map` to map f over a two-dimensional multidimensional array, we would actually apply f to each element in the multidimensional array, and not each row or column in the multidimensional array.

Implementing SOACs for multidimensional arrays would require a significant effort, as opposed to with jagged arrays, where most SOACs already had equivalent or near-equivalent counterparts in the `F#` library.

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

As mentioned in section 4.3.2, 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.

In figure **??** we see a three dimensional array that is being flattened. The array has $n \times m \times k$ elements.
First, we split the three dimensional array into k two dimensional arrays. The k elements are sorted by their previous k-index.
We then take each of the k two dimensional arrays and split them into $k \times m$ dimensions of n elements each. These $k \times m$ n-elements arrays are sorted by first by their k index (lowest first), and then by their m index.

To reshape the flattened array, just follow the arrows backwards.

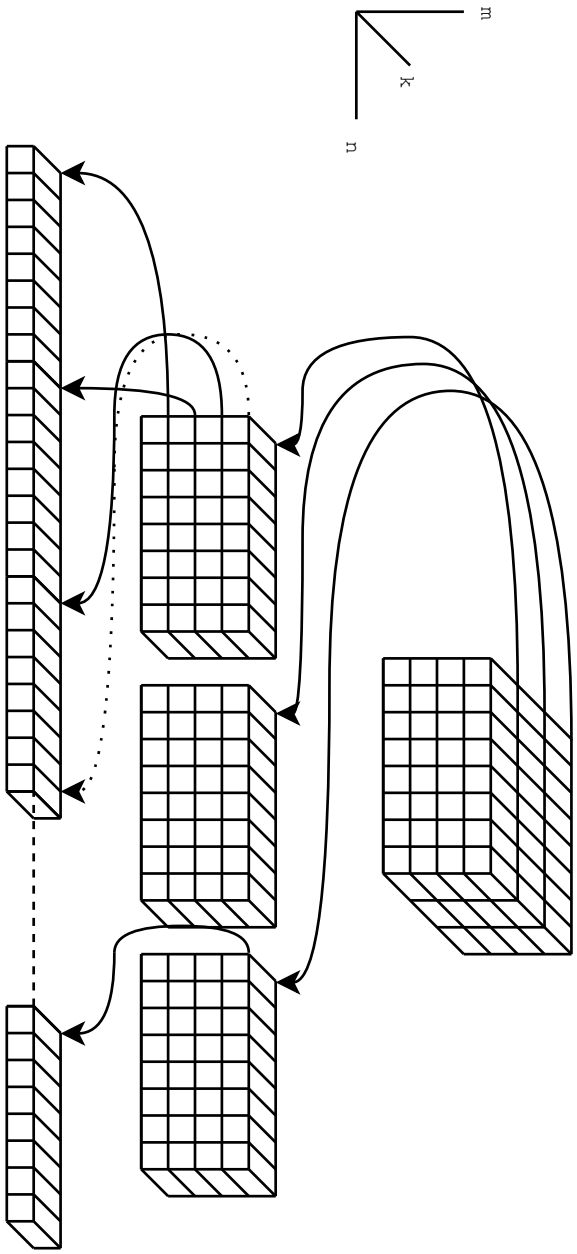


Figure 5.26: The memory representation of an 8 by 5 jagged array in C#.

5.7.1 Analysis of FlattenArray

The simple algorithm for this flattening is described in pseudocode in figure 5.27. 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.

```
1  FlattenArray (array : Array of a) : (Array of b * Array of int) =
2    if a is not (Array of a):
3      return (array, [len(array)])
4    else:
5      subarrays_and_lengths = map FlattenArray array
6      (subarrays, subarrays_lengths) = unzip subarrays_and_lengths
7      subarray_lengths = head(subarrays_lengths)
8      concatenated_subarrays = concat subarrays
9      this_length = len(array)
10     lengths = [this_length] @ subarray_lengths
11     return (contatenated_arrays, lengths)
```

Figure 5.27: Flattening jagged arrays, pseudocode

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 $\prod_{n=1}^{d-1}$(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 $\prod_{n=1}^{d-1}$(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.

5.7.2 Analysis of UnflattenArray

The algorithm `UnflattenArray` in figure 5.28 restores the flat array from the Futhark `C#` program, to a jagged array in `F#`.

```
1 UnflattenArray (lengths : Array of int) (data : Array of a) =
2   if len(lengths) = 1:
3     return data
4   else:
5     length = head(lengths)
6     lengths' = tail(lengths)
7     data' = chunk_array length data
8     data'' = map (UnflattenArray lengths') data'
9     return data''
```

Figure 5.28: Recreating a jagged array from flat array with dimensions

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.

5.7.3 An alternative solution (FSharkArrays)

Instead of using jagged arrays (or even multidimensional arrays), we initially considered implimenting an `FShark` specific array type, which could be directly translated to Futhark's flat array structure.

How they work

How they would alleviate the problem

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

5.7.4 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 6

The `FShark` Compiler and Wrapper

Although `FShark` code can be executed directly in `F#` as normal `F#` code, our benchmarks in chapter **??** shows that compiling our `FShark` code to Futhark OpenCL modules gives us performance increases by several orders of magnitudes (from $\times 100$ to $\times$ 1000).

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 6.1.

```
1  module FSharkExample
2  open FShark.Main
3
4  [<EntryPoint>]
5  let main argv =
6    let wrapper =
7      new FSharkWrapper(
8        libName="ExampleModule",
9        tmpRoot="/home/mikkel/FShark",
10       preludePath=
          ↪  "/home/mikkel/Documents/fshark/FSharkPrelude/bin/Debug/FSharkPrelude.dll",
11       openCL=true,
12       unsafe=true,
13       debug=false
14       )
15   wrapper.AddSourceFile "../../srcs/ExampleModule.fs"
16   wrapper.CompileAndLoad
17   let xs = [|1;2;3;4|]
18   let input = [|xs|] : obj array
19   let xs' = wrapper.InvokeFunction "MapPlusTwo" input :?> int
      ↪  array
20   printfn "Mapping (+2) over %A gives us %A" xs xs'
21   0
```

Figure 6.1: An F# program using `FShark`

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

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

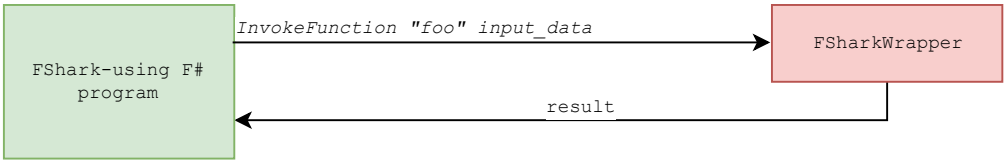


Figure 6.2: The `FShark` compilation pipeline

When calling `CompileAndLoad`, the supplied `FShark` source files are concatenated into one long source file, and written to a temporary location. An FSharpChecker is

[1] See subsection **??**

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`.

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 * FSharpMemberOrFunctio`

: 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 6.3 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.

```
1   module ExampleModule
2   open FSharkPrelude
3
4   module SomeValues =
5     let Four : int = 4
6
7     let SomePlus (x : int) (y : int) : int = x + y
8
9   [<FSharkEntry>]
10  let TimesTwo (x : int) : int =
11    SomeValues.SomePlus x x
12
13  [<FSharkEntry>]
14  let MapPlusTwo (xs : int array) : int array =
15    Map ((+)2) xs
16
17  let PlusSeven (x : int) : int =
18    SomeValues.SomePlus x 7
```

Figure 6.3: A valid 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 6.4, the top module declaration falls away during compilation, so only the top module contents are left.
- 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 6.4 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.)

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.

```
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 6.4: A valid `FShark` program, compiled to Futhark

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 `FSharpExpr`s are union cases of the F# data type `BasicPatterns`.

6.1 Design choices in writing the FShark Compiler

$$[\![Entity(\texttt{IsFSharpRecord}, [(field_0 : \tau_0), \ldots, (field_n : \tau_n)])]\!]$$
$$= \texttt{FSharkRecord}\,\texttt{([}(field_0 : [\![\tau_0]\!]), \ldots, (field_n : [\![\tau_n]\!])\texttt{])}$$

$$[\![Entity(\texttt{IsFSharpTypeAbbreviation}, alias, \tau)]\!]$$
$$= \texttt{FSharkTypeAlias}\,\texttt{(}alias, [\![\tau]\!]\texttt{)}$$

$$[\![Entity(\texttt{IsFSharpModule}, [decl_0, \ldots, decl_n])]\!]$$
$$= [[\![decl_0]\!], \ldots, [\![decl_n]\!]]$$

$$[\![MemberOrFunctionOrValue((name, \tau^*, IsEntryFunction), [(arg_0 : \tau_0), \ldots, (arg_n : \tau_n)], e)]\!]$$
$$= FSharkVal(IsEntryFunction, \texttt{FSharkFunction}([[\![\tau_0]\!], \ldots, [\![\tau_n]\!]], [\![\tau^*]\!]), name, [arg_0, .., arg_n], [\![e]\!])$$

Figure 6.5: Rules for translating FSharp declarations to FShark declarations

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

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

Figure 6.6: Rules for translating .NET types to FSharkIL types

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

Figure 6.7: `FSharkIL` types to Futhark types

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

Figure 6.8: Translation rules for FSharp expressions to FSharkIL expressions

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

Figure 6.9: `FSharkIL` expressions to Futhark

Chapter 7

Evaluation and benchmarks

`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:

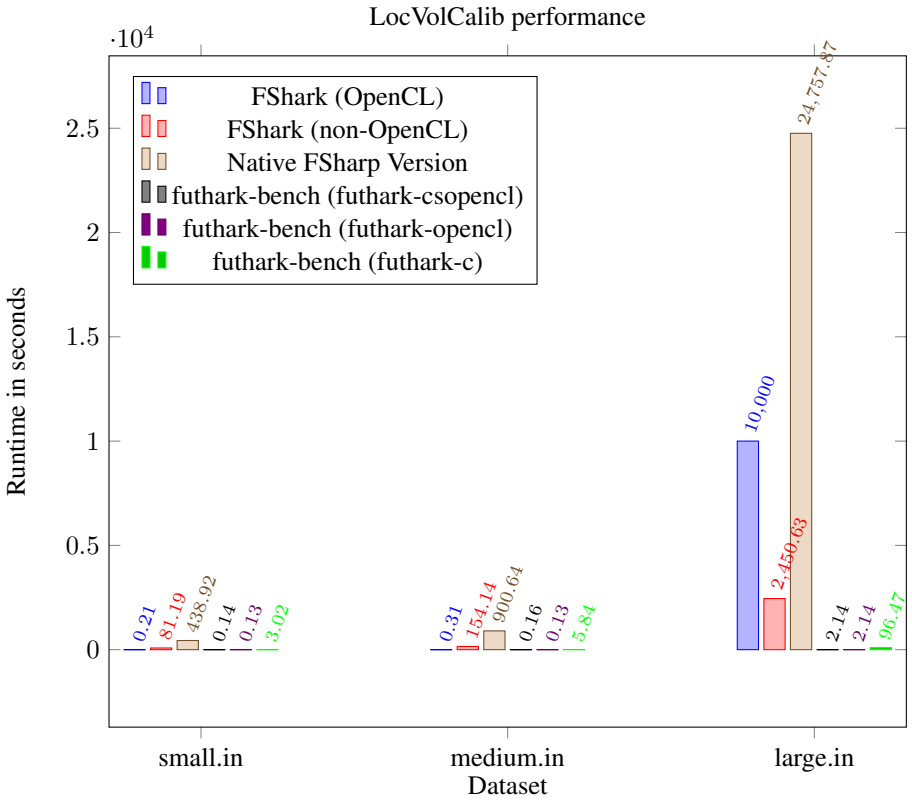


Figure 7.1: Comparison between Python and Futhark performance for simple model

- 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

Testing arithmetic operators

For all arithmetic operators available in `FShark`, I have written an accompanying test, suitably located in the directory `FSharkTests/UnitTests/Operators` in the `FShark` project.

The test suite for the operators are quite rudimentary, and are primarily designed to confirm that the types of the operands are preserved when passed to Futhark `C#` and back again.

The test suite does not test edges (such as integer overflows or dividing decimals by zero) behaves equally in natively executed `FShark` code and translated Futhark `C#` code. These edge cases has been left out for SOME reasons.

(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

Testing `F#` standard library functions

On comparing floating point values

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 target language result is semantically equivalent to the source language code.

In `FSharks` 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 run as `FShark` compiled Futhark code.

Take for example the logarithm function `log`. In both `F#` and Futhark, this function denotes the natural logarithm. If we were translating from a careless language l, where `log` instead denoted the binary logarithm, the translation would still be succesfull and not trigger any type errors. However, the native result and 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.

An `FShark` test is an `FShark` module, but with two extra values added, namely an input and an output value for the test. For example, the test written for the division operator is shown in figure 7.2.

```
module Div
open FSharkPrelude.FSharkPrelude
open FShark.TestTypes.TestTypes
open System

[<FSharkEntry>]
let div (fiveByte : int8) (fiveShort : int16) (five : int)
        (fiveLong : int64) (fiveSingle : single) (fiveDouble : double)
        : (int8 * int16 * int * int64 * single * double) =

    (fiveByte / 2y, fiveShort / 2s, five / 2,
     fiveLong / 2L, fiveSingle / 2.0f, fiveDouble / 2.0)

[<FSharkInput>]
let value = [|5y; 5s; 5; 5L; 5.0f; 5.0|] : obj array

[<FSharkOutput>]
let sameValue =
  (2y, 2s, 2, 2L, 2.5f, 2.5) : (int8 * int16 * int * int64 * single *
  ↪  double)
```

Figure 7.2: The unit test for the `FShark` division operator

Chapter 8

Current limitations

Chapter 9

Related work

Chapter 10

Conclusion and future work

```
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;

namespace ConsoleApplication2
{
    internal class Program
    {
        static private int TEST_SIZE = 1000000;

        static void UsingBuffer()
        {
            byte[] target = new byte[TEST_SIZE*sizeof(int)];
            for (int i = 0; i < TEST_SIZE; i++)
            {
                var intAsBytes = BitConverter.GetBytes(i);
                Buffer.BlockCopy(intAsBytes, 0, target, i *
                ↪  sizeof(int), sizeof(int));
            }
        }

        static void UsingUnsafe1()
        {
            byte[] target = new byte[TEST_SIZE*sizeof(int)];
            for (int i = 0; i < TEST_SIZE; i++)
            {
                unsafe
                {
                    fixed (byte* ptr = &target[i *
                    ↪  sizeof(int)])
                    {
                        *(int*) ptr = i;
                    }
                }
            }
        }

        static void UsingUnsafe2()
        {
            byte[] target = new byte[TEST_SIZE*sizeof(int)];
            unsafe
            {
                fixed (byte* ptr = &target[0])
                {
                    for (int i = 0; i < TEST_SIZE; i++)
                    {
                        *(int*) (ptr+i*sizeof(int)) = i;
                    }
                }
            }
        }

        public static void Main(string[] args)
```

```
52          {
53              var TESTS = 10;
54              var stopwatch = new Stopwatch();
55              for (int i = 0; i < TESTS; i++)
56              {
57                  stopwatch.Start();
58                  UsingBuffer();
59                  stopwatch.Stop();
60              }
61
62              Console.WriteLine("Safe took {0} ticks on avg.",
                ↪  stopwatch.ElapsedTicks / 10);
63
64              stopwatch.Reset();
65
66              for (int i = 0; i < TESTS; i++)
67              {
68                  stopwatch.Start();
69                  UsingUnsafe1();
70                  stopwatch.Stop();
71              }
72
73          Console.WriteLine("Unsafe1 took {0} ticks on avg.",
            ↪  stopwatch.ElapsedTicks / 10);
74
75              stopwatch.Reset();
76
77              for (int i = 0; i < TESTS; i++)
78              {
79              stopwatch.Start();
80              UsingUnsafe2();
81              stopwatch.Stop();
82              }
83
84              Console.WriteLine("Unsafe2 took {0} ticks on avg.",
                ↪  stopwatch.ElapsedTicks / 10);
85
86          }
87      }
88  }
```

Short C# program that measures performance differences between various methods of writing scalars to byte arrays

Chapter 11

FShark 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 `FSharks` 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`.