UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's thesis

Mikkel Storgaard Knudsen

FShark

Futhark programming in FSharp

Advisor: Cosmin Eugen Oancea
Co-advisor: Troels Henriksen

Handed in: August 6, 2018

Abstract

Futhark is a purely functional programming language, designed to be compiled to efficient GPU code. The GPU code has both C and Python interfaces, but does not interoperate with other mainstream languages such as `C#`, `F#` or Java.

We first describe and implement a `C#` code generator that allows GPU kernels written in Futhark to be compiled and used as external libraries in `C#`- and `F#` programs.

We then describe and implement the language `FShark`, which lets developers write and prototype their own GPU code entirely as declarative `F#` code. The language comes with a compiler of its own, which compiles `FShark` programs to Futhark code.

We argue that the `FShark`-to-Futhark translations are correct by comparing the results of `F#`-evaluated `FShark` code with their Futhark-evaluated counterparts, by running a comprehensive suite of 128 unit tests.

We show that Futhark can be compiled to `C#` GPU code, resulting in comparable performance with the `C` OpenCL backends ($0-3\%$ for complex benchmarks), by comparing runtimes across over 20 Futhark benchmark programs ported from other array programming languages.

Finally, we show that complex GPU benchmarks can be easily written in `FShark` in a way that preserves the idiomatic `F#` style. The GPU benchmarks are compiled to GPU code and executed in an `F#` context, where they are shown to perform comparably with benchmarks handwritten in Futhark.

Contents

Preface

This thesis is submitted in fulfillment of a 30 ECTS master's thesis in Computer Science (Datalogi) at the University of Copenhagen, for Mikkel Storgaard Knudsen.

(Front page picture: The humble hedgehog is a peaceful omnivore firmly placed in the lower-middle part of the food chain. However, sneaky disguises occasionally allows it to partake in feasts that are otherwise reserved for apex predators.)

Acknowledgements

First and foremost I would like to thank my supervisor Cosmin Oancea for his help throughout the entire project. It has been absolutely invaluable. Second, I would like to thank Troels Henriksen for putting up with both qualified and less qualified Haskell and Futhark questions, sent to him on IRC, all hours of the day.

Lastly, I would like to thank Christopher Pritchard and Abe Mieres from the official `F#` Slack channel, for their help with type-related `F#` questions.

1.0

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 [8].

Since increasing the frequency of the single CPU has hit the power wall[9] (among other things), achieving higher performance is realized nowadays by scaling the parallelism of hardware, for example leading to multi-core and then to many-core architectures. As the number of cores increases, so does the number of active threads available for parallel data processing.

Modern mainstream (general-purpose) graphical processing units (GPUs) can run tens of thousands of hardware threads in parallel. Modern mainstream CPUs, like the current Ryzen series by AMD, usually support between 10 and 20 simultaneous threads. This makes GPUs an attractive target for data-parallel programming.

While re-writing a program for multi-core execution is not without challenges, GPU programming is significantly more difficult. For example, the typical approach for multicores is to exploit (only) the parallelism of the outermost (parallel) loop – this is because, the loop count is typically greater than the number of cores (maximum 32), which means that the hardware is fully utilized. GPUs, on the other hand, expose a massive amount of hardware parallelism, whose full utilization often requires to exploit multiple (nested) levels of the application's parallelism. Re-structuring the program in this manner is notoriously difficult, not only in terms of tedious work, but also because it requires specialized knowledge of compiler analysis.

To make matters even more unattractive, the mainstream APIs for programming GPUs (e.g., CUDA, OpenCL) are quite low-level and verbose. For example, GPU developers must not only write the computational kernels for the GPUs, but also manually handle the memory (de)allocations and transfers between the host program and the GPU device. All these difficulties in the development of GPU programs, compared to normal (sequential) CPU programming, severely hinders the adoption of GPU programming for the masses.

For example, `C#` [15] and `F#` [14] are two mainstream languages which, to our knowledge, lack high-level (easy-to-use) solutions for GPU programming. It is

safe to say that there exists plenty of large, real-world projects developed in `C#` and `F#`, which could greatly benefit from accelerating their computational kernels on GPUs, but current, low-level solutions (such as CUDA and OpenCL) would demand that those parts are rewritten as GPU code, and linked through foreign interfaces. This process affects code modularity and maintainability, and also requires expert-GPU programmers; often enough this associated cost is deemed too high, so the acceleration benefits are discarded in favor of maintaining a more accessible code base.

1.1. Relation to Related Work

A rich body of scientific work has been dedicated to the design of high-level programming models aimed primarily at GPU acceleration. For example, many domain-specific language (DSL) solutions target specific applications from areas such as image processing [24], data analytics [31], deep-learning [28], mesh computations [22], iterative stencils [30].

Various embedded libraries (DSLs) have been developed to allow high-level expression and transparent-GPU acceleration of (simple) computational kernels written in a host (mainstream) language. For example, Java 8 supports a stream (flat-parallel) construct [19], and Accelerate [4, 7] and Obsidian [29] are embedded in Haskell and support flat parallelism by means of operators such as map, reduce, scan, filter. Other data-parallel and hardware-independent languages, such as Nesl [3, 2], Futhark [18, 17, 10], SAC [11, 12], and Lift [27, 26] support more complex parallel patterns (e.g., nested parallelism), and follow a standalone design.

In essence, all reviewed solutions are primarily aimed at developing (often) complementary techniques for optimizing parallelism in the context of GPU execution, but they do not address well the practical problem of integrating the use of the (now) accelerated computational kernels across different mainstream, productivity-oriented (programming) environments. At best, they provide integration within one language (the host), and at the worst they require the user to hack an interoperability solution (based on a rudimentary foreign-function interface of the host).

This thesis investigates the latter, less-studied problem: we assume that there exists a standalone data-parallel language that can express and execute efficiently computational kernels on GPUs (for example Futhark), and we study two transpilation techniques aimed at aggressively integrating these kernels within the `F#` environment, ideally to the extent that the user is not even aware that Futhark is even part of the picture.

In this we take inspiration from early compiler solutions [23, 6] for interoperating across computer-algebra systems, in which for example the large libraries of Aldor [32]—a strongly and statically typed language supporting dependent types—have been seamlessly made available for use from Maple [21]—a very dynamic language aimed at ease of inspection and scripting. Finally, perhaps the work most related to ours is the experiment [16] that showed that programs written in a usefully-large subset of the old and interpreted (hence slow) APL language [20] can be automatically translated to Futhark, optimized for

GPU execution, and easily integrated in a mainstream application written in Python. Whereas others have translated of F#[25] into efficient C code, this thesis aims at providing a similar solution in which the automatically-translated and mainstream-target languages are F# and C#, respectively.

In summary, the hardware for massively parallel programming is widely available, and many solutions exists for writing efficient GPU programs in high level languages. The problem is that these solutions do not allow integration across different mainstream, productivity-oriented languages.

1.2. What **FShark** sets out to do

This thesis takes inspiration from the APL-Futhark-Python experiment [16], and creates an interoperability solution between F#, Futhark, and C#. The ultimate goal is (1) to allow computational kernels to be written in a usefully-large subset of the F# language, which can be automatically (and efficiently) translated to Futhark, and (2) to also allow the resulting code to be integrated back into a mainstream C# or F# application. The advantages of this approach are that:

- it requires a gentle learning curve for the user, who needs not learn Futhark,
- it allows full access to F#'s productivity tools, e.g., debugger, IDE, which is very useful when developing the computational kernels, and
- it allows the code to be optimized by an aggressive compiler (Futhark), specialized in efficiently mapping nested data-parallelism to GPU hardware.

The solution proposed in this thesis is organized in two steps. The first step is relatively straightforward and it refers to designing a C# code generator (backend) for the Futhark compiler. Starting from a Futhark specification, the code generator must be able to generate C# source files, that can be compiled and used either as standalone executables, or as importable (C#) libraries in any other C# or F# program. There were several notable challenges in this process, namely

- designing a layout that can encapsulate the exports of a Futhark program in a single C# class,
- designing static and efficient helper libraries that help and simplify the code-generation process, and
- designing a way to write sequential (non-GPU) Futhark code as pure C# code, in cases where GPU devices are unavailable.

The second step is more challenging and refers to identifying a useful subset of F# that can be automatically transpiled to Futhark, and, of course, designing and implementing the transpilation scheme. The main challenges here were:

- identifying which parts of the F# language can be suitably translated to Futhark and what the translation should be,

- implementing a standard library written in F#, which contains the implementation of Futhark's data-parallel operators.[1] On the one hand this library publishes/fixes the types of the data-parallel operators, and, on the other hand, it provides an implementation semantically-equivalent to the one of Futhark, for the case when the user is prototyping the application solely within the F# environment.
- designing and implementing a compiler pipeline that would allow efficient and transparent GPU execution of F# programs (via transpilation to Futhark), without the need to manually call Futhark compilers or to import external libraries.

Finally, we demonstrate the feasibility of the approach by means of empirical validation: We show that unit tests written in the identified subset of F# can be compiled and executed correctly as computational kernels on the GPU. More convincingly, we also take several complex benchmark programs written in Futhark, we translate them to the identified subset of F#, and show that the GPU-compiled program, not only runs correctly, but also achieves performances comparable to the one written natively in Futhark. Finally, we show that the exports of the computational kernels originally written in either Futhark or the translatable subset of F#, can be easily integrated back in an application written in C#.

1.3. The contributions of this thesis

The principal 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# language which can be translated directly to Futhark source code of equivalent functionality. This includes a library which implements Futhark SOACs [18] 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.

1.4. Roadmap

- In chapter 2 we describe the current state of programming for GPUs, and explore the differences between low level GPU programming in CUDA, vs. high level GPU programming in Futhark.

[1] For example F# does not support a `reduce` operator of type $(\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha$.

- In chapter 3 we explore the architecture and use cases of both the Futhark `C#` generator and the `FShark` programming language.
- In chapter 4 we present the architecture of the Futhark compiler which our `C#` code generator operates in, and describe the design and implementation of the standalone `C#` computational kernels.
- In chapter 5 we describe the `FShark` language and its standard library, and discuss design choices and consequences regarding array usage in `FShark`.
- In chapter 6 we describe the `FShark` wrapper, which orchestrates `FShark` compilation and GPU code invocation in from within `F#` programs.
- In chapter 7 we describe the `FShark` compiler, which compiles `FShark` source code into Futhark source code.
- In chapter 8 we list the current known limitations of our solution.
- In chapter 9 we evaluate our code generator by testing the performance of its generated CPU code compared to similar GPU code generated by existing code generators. We evaluate the language design of `FShark` by comparing it with other GPU languages. Finally we show the feasibility of the `F#`-to-GPU-code compilation, by comparing the performance of `FShark` code used as `F#` code to `FShark` code compiled to GPU code.

2.0

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.

```
1   #include <stdio.h>
2
3   __global__
4   void saxpy(int n, float a, float *x, float *y)
5   {
6     int i = blockIdx.x*blockDim.x + threadIdx.x;
7     if (i < n){
8       y[i] = a*x[i] + y[i];
9     }
10  }
11
12  int main(void)
13  {
14    int N = 1<<20;
15    float *x, *y, *d_x, *d_y;
16    x = (float*)malloc(N*sizeof(float));
17    y = (float*)malloc(N*sizeof(float));
18
19    cudaMalloc(&d_x, N*sizeof(float));
20    cudaMalloc(&d_y, N*sizeof(float));
21
22    for (int i = 0; i < N; i++) {
23      x[i] = 1.0f;
24      y[i] = 2.0f;
25    }
26
27    cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
28    cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
29
30    // Perform SAXPY on 1M elements
31    saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
32
33    cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
34
35    cudaFree(d_x);
36    cudaFree(d_y);
37    free(x);
38    free(y);
39  }
```

Figure 2.1: $ax + y$ in CUDA

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 several arguments: (1) `a` is a scalar constant denoting a multiplicative constant, (2) `x` and `y` are references (pointers) to floating point arrays located in GPU-device memory, and (3) `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 allocate memory for two arrays `x` and `y` in host (CPU) memory.
Line 19 and 20 allocate 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 host memory to the corresponding arrays `d_x` and `d_y`, which are located in (CUDA) device memory.
Line 31 spawns a number of threads, each executing the code in the `saxpy` function. The amount and structure of the spawned parallelism is defined by the argument denoted by the `«<g, b»>` syntax. More precisely, paramter `b` denotes the number of threads spawned in a block [1], and parameter `g` denotes the number of blocks scheduled in a grid. It follows that the total amount of parallelism is `g×b`, and that the global thread identifier can be computed from the position of a thread inside its corresponding grid (`blockIdx.x`) and block (`threadIdx.x`), by means of the formula `blockIdx.x*blockDim.x + threadIdx.x`. Finally, the grid and block can have as high as three dimensions each (denoted by `.x`, `.y` and `.z`), albeit our example uses a one-dimensional grid and block (for example `.x`).
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.

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] Separating the parallelism into blocks and grids is useful because the threads in a block can be synchronized by means of barriers and can communicate with each other in fast/scratchpad memory. These properties do not hold across threads in different blocks of the grid.

```
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`[2] 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.

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

F#s syntax follows a classical functional programming style. For instance, this means we 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 figure 2.2 re-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 seamless interoperability with the rest of the .NET language family. We can therefore readily use the C# libraries, which exports an object-oriented class structure in F#, and vice versa.

C#

C# is an multi-paradigm programming language developed by Microsoft. In this thesis we implement a Futhark code generator that generates C#-integrable GPU code. For this thesis, C# was chosen for multiple reasons:

- C# is a mainstream programming language, and widely used in commercial and academic settings.
- C# supports imperative C-style programming, which is suitable for writing concrete implementations of programs written in Futhark's intermediate language `ImpCode`, as `ImpCode` itself is an imperative language.

3.0

Birds-Eye View: Architecture and Use Cases

In this chapter, we will take a birds-eye view at the architectures we wish to implement with this thesis. First, we will show how what architecture we need to generate GPU kernels for `C#`, and then show a use case for such kernels.

Second, we show an architecture for making Futhark-generated GPU kernels from `F#` source code and using them in `F#` projects, and then show a use case for this architecture.

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. For 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 a high-level compiler representation. As the compiler applies different passes to optimize the program, it also successively re-writes it using lower-level but still pure-functional intermediate representations (IR). The last step in the blue box is to generate an imperative IR, which is not subject to further Futhark optimizations. This imperative IR is the input to various code generators, for example, the second box in the figure will perform a translation to the C language. The resulting C souce file contains allows the exports of the original Futhark to be used in an intuitive fashion from any other C program.

The Python compilation pipeline follows a similar strategy.

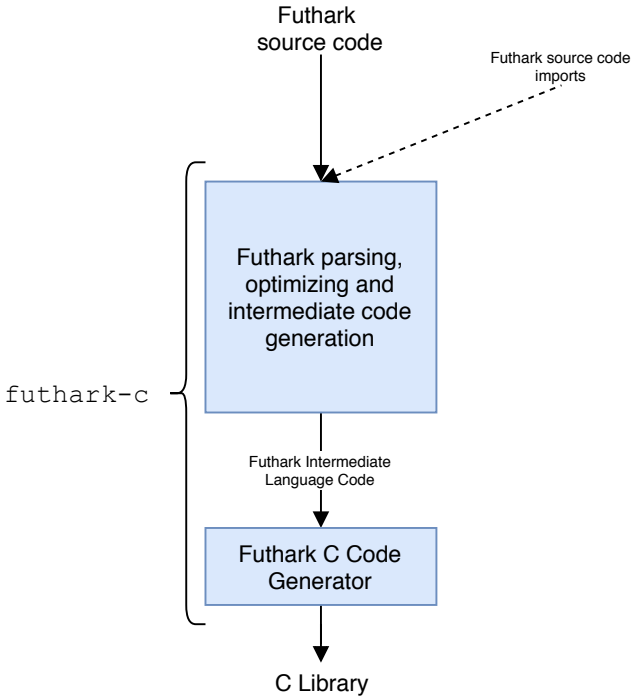


Figure 3.1: The Futhark-to-C compilation pipeline

3.1.1. 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 saved in file `mapPlus2.fut` and is shown below:

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

2. We then compile the Futhark program into a library file, by calling the Futhark compiler from the command line, as shown below:

```
1   $ futhark-py --library -o MapPlus2.py mapPlus2.fut
2
```

   This compiles `mapPlus2.fut` to a Python file called `MapPlus2.py`. In essence, the transpilation process gathers all the Futhark entry-points into a Python class, contained in a Python module, both named `MapPlus2`. For example the Python class can be instantiated with different options, which enable gathering profiling information, or, setting default values for program-parameters such as tile sizes of CUDA block sizes.

3. Finally, we write a short Python program, which uses the exports (entry points) of the original Futhark program, for example, the `mapPlus2` function. One such simple program is shown below.

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

As expected, the Python program first imports the Futhark library on line 1 and then it constructs an instance of the corresponding 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 Futhark function mapPlus2 to add 2 to every element in our array.

3.1.2. Using Futhark in **C#**

The intended way for using Futhark-generated libraries from `C#` follows faithfully the interface already used by Python (and C). For example a use case is described below:

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:

```
1 entry mapPlus2 (xs : []i32) : []i32 =
2   map (+2) xs
3
```

2. We then compile the Futhark program into a library file, by calling the Futhark compiler from the command line, which will compile `mapPlus2.fut` to a `C#` file called MapPlus2.cs

```
1 $ futhark-cs --library -o MapPlus2.cs mapPlus2.fut
2
```

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 3.2. 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.

```
using System.Linq;
using MapPlus2;

public class Program
{
    public static int Main(string[] args)
    {
        var mapplus2Class = new MapPlus2();
        var xs = Enumerable.Range(0, 1000000).ToArray();
        var xs_result = mapplus2Class.mapPlus2(xs)
    }
}
```

Figure 3.2: 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. Transpiling **F#** Computational Kernels to Futhark

The second goal of this thesis is to create an architecture which allows the user to express the computational kernels directly in a subset of a mainstream language. (These kernels can be then integrated back into a mainstream program by means of the code generators discussed in the previous section.) Specifically, we want to obtain GPU kernels directly from `F#` source code, and to use these kernels in a program written in `F#` (or `C#`) afterwards.

In figure 3.3, we show such an architecture for the `F#` language.

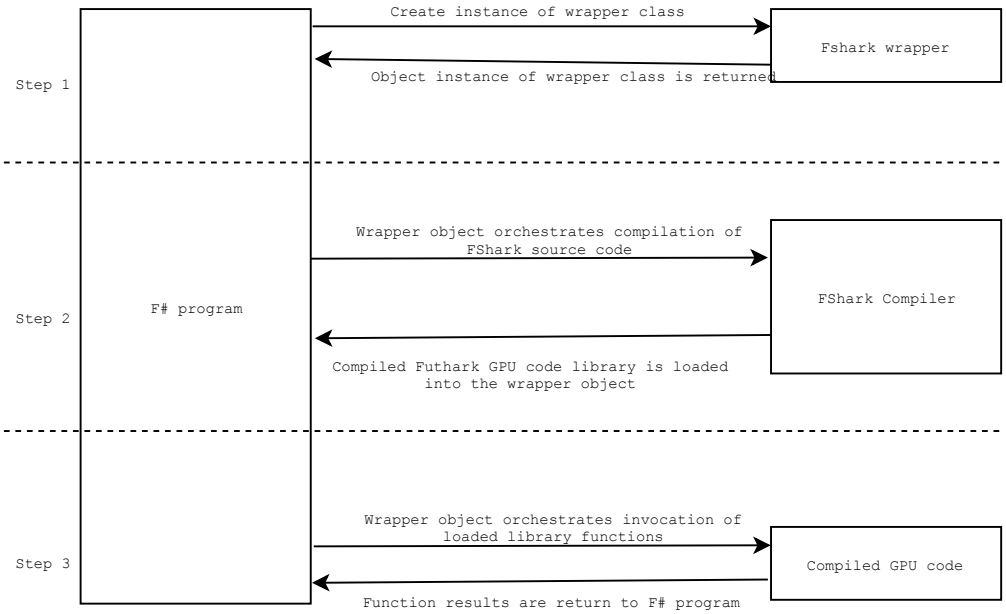


Figure 3.3: The FShark architecture

3.2.1. A use case

Following the architecture shown above, we demonstrate a use case which follows the steps shown in the architecture.

1. We write a small FShark program that we wish to use in our F# program to a file called MapPlus2.fs. Such an FShark program is shown below:

```
1 let saxpy (a : int) (x : int) (y : int) : int =
2   a*x+y
3
4 [<FSharkEntry>]
5 let run_saxpy (a : int) (xs : int array) (ys : int array): int array =
6   let res = Map2 (saxpy a) xs ys
7   in res
```

2. We create an instance of the FShark wrapper class, add the file path of our FShark program to it. Then we run the FShark compilation to compile and load the FShark program into the wrapper. This corresponds to step one and two in the architecture sketch, and is done in line 3-5 in figure 3.4.

3. Finally in line 7-10, we declare some arguments for our FShark function, and pass them to the compiled FShark program through the FShark wrapper.

```
[<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 3.4: Compiling and using MapPlus2.fs from within an F# program.

4.0

The Futhark **C#** backend

In this chapter we first demonstrate how we want to use Futhark-generated C# GPU libraries in C# programs. We then describe the architecture needed for compiling such C# libraries using the Futhark compiler, and what features we need in our code generator to successfully generate standalone C# programs embedded Futhark-generated GPU kernels. We then show the finished design of the generated Futhark C# code, and describe the segments in the code segment by segment.

We discuss choices taken for memory management in Futhark C# programs, and compare different methods of memory management by their runtime performance.

Finally, we discuss how and why the OpenCL library "Cloo" was chosen as a backend for Futhark's C# compiler.

4.1. Recap on using Futhark **C#** libraries

For a given Futhark program (such as the one shown in 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.dll mapPlus2.fut

```

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

```
using System.Linq;
using MapPlus2;

public class Program
{
    public static int Main(string[] args)
    {
        var mapplus2Class = new MapPlus2();
        var xs = Enumerable.Range(0, 1000000).ToArray();
        var xs_result = mapplus2Class.mapPlus2(xs)
    }
}
```

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 into 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; the command line is shown below:

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

```

Here, the compiler produces an executable named `MapPlus2` from the original source file, named `mapPlus2.fut`.

We can now execute this program in one of two ways. The first way is to write our arguments in a string in the command line, and to `echo` them through a pipe into the executable, as shown below:

```
$ echo "[1,2,3,4,5,6,7]" | ./MapPlus2
[3i32, 4i32, 5i32, 6i32, 7i32, 8i32, 9i32]

```

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 result array holds 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, named `array.in`, as shown below:

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

We can then use the command line to redirect the contents of the dataset file into our command:

```
1 $ ./MapPlus2 < array.in
2 [3i32, 4i32, 5i32, 6i32, 7i32, 8i32, 9i32]
3
```

If we want to, we can redirect the output from stdout to a file of our own choice:

```
1 $ ./MapPlus2 < array.in > result.txt
2 $ cat result.txt
3 [3i32, 4i32, 5i32, 6i32, 7i32, 8i32, 9i32]
4
```

Here, we redirect Futhark's output to `result.txt`, and print it to `stdout` by using `cat` to confirm that we have obtained indeed the correct result.

4.2. The Futhark **C#** compiler architecture

In figure 3.1 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 4.4.

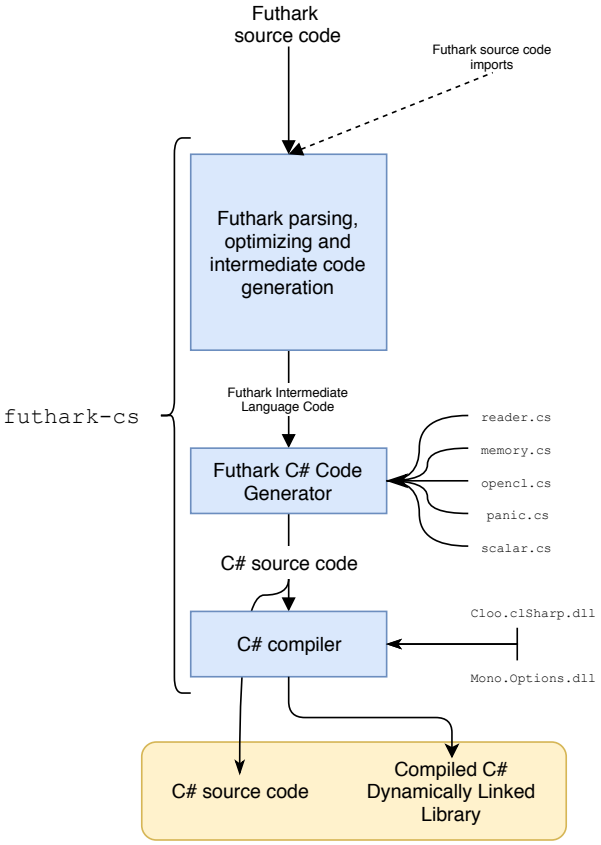


Figure 4.4: The Futhark `C#` architecture, including necessary imports.

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 [18]. The result of this process is a Futhark program expressed in imperative internal intermediate called `ImpCode`. The `ImpCode` grammar is available in the Futhark repository[2]. 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.

[2] `https://github.com/diku-dk/futhark/blob/master/src/Futhark/CodeGen/ImpCode.hs`

Step 2:

The C# code generator takes the Futhark program written in ImpCode, and expresses it as C# source code. For example, we can take the simple ImpCode expression in figure 4.5, and rewrite it as C# code, shown in figure 4.6.

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

Figure 4.5: Setting int x to 4+5 with simplified ImpCode

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

Figure 4.6: Setting int x to 4+5 in C#

Besides taking an ImpCode program as input, it also embeds a set of prewritten C# libraries[3] into the generated C# code. These libraries are ncessary for the finished C# program, and are described in section 4.3.1. The resulted 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 4.7. We tell the compiler that we have external libraries stored at the location stored at $MONO_PATH[4], 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 (-lib) so that 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 4.7: Calling the C# compiler on the resulted mapPlus2.cs file.

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

[3] reader.cs et al

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

This thesis leverages the already existent Futhark codebase to implement steps 1 and 3, hence does not bring important contributions to them. Instead, the contribution of this thesis refers to implementing the code generator described in step 2.

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. In the remaining of this chapter we will instead discuss at a high level the structure of the generated code.

4.3. The Structure of the Generated **`C#`** Code

As previously discussed, Futhark code can be compiled (1) to a library that is usable in both `C#` and `F#` programs, and (2) also to a standalone `C#` executable, which, for example, 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 mention design differences in the cases in which the Futhark executables differs from the translation used for libraries. Figure 4.8 shows a high level representation of the generated `C#` classes for the standalone- and library-compilation cases.

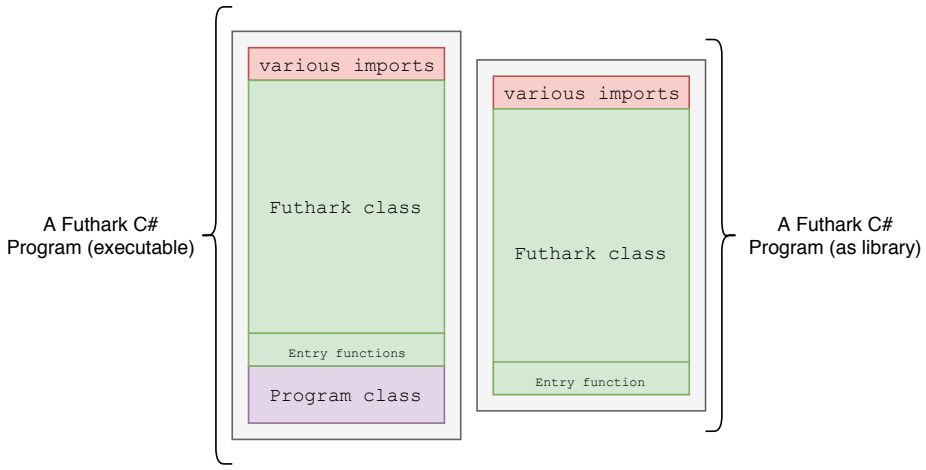


Figure 4.8: The two possible types Futhark `C#` programs

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

`various imports`
: This part consists of `using` statements that import the various libraries on which the translation relies into the `C#` program.

`Futhark class`
: The Futhark class is a singleton class that encapsulates all needed functionality for executing the exports (entry-points) defined in the original

Futhark file. The Futhark class is discussed in subsec 4.3.1.

`Entry functions`
: The entry functions wrappers for the exports declared in the Futhark program, that are mainly responsible for converting the human-readable input into the internal ("machine") representation expected by the export implementation. For example an array can be passed string, but needs to be translated to a one dimensional array of bytes.

`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 4.3.6

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 4.9. The following subsections explains the various parts of the class.

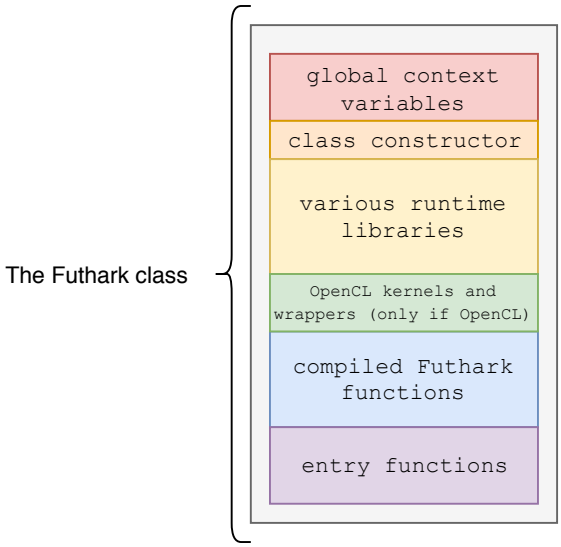


Figure 4.9: An overview of the Futhark class

Global context variables

Compiled Futhark programs need to keep track of several variables. Both normal and OpenCL-enabled Futhark `C#` programs can take several options when they're launched from the command line. For 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 do if we were writing purely functional programs, we instead represent these arguments as class variables, which are set when the class is instantiated. This allows to refer to them from wherever in the rest of the class (without passing them explicitly as arguments).

For non-OpenCL programs, these variables are exclusively used 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 first variable is `struct futhark_context ctx` and contains the global state of the current program's execution. The global state consists of

(1) the current list of unused but allocated OpenCL buffers on the device,

(2) kernel handles for all the OpenCL kernels used in the Futhark program,

(3) a counter for the total running time of the program.

(4) the `opencl_context`, which holds the metadata necessary for OpenCL execution, for example the current state of the device, the queue of in-execution OpenCL actions, and so forth.

The second variable is `struct futhark_context_config cfg` and records configuration information necessary 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`
: Futhark's imperative IR (`ImpCode`) represents all arrays—no matter of their dimensionality and primitive-element type—as a flat one-dimensional array of bytes, which are accompanied by an array of 64-bits integers

recording the dimensions of the flat array. As such it was necessary to define a set of functions that are able to interact with these byte arrays, e.g., if the original array was holding `floats`, than we need to be able to read-/write a float value from/into the byte array. For example, `memory.cs` contains the `writeScalarArray` functions, which writes a scalar value to a certain location into the byte array. The function is overloaded so it works with scalars of any integer or floating point primitive types. Figure 4.10 shows the instance of `writeScalarArray` that writes a value of type `double` into the byte array.

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

Figure 4.10: `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. For example, integer addition in `C#` Futhark is actually implemented by 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. For 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`[5], expressed in the target language, in this case `C#`.

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, for 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,

[5] See figure 4.5

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; for 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 4.11).

```
1 entry main (xs : []i32) : []i32 =
2   map (+2) xs
3
```

Figure 4.11: A short Futhark program called mapPlus2.fut

If we compile this program as an executable, we get Futhark/entry function pair shown in figure 4.12. 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 4.12: A simplified Futhark/entry function pair from the mapPlus2 executable

4.3.4. Entry functions in libraries

If we compile the program in fig 4.11 program as a library, we get Futhark/entry function pair shown in figure 4.13. The example is very simplified but does resembles the actual implementation in functionality.

The difference between this example and the example in section 4.3.3 is that the arguments of the Futhark export are passed directly to the wrapper (e.g., `entry_main`) instead of being parsed from `stdin`. Similarly, the result of the wrapper is the type-casted result of the Futhark export as opposed to the result being written 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 4.13: A simplified Futhark/entry function pair from the mapPlus2 library

4.3.5. On calling Futhark entry functions that takes arrays as arguments

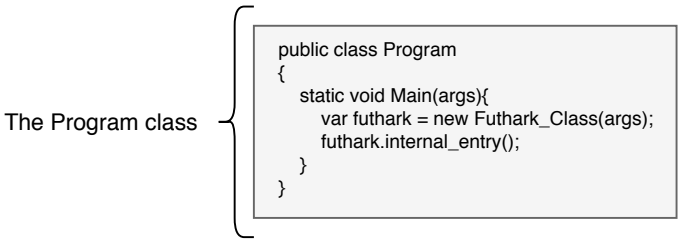
Currently, library functions aren't callable with jagged arrays (arrays of pointers to arrays), but must instead be called with a flat element array, which is paired up with an array of lengths denoting the dimensionality of the flat element array. This is explained in sec 5.8.

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.

4.3.6. The Program class design

As shown in figure 4.8, we only add the Program class to the Futhark program so we have an entrypoint for the executable.

The Program class is a `C#` necessity for compiling executable `C#` programs, as the `C#` standard demands that executable programs must have a `Program` class with a public `Main` method, that it can use as an entrypoint. In Futhark's case, the Main method initializes the internal Futhark class and calls the entry function in the class. The class is shown in the figure below:



4.4. Memory management in Futhark C#

As Futhark stores array values 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 4.14, we see a `for`-loop that performs a prefix-sum operation on an array of integers. On line 6, reading from right to left, we are first creating a 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 dereference 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, for example `var x = myArray[10];`. These arrays are managed by .NET's CLR[13], 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.

```
1  memblock mem_4226;
2  memblock_alloc(&mem_4226, bytes_4224);
3  int32_t scanacc_4187 = 0;
4
5  for (int32_t i_4189 = 0; i_4189 < sizze_4135; i_4189++) {
6      int32_t x_4147 = *(int32_t *) &xs_mem_4223[i_4189 * 4];
7
8      scanacc_4187 += x_4147;
9
10     *(int32_t *) &mem_4226[i_4189 * 4] = scanacc_4187;
11 }
```

Figure 4.14: A short snippet from a Futhark C program

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. Note that this does not necessarily result in unsafe code, because the Futhark generated code is performing the safety checks already.

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 4.15. 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.

```
1  void writeScalarArray(byte[] dest, int offset, double value)
2  {
3      unsafe
4      {
5          fixed (byte* dest_ptr = &dest[offset])
6          {
7              *(double*) dest_ptr = value;
8          }
9      }
10 }
```

Figure 4.15: `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 the three methods shown in figure 4.16.

```
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 4.16: Three methods of writing integers to an array.

The full program is available in listing C in the appendix, to compare safe and unsafe methods of writing values to byte arrays, and the results (as shown in fig 4.17) 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, in each of the N iterations, the BitConverter allocates a small array of bytes, where the value of integer-scalar `i` is recorded, and then it copies this small array into the target byte array. The unsafe methods do not exhibit this overhead, since they update directly the target byte array. The performance difference between the second and third method corresponds to the small overhead that comes from fixing the target buffer in memory.

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 abstractions, for example by wrapping pointer operations in private methods.

`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. Fur-

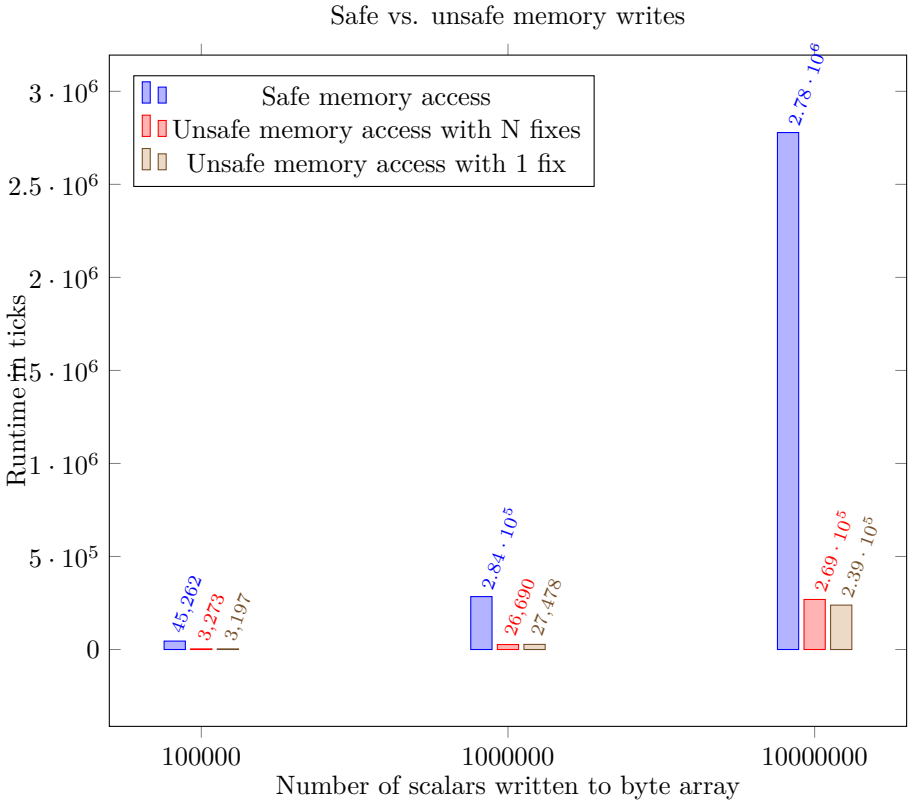


Figure 4.17: Comparison between safe and unsafe methods for writing scalars in a byte array.

thermore, 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.

Furthermore, the `Cloo` project is still alive, and the `Cloo develop` branch on Github is actively being updated as per July 2018[6].

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.

[6] `https://github.com/clSharp/Cloo/commits/develop/Cloo/Source`

5.0

The **FShark** language

test test In this chapter we present the FShark language and it's standard library. We start out by presenting FShark's syntax and discussing the syntax's limitations. Afterwards we present first the built-in F# operators and F# standard library functions available in FShark. We then present our own standard library written specifically for FShark, and discuss why we implemented our own standard library instead of using the standard library already available in F#.

Then, we discuss how array types are implemented in FShark, and how F# arrays differ from Futhark arrays. Finally, we describe and solve the implementation challenges caused by our design choices, and discuss alternative solutions to the one that was chosen.

5.1. What the **FShark** language is

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.

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 7. 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 have 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 the entire FShark language. Although the subset is just a part of F#, we will describe it as if it was a new language.

5.2. **FShark** syntax

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

$$
\begin{array}{lcll}
\mathit{prog} & ::= & \mathit{module}\ \mathit{prog} & \\
 & | & \mathit{prog}'\ \mathit{prog} & \\
 & | & \epsilon & \\
\mathit{prog}' & ::= & \mathit{typealias} & \\
 & | & \mathit{fun} & \\
\mathit{progs}' & ::= & \mathit{prog}'\ \mathit{progs}' & \\
 & | & \epsilon & \\
 & & & \\
\mathit{typealias} & ::= & \texttt{type}\ v\ = t & \\
\mathit{module} & ::= & \texttt{module}\ v = \mathit{prog}'\ \mathit{progs}' & \text{(See subsec 5.3.2 on } \texttt{FShark} \text{ modules)} \\
 & & & \\
\mathit{fun} & ::= & \texttt{[<FSharkEntry>] let}\ \mathit{id}\ (v_1 : t_1)\ \ldots\ (v_n : t_n) : t = e & \\
 & | & \texttt{let}\ v\ (v_1 : t_1)\ \ldots\ (v_n : t_n) : t' = e, & \text{See subsec 5.3.1}
\end{array}
$$

Figure 5.3: `FShark` statements

$$
\begin{array}{lcll}
e & ::= & (e) & \text{(Expression in parenthesis)} \\
 & | & k & \text{(Constant)} \\
 & | & v & \text{(Variable)} \\
 & | & (e_0,\ \ldots,\ e_n) & \text{(Tuple expression)} \\
 & | & \{\texttt{id}_0 = e_0; \ldots; \texttt{id}_n = e_n\} & \text{(Record expression)} \\
 & | & [|e_0; \ldots; e_n|] & \text{(Array expression)} \\
 & | & v.[e_0] \ldots .[e_n] & \text{(Array indexing)} \\
 & | & v.\mathit{id} & \text{(Record indexing)} \\
 & | & v.\mathit{id} & \text{(Module indexing (See subsec 5.3.2))} \\
 & | & e_1 \odot e_2 & \text{(Binary operator)} \\
 & | & -e & \text{(Prefix minus)} \\
 & | & \texttt{not}\ e & \text{(Logical negation)} \\
 & | & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \text{(Branching)} \\
 & | & \texttt{let}\ p = e_1\ \texttt{in}\ e_2 & \text{(Pattern binding)} \\
 & | & \text{fun}\ p_0\ \ldots\ p_n \rightarrow e & \text{(Anonymous function)} \\
 & | & e_0\ e_1 & \text{(Application)}
\end{array}
$$

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

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

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

5.3. Notes to the **FShark** grammar

5.3.1. 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. For 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. Figure Figure 5.8 shows two two F# functions, `foo` and `bar`, and figure Figure 5.9 shows a simplified representation of how these functions are represented in the F#-compiler intermediate representation.

   When translating from `FShark` to Futhark, the Foo function is correctly written in Futhark as a function with a tuple argument, but since the call expression `Call foo 4 5` now treats `foo` as having a different type than first defined, the corresponding Futhark translation will trigger a type error at compile time.

```
1 let foo ((x , y) : (int * int)) : int = x + y
2
3 let bar = foo (4,5)
```

Figure 5.8: A function calling another function in F#

```
1 Function foo ((x , y) : (int * int)) : int = x + y
2
3 Function bar () : int = Call foo 4 5
```

Figure 5.9: F# compiler 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.

5.3.2. **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.

5.4. **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

5.5. **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.

5.5.1. 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

[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` built-ins

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 an obvious decision to support these functions and operators in `FShark`.

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. For example, the `FShark` code shown below:

```
1  exp x
```

can be expressed in Futhark directly as shown below:

```
1  exp x
```

However, the hyperbolic cosine function (shown below) is available in `F#`, but not in Futhark.

```
1  cosh x
```

Therefore, the compiled `FShark` Futhark code is just the hyperbolic function inlined. When the `FShark` compiler encounters the `cosh` function the `cosh`

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

call is replaced with cosh's definition in the FShark generated Futhark code. This is shown below:

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

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.

5.5.2. **FShark** function meaning has precedence to Futhark similars

In FShark, the sign function has type $t \rightarrow int$. However, the similar function in Futhark, sgn, has type $t \rightarrow t$. FShark is designed to be translated into equivalent Futhark functions, which is why the FShark compiler does not translate sign directly to sgn, but does instead inline a type converter $t \rightarrow int$ to wrap the sgn call. See the example below:

```
sign 4.0f
```

is represented in the FShark compiled Futhark source code as

```
i32.f32 (f32.sgn (4.0f))
```

5.5.3. 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.

5.6. 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 9.5 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. 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. For example, FSharkPrelude.Map, -Map3 and -Map4 are shown below:

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

let Map3 f aa bb cc =
    let curry f (a,b,c) = f a b c
    let xs = Zip3 aa bb cc
    in Array.map (curry f) xs

let Map4 f aa bb cc dd =
    let curry f (a,b,c,d) = f a b c d
    let xs = Zip4 aa bb cc dd
    in Array.map (curry f) xs
```

2. Some Futhark SOACs have F# counterparts that are very close to their original definition. for example, 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 example below:

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

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 function calls are not translated by the FShark compiler, but detected and exchanged for Futhark function calls of the same name during the FShark compilation. FSharkPrelude.Scatter[4] is shown

[3]For parallelization purposes

[4]The SOAC Scatter actually has limitations regarding FShark use. See section 8.2.1

below:

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

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

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[5] 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. For 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 `FSharkPrelude.Reduce` example, `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 5.12.

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

Figure 5.12: 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 5.13. Similarly,

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

in an OpenCL program, the short code example is translated into a single kernel, as shown in figure 5.14.

```
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 5.13: Figure 5.12 compiled as (simplified) non-OpenCL `C#` code.

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

This is a concrete implementation Futhark fusion rules as defined in [18]; which states that $(map\ f) \circ (map\ g) \equiv map(f \circ g)$

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

Futhark and nested maps

Futhark's compiler also specializes in parallelizing nested SOAC calls[18], which for example transforms nested `map` expressions into one single `map` expression. For Futhark programs like the one in figure 5.15, the resulting OpenCL program contains a single map kernel with $i * j$ active threads.

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

5.7. Arrays in **F#** versus in Futhark

As Futhark is an array language, designing the array handling for `FShark` was an important part of the design process. Whereas multidimensional array types in Futhark are written as, for example, `[][]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 (for example `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#`.

```
__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 5.14: Figure 5.12, but compiled as a simplified OpenCL kernel.

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

Figure 5.15: A nested `FShark` program

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 5.16, 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 5.16: 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 5.18 we initialize a jagged array of integers. To see how the jagged array is stored in memory, see figure 5.17.

`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.

Denoting by r the rank of the jagged array, the lookup requires r queries to memory, because accessing an array requires a memory acces, and we have to follow r references to get to our element. If we just wanted a reference to the second array in `xss`, we would be chasing the first reference to `arr`, and then return one of the references stored within.

`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 in 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.
For 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.

5.8. 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 5.19 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.

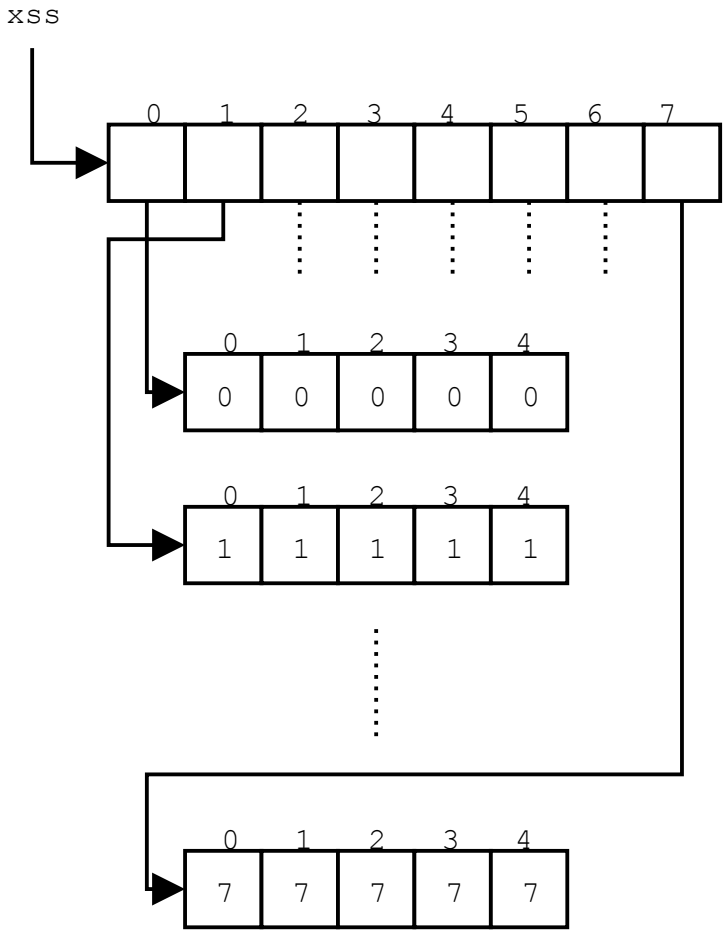


Figure 5.17: The memory representation of an 8 by 5 jagged array in C#.

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

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.

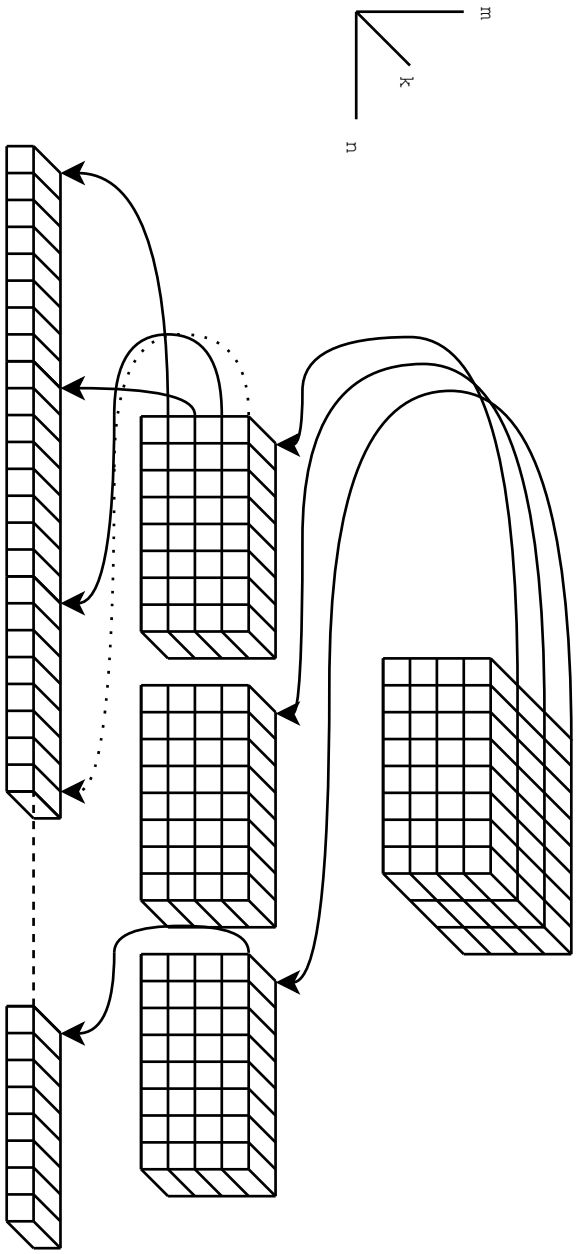


Figure 5.19: Flattening a three dimensional jagged array into one flat contiguous array.

5.8.1. Analysis of FlattenArray

The simple algorithm for this flattening is described in pseudocode in figure 5.20. The implemented algorithm is slightly more complex, as it has perform various type castings, and also checks for invalid arrays such as irregular arrays. In the example, the type $'b$ denotes a primitive type such as booleans or integers.

Type a on the other hand, is any type variable - namely array types.

```
1  FlattenArray (array : Array of a) : (Array of 'b * Array of int) =
2    match typeOf(array) with
3    | Array of 'b ->
4      return (array, [len(array)])
5    | Array of _ ->
6      subarrays_and_lengths = map FlattenArray array
7      (subarrays, subarrays_lengths) = unzip subarrays_and_lengths
8      subarray_lengths = head(subarrays_lengths)
9      concatenated_subarrays = concat subarrays
10     this_length = len(array)
11     lengths = [this_length] @ subarray_lengths
12     return (contatenated_arrays, lengths)
```

Figure 5.20: 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: As the array in the function call was an array of arrays, we call the flattening algorithm recursively on the subarrays.

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.

Notes on the implementation of the FlattenArray algorithm

We have designed and implemented the original FlattenArray algorithm ourselves. However, the first implementation used a variant of the FSharkArray datatype (discussed in section 5.8.4). That algorithm is shown below:

```
let rec FSharkArrayToFlatArray (arr : FSharkArray<'a>) : ('a [] * int64 []) =
   let checkRegularity arrs : unit =
      if Array.length arrs = 0
      then failwith "Empty array"

      let head = Array.head arrs
      if not <| Array.forall (fun l -> l = head) arrs
      then failwith "Irregular array"

   match arr with
   | Bottom list -> (list, [|int64 <| Array.length list|])
   | Dimension(subarrays) ->
      let a =  List.toArray <| List.map FSharkArrayToFlatArray subarrays
      let (subarrs, lens) = Array.unzip a
      checkRegularity lens
      let subarrs' = Array.concat subarrs
      let lens' = Array.head lens
      let len_subarrs = int64 <| List.length subarrays
      let lens_out = Array.append [|len_subarrs|] lens'
      (subarrs', lens_out)
```

However, as we changed the `FShark` implementation to use jagged arrays instead, we also had to change the implementation of the FlattenArray algorithm. To make the FlattenArray work with jagged arrays, we posted a question on the official `F#` slack channel for advice on removing the type checks in the function, together with the at the time current implementation of the algorithm.

This led the users Christopher Pritchard and Abe Mieres to post a rewritten version of the algorithm, which is the version that is used in the current implementation of the `FShark` compiler.

```
let rec ArrayToFlatArray (array : System.Array) =
    if array.Length = 0 then failwith "Empty array"
    let array0 = array  |> Seq.cast |> Seq.toArray
    let arrays = array0 |> Array.choose (box >> function
                                         | :? System.Array as xs -> Some xs
                                         | _                     -> None
                                        )
    let lengths = Array.map (Seq.cast >> Seq.length) arrays |> Array.distinct
    if   arrays .Length  = 0            then array0 |> Array.map unbox, [|int64 array.Length|]
    elif arrays .Length <> array.Length then failwith "Invalid array"
    elif lengths.Length  > 1            then failwith "Irregular array"
    else
        let a             = Array.map ArrayToFlatArray arrays
        let subarrs, lens = Array.unzip a
        let subarrs'      = Array.concat subarrs
        let lens'         = Array.head lens
        let len_subarrs   = int64 array.Length
        let lens_out      = Array.append [|len_subarrs|] lens'
        subarrs', lens_out
```

5.8.2. Analysis of UnflattenArray

The algorithm `UnflattenArray` in figure 5.21 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    match len(lengths) with
3    | 1 ->
4      return data
5    | _ ->
6      length = head(lengths)
7      lengths' = tail(lengths)
8      data' = chunk_array length data
9      data'' = map (UnflattenArray lengths') data'
10     return data''
```

Figure 5.21: 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 $O(n * d)$.

5.8.3. 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.

5.8.4. 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.

This data type is shown in figure 5.22. An `FSharkArray<'a>` contains a flat array of `<'a>`, and a list of integers denoting the lengths of the arrays contained

in the flat array.

```
type FSharkArray<'a> = class
  val mutable flatArray : 'a array
  val mutable dimensions : int array
  end
```

Figure 5.22: The basic structure of an FSharkArray

This would allow us to skip the flattening and unflattening algorithms that are currently used for invoking imported Futhark functions, and instead just pass the contents of the arrays as is.

However, this approach was deemed impractical for several reasons. Jagged arrays readily support getting subarrays and elements using the array indexing operator intuitively. for example, for a two-dimensional array `xss : int [][]`, we can expect that `xss.[1]` returns a subarray, and that `xss.[1].[4]` returns an integer. For example, to get the same functionality for `FSharkArrays`, we would have to implement the array operator for `FSharkArrays` manually. The array operator would have to access the flat array by calculating an offset using the array operator operands together with the lengths stored in the `dimensions` integer array.

Besides calculating array indexes manually, we would also have to handle that the index operator must be able to return either an element of some type `'a`, or another `FSharkArray`. This could be handled by implementing `FSharkArray` as a discriminated union type instead; as shown in figure 5.23.

```
type FSharkArray 'a = FlatArray of ('a array * int array)
                    | Element of 'a
```

Figure 5.23: FSharkArray as a discriminated union type

This way, an `FSharkArray` can be either an array or an element. However, we then have a third problem. Wherever we are using `FSharkArray 'a`, our elements from the arrays will be wrapped as `Element of 'a`s.

This means that we will have to either implement a custom set of `F#` operators and standard library functions which unwraps `Elements` before passing them on to the actual operator or function, or at least implement an unwrapper function of type $unwrap : Element\ of\ 'a \rightarrow 'a$, which must be applied everywhere in functions that uses both `FSharkArrays` and `F#` standard library functions.

5.8.5. 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 multi-dimensional 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`.

6.0

The **FShark** Wrapper

In this chapter we will first demonstrate how to compile and use an `FShark` module within an `F#` project. We will then delve further into how and why compiled `FShark` functions are actually invoked through through a wrapper class called the `FShark Wrapper`.

Finally, we will discuss the performance disadvantages that comes with the current implementation of the `FShark` wrapper.

6.1. Using the **FShark** Wrapper

Although `FShark` code can be executed directly in `F#` as normal `F#` code, our benchmarks in section 9.6.3 shows that compiling our `FShark` code to Futhark GPU modules gives us performance increases by several orders of magnitudes (from $\times 100$ to $\times 1000$).

We therefore need to implement a wrapper which enables us to compile our `FShark` programs as well as utilize them in our `F#` programs.

6.1.1. Another short **FShark** module

Below we see a simple `FShark` module that we want to compile into a GPU kernel and use in our `F#` program.

```
1  module ExampleModule
2  open FSharkPrelude
3
4  let saxpy (a : int) (x : int) (y : int) : int =
5    a*x+y
6
7  let getArrayPair (a : int) : (int array * int array) =
8    let xs = Iota a
9    let n = Length xs
10   let ys = Rotate (n / 2) xs
11   in (xs, ys)
12
13 [<FSharkEntry>]
14 let entry (a : int) : int array =
15   let (xs, ys) = getArrayPair a
```

```
16    let res = Map2 (saxpy a) xs ys
17    in res
```

Line by line, this module does the following:
L1: We define the name of this module as `ExampleModule`. If we want to use this module in an `F#` program without compiling it as `FShark` first, we can refer to this module by this name.
L2: We open `FSharks` standard library `FSharkPrelude` in this module, so we can access the standard functions in the `FShark` module. In this module, we are using the standard functions `Iota`, `Length`, `Rotate` and `Map2`.
L4-5: We define the function `saxpy`.
L7-11: We define the function `getArrayPair`.
L8: `Iota a` returns the integer array of the numbers from 0 up to, but not including, a.
L9: `Length xs` returns the length of the array `xs`.
L10: `Rotate n` rotates the contents of an array n places in either the right or left direction.
For example, `Rotate 2 [1;2;3;4;5;6] = [5;6;1;2;3;4]`,
and `Rotate (-2) [1;2;3;4;5;6] = [3;4;5;6;1;2]`
L11: Here we return the pair of arrays `(xs, ys)`.

L13-17: We define the entry function `entry`.
L15: We call `getArrayPair` to get two arrays.
L16: We use `Map2` to map the curried function `(saxpy a)` over the arrays `xs` and `ys`.

For two arrays xs = $[x_1, x_2, \ldots, x_n]$ and ys = $[y_1, y_2, \ldots, y_n]$,
Map2 (saxpy a) xs ys = $[\text{saxpy a } x_1\ y_1, \text{saxpy a } x_2\ y_2, \ldots, \text{saxpy a } x_n\ y_n]$.

L17: The entry function returns the result of the call to `Map2`.

This concludes the short `FShark` module.

6.1.2. Compiling and using the short **`FShark`** module

With our `FShark` module ready, we now proceed to compile, load and use it. This is shown 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 "ExampleModule.fs"
16   wrapper.CompileAndLoad
17   let a = 1000000
18   let result = wrapper.InvokeFunction("entry", a) :?> int
      ↪  array
19   printfn "Mapping (+2) over %A gives us %A" xs xs'
20   0
```

Figure 6.1: An F# program using `FShark`

Let us now explain line by line what is happening in the figure:
L6: 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 9.6.3 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.

L15: Now we can pass a source file to the `FShark` wrapper.
L16: We tell the wrapper to compile the source file that we have added to the wrapper object, and load the compiled library into the wrapper afterwards.
L17: As our entry function defined in the program in figure 6.1 takes an integer as argument, we define an integer variable we can pass to it.
L18: We use the wrapper to invoke the entry function from the compiled and loaded library, using our previously declared `a` as the only argument. As the `FShark` wrapper uses reflection to dynamically load compiled libraries at runtime, we are not able to statically determine what type of result we will get from the `InvokeFunction` call. Therefore, we use F#s downcast operator `(:?>)` to declare the return value as an `int array`.

If we are in doubt of which type to downcast to, we can always lookup the return type by reviewing the `FShark` module's source code. We can downcast to any of the types usable in `F#`, including tuples and arrays.

6.2. On the design decisions of the **FShark** wrapper

To summarize, the current design of `FShark` usage is dependent on a wrapper object, which for all `FShark` projects must compile and load the input `FShark` modules once, and, afterwards, it must pass arguments from `F#` to the resulting GPU kernels by using .NET reflection. This design has several costs for both usability and performance, and we will here discuss some of these costs, and what we could do to alleviate them in the future.

6.2.1. Compiling and loading **FShark** modules at every startup

At this time, `FShark` works by compiling and loading `FShark` modules just in time before they are needed in the containing `F#` project. However, this is more often than not redundant work. For the developer who is using `FShark` to develop prototypes of `FShark` GPU kernels, it is of course beneficial to continuously recompile the `FShark` program under development to verify that changes are being made.

However, when the `FShark` program is finished and ready to be used in projects, it isn't necessary to compile it again.

How much time do we spend on compiling and loading the **FShark** modules?

If we, instead of loading the compiled `FShark` GPU kernels through the `FShark` wrapper, just open the compiled kernel libraries as any other `C# .dll` file, we

can circumvent the FShark compiler completely, and use the compiled kernel directly.

For the benchmark LocVolCalib 9.6.3), we have measured the average time cost of the program compilation time. In figure 6.2 we see how the time is spent during the compilation.

Parsing FShark code using F# parser	217984 ms
Converting F# declarations to FSharkIL	+ 19129 ms
Converting FSharkIL to Futhark source code	+ 98949 ms
Compiling Futhark to C# with futhark-cs	+ 8037165 ms
Compiling C# code using C# compiler	+ 999251 ms
Loading compiled C# class using reflection	+ 101601 ms

Figure 6.2: Time spent on compiling LocVolCalib at runtime in FShark-using program

So the dynamic compilation process takes about 10 seconds, while running the accelerated program on the largest dataset takes only a couple of seconds. This is a costly affair, which we could definitely live without.

Suggestions for changes

We have two main suggestions for change.
1) The easiest way would be to add a AddModulePath function to the FShark wrapper. The benchmarks show that we aren't spending that much more time when loading compiled assemblies into F# using reflection, than if we opened the assembly as a library in the project.

Therefore, we could add a function that takes a path to a compiled FShark module, and loaded the path's target into the wrapper.

2) We could also go for a second, more permanent solution. Instead of relying on F#s reflection functionality to load our compiled assemblies into scope dynamically, we could redesign the FShark use case itself, so that it uses just the compiler, and not the wrapper.

In this case, the new use case would be to compile the FShark modules using the FShark compiler, and then manually reference- and open them in F# projects. This would not only remove the repeated module compilation, but also let us use static typing with the compiled FShark modules, enabling autocompletion and type checking for function arguments, and also removing the need to manually downcast the FShark invokation results.

6.2.2. The overhead of invoking GPU kernels

The second issue with the current approach is, that every single call to a FShark function carries significant overhead for copying data back and forth between CPU and GPU buffers. This is a problem when we are chaining together GPU function calls: that is when we take the array output of function f and use it as an argument for function g without any changes to it.

In figure 6.3 we see a chain of three calls to a compiled `FShark` module. Although we are calling the second function with the result from the first function together with another array, and calling the third function with result from the second function, we are still moving the results from the GPU buffers to our system RAM between each call, and deallocating the buffers on the GPU, even though we are going to reallocate them soon thereafter.

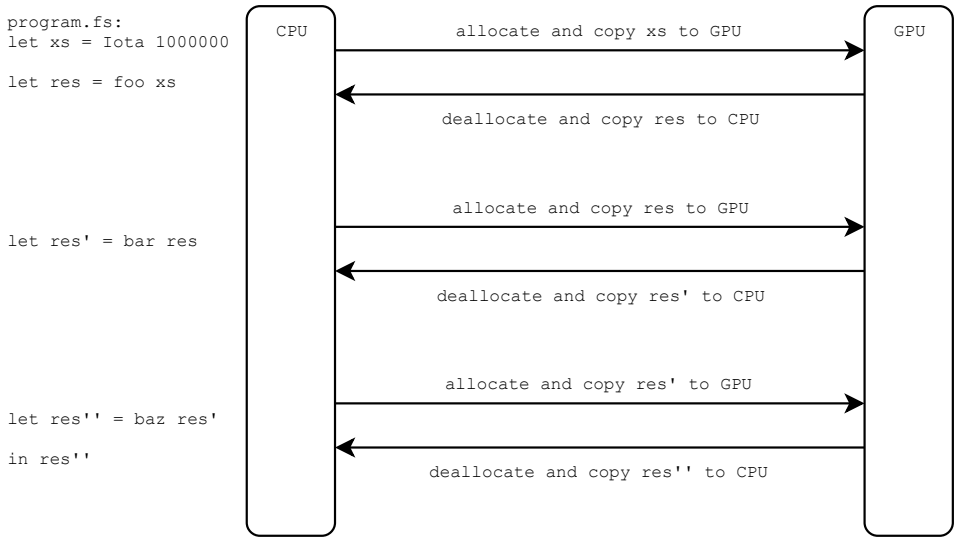


Figure 6.3: Buffers are copied back and forth between CPU and GPU between calls

In the future, we could eliminate this overhead by allowing the compiled Futhark functions to opaque arrays instead of actual data arrays. Opaque arrays are merely references to already allocated GPU buffers, and can be resolved into actual data arrays only when the data is needed in the remaining program.

We could then also have multiple versions of the compiled Futhark functions; one version that takes an actual data array as input, and one that can use a reference to a GPU buffer instead.

In this case, we could wait until after the three function calls to actually copy the referenced GPU buffer back to the system RAM. This would strongly reduce the number of copies back and forth between the GPU and the system RAM: Instead of the allocations/deallocations increasing linearly with the number of chained GPU kernel calls, we can make do with one allocation and one deallocation between system RAM and GPU, pr. chain, as in figure 6.4

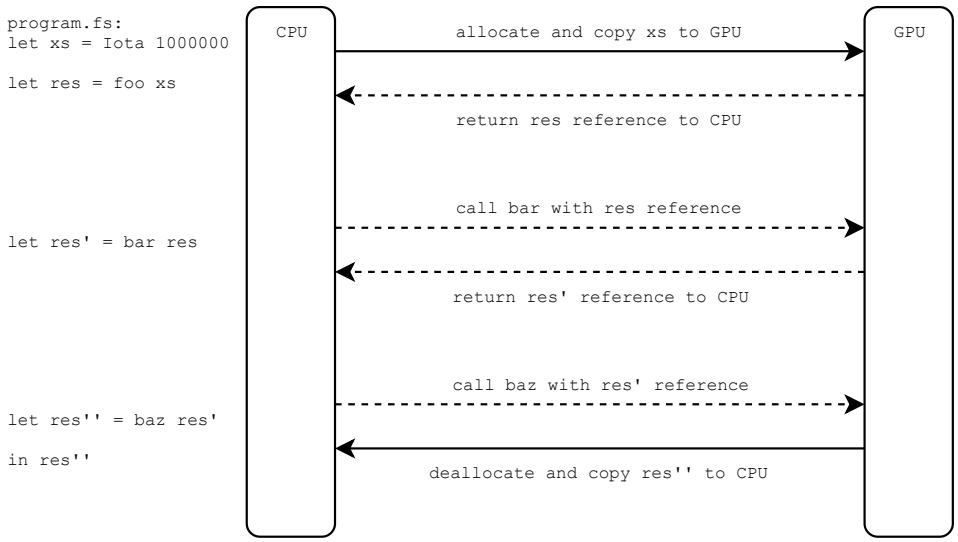


Figure 6.4: Buffers aren't copied between CPU and GPU unless necessary

This functionality is already implemented in Python's `PyOpenCL` library, and is used in Futhark programs that are compiled as Python libraries.

7.0

The **FShark** Compiler

In this chapter, we present the `FShark` compiler pipeline. We first present the compiler architecture, and followingly describe the four parts of the compiler piece by piece, with accompanying compilation and translation examples.

7.1. The **FShark** compiler architecture

At this point in the report, we are able to generate GPU-accelerated computational kernels for `C#`. We have also defined a language for writing GPU kernels using `F#`, and we have offered several methods of integrating these compiled `C#` kernels in `F#` projects. What remains is to build a compiler, that takes `FShark` code as input, and returns a compiled `C#` library that runs GPU computational kernels.

In practice, we need to build an architecture that supports the functionality defined in figure 7.1.

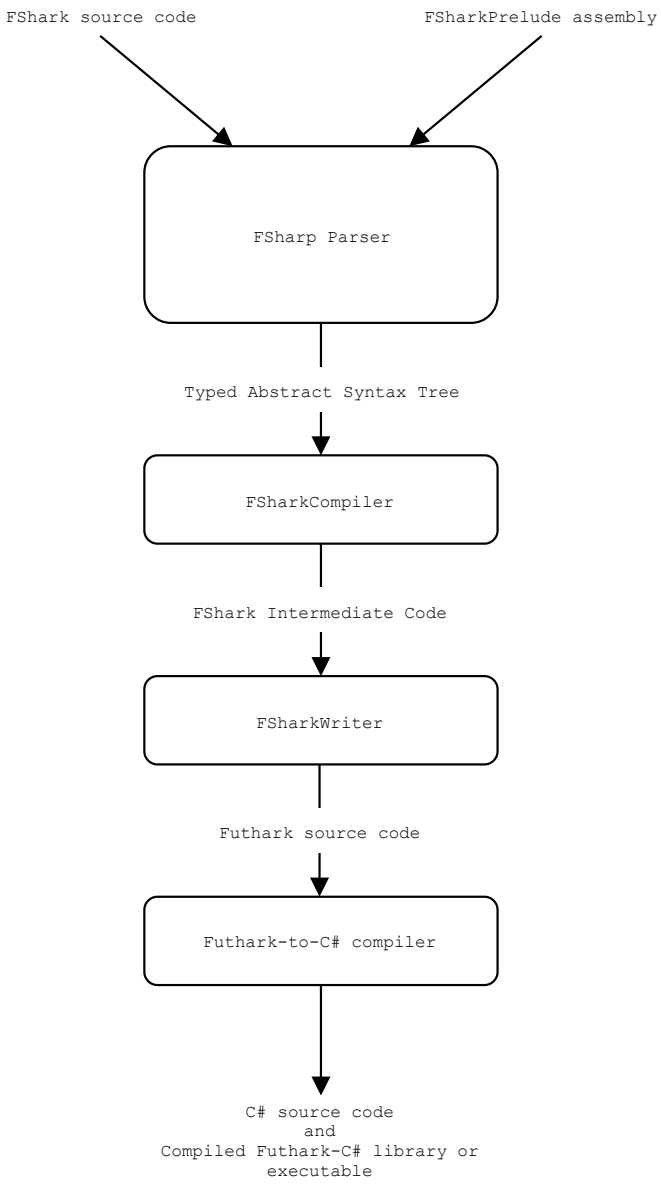


Figure 7.1: The complete architecture from `FShark` source code to compiled `C#` program.

7.2. The FSharp parser

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 output of the `F#` compiler. Instead, we use only part of the `F#` compiler's pipeline: By passing our `FShark` source code through the `F#` compiler's parser features, we retrieve its corresponding Typed Abstract Syntax Tree.

The Typed Abstract Syntax Tree (TAST) contains the function and value declarations that makes up our FShark program. The Typed Abstract Syntax Tree is merely an AST that already has tagged all the contained expressions with their respective types.
We take this TAST and pass it on into the FSharkCompiler.

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, instead of writing our own parser. So as the F# parser part of the pipeline amounts to calling some library functions from an imported library, we will not use more time on this part.

7.3. The **FShark**Compiler

For the FSharkCompiler, we need to build a module that takes an F# TAST as input, and returns the corresponding program as written in an intermediate language defined for FShark, called FSharkIL.
The declarations in the TAST are called FSharpDecls, and in FShark we work with two kinds of FSharpDecls.

The first kind of FSharpDecl that FShark supports is the FSharpDecl.MemberOrFunctionOrValue, which are declarations of members, functions and values. We don't use members in FShark, as they are for object oriented F# programming, but we do use functions and values. For all intents and purposes, F# values are just functions without arguments.

The other kind is the FSharpDecl.Entity. Entities are F# declarations that are neither functions or values themselves, but instead alterations to the present F# program. The FSharkCompiler supports three different entities.

In total, this means that our intermediate language must also support four different declarations, which are the FSharkDecls. These are shown in figure 7.6.

7.3.1. FSharpDecl.Entity

The FSharkCompiler supports three different entities.

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 FShark we don't support parameterized modules, so in reality the just work as namespaces for functions and values. 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.

An example of this module flattening is shown below.

```
module Vec3 =
    type Vec3single = {x:single ; y:single ; z:single}
    let plus (a : Vec3single) (b : Vec3single) : Vec3single =
        {x=a.x+b.x; y=a.y+b.y; z=a.z+b.z}

type vec3 = Vec3.Vec3single
type mass = single
type position = vec3
type acceleration = vec3
type velocity = vec3
```

```
type Vec3single = {x : f32, y : f32, z : f32}
let plus (a : Vec3single) (b : Vec3single) : Vec3single =
  {x=((a.x + b.x)), y=((a.y + b.y)), z=((a.z + b.z))}

type vec3 = Vec3single
type mass = f32
type position = Vec3single
type acceleration = Vec3single
type velocity = Vec3single
```

Here, the `Vec3` module is flattened and made part of the outer scope of the file. Due to time constraints, this current solution was chosen. The solution does introduce the danger of namespace collisions. for example, we could get in trouble by having a function further down which was also called `plus`.

A later version of the `FSharkCompiler` could very well contain a better solution to the module problem, either by translating `FShark` modules to Futhark modules, or at least by naming the flattened module declarations in a special way; for example by keeping the containing module's name in the flattened declarations' name, like in the example below:

```
module Vec3 =
    type Vec3single = {x:single ; y:single ; z:single}
    let plus (a : Vec3single) (b : Vec3single) : Vec3single =
        {x=a.x+b.x; y=a.y+b.y; z=a.z+b.z}

type vec3 = Vec3.Vec3single
```

```
type Vec3_Vec3single = {x : f32, y : f32, z : f32}
let Vec3_plus (a : Vec3_Vec3single) (b : Vec3_Vec3single) : Vec3_Vec3single =
  {x=((a.x + b.x)), y=((a.y + b.y)), z=((a.z + b.z))}

type vec3 = Vec3_Vec3single
```

7.3.2. **F#** expressions

An F# function or value is not without its accompanying F# expression. The F# compiler compiles these F# expressions into FSharpExprs, which we can parse ourselves, and rewrite as FSharkIL expressions. In figure 7.2 we see three different F# expressions, and their representations as FSharpExpr's.

For the first example, we just create a tuple literal. In the FSharpExpr version, we see how the literal is created using NewTuple. NewTuple takes a list of FSharpExprs as arguments. In this case, we are using two constants, each of which takes some primitive object, and the .NET type of that object.

In the second example, we use the Let expression. Semantically, Let takes a name n and two expressions e_1 and e_2, and exchanges every instance of n in e_2 with e_1. Then we encounter two Let-expressions that we didn't write ourselves. That is because the F# compiler turns tuple assignments into chains of Let expressions instead. Here, we use TupleGet to get each field of the tuple.

Finally, we use a typed instance of the Call expression to call the overloaded function Plus as the integer version of the plus function, giving it the two variables Value a and Value b as arguments.

In the third example, we see the Lambda expression in use. Lambda takes a list of variables (in the form of name/type pairs), and an FSharpExpr. In this case, Lambda only has one variable.
The innermost expression here is the Application expression. Application takes a function or a lambda, and a list of arguments, and applies the list of arguments one by one to the function or lambda.

```
// example 1
(79, 42.0f)

// example 2
let tuple = (79, 42.0f)
let (a,b) = tuple
in a + a

// example 3
let a = 2.0f
let foo = fun x -> x + 3.0f
in foo a
```

```
; example 1
NewTuple ([Const(79, System.Int32), Const(42.0f, System.Single)])

; example 2
Let tuple (NewTuple ([Const(79, System.Int32), Const(42.0f, System.Single)]))
(
  Let a (TupleGet 1 tuple)
  (
    Let b (TupleGet 2 tuple)
    (
      Call Plus System.Int32
        ([Value a, Value b])
    )
  )
)

; example 3
Let a Const(2.0f, System.Single)
(
  Let foo (Lambda ([(x, System.Single)])
                  (Call Plus System.Single
                    ([Value x, Const(3.0f, System.Single)])
                  ))
  (
    Application foo ([Value a])
  )

)
```

Figure 7.2: Three F# expressions, and their representation as FSharpExprs

The entire set of FSharpExprs used in the FSharkCompiler is available in figure 7.8, and the entire set of .NET types used in the FSharkCompiler is available in figure 7.7.

7.3.3. Translating from FSharpExprs to **FShark**IL

Now that we have the F# expressions and types in order, we can translate them into our FShark intermediate language, FSharkIL.

Continuing the example from figure 7.2, we will see three examples of such translations in figure 7.3. The expressions as written in `FSharkIL` happens to look very much like `FSharks` own expressions, except for one thing. In `F#`, infix operators are translated into normal function calls at compilation.

Therefore, the `FSharkCompiler` detects these specific function calls (the calls to infix operators), and translates them back into infix operations. Although it would possible to treat infix operators as functions in the `FShark` generated Futhark code, we have chosen to write the operator calls as infix operations, for readability.

```
; example 1
NewTuple ([Const(79, System.Int32), Const(42.0f, System.Single)])

; example 2
Let tuple (NewTuple ([Const(79, System.Int32), Const(42.0f, System.Single)]))
(
  Let a (TupleGet 1 tuple)
  (
    Let b (TupleGet 2 tuple)
    (
      Call Plus System.Int32
        ([Value a, Value b])
    )
  )
)

; example 3
Let a Const(2.0f, System.Single)
(
  Let foo (Lambda ([(x, System.Single)])
                  (Call Plus System.Single
                    ([Value x, Const(3.0f, System.Single)])
                  ))
  (
    Application foo ([Value a])
  )

)
```

```
; example 1
Tuple ([Const(79, FInt32), Const(42.0f, FSingle)])

; example 2
LetIn tuple (Tuple([Const(79, FInt32), Const(42.0f, FSingle)]))
(
  LetIn a (TupleGet tuple 1)
  (
    LetIn b (TupleGet tuple 2)
    (
      InfixOp Plus FInt32 (Var a) (Var b)
    )
  )
)

; example 3
LetIn a Const(2.0f, FSingle)
(
  LetIn foo (Lambda ([(x, FSingle)])
                  (InfixOp Plus FSingle (Var x) (Const(3.0f, FSingle)))
            )
  (
    Application foo ([Var a])
  )

)
```

Figure 7.3: Three `FSharpExprs` expressions, and their representation as `FSharkExprs`

The complete set of rules for translating .NET types to `FShark` types are available in figure 7.9, and the complete set of rules for translating `FSharpExprs` to `FSharkExprs` are shown in figure 7.10.

With the `FSharpExprs` translated to `FSharkExprs`, we can pass the entire program as written in `FShark` intermediate code onto the next part of the compiler pipeline; the `FShark`Writer.

7.4. The **`FShark`**Writer

The `FShark`Writer takes programs written in `FSharks` intermediate language, and translates them into valid Futhark source code.

Continuing the example from figure 7.3, we will see three examples of such translations in figure 7.4.

The complete set of `FSharkExpr`-to-Futhark rules are shown in figure 7.12, and the rules for translating `FSharkIL`-types to Futhark-types are shown in figure 7.11.

```
; example 1
Tuple ([Const(79, FInt32), Const(42.0f, FSingle)])

; example 2
LetIn tuple (Tuple([Const(79, FInt32), Const(42.0f, FSingle)]))
(
  LetIn a (TupleGet tuple 1)
  (
    LetIn b (TupleGet tuple 2)
    (
      InfixOp Plus FInt32 (Var a) (Var b)
    )
  )
)

; example 3
LetIn a Const(2.0f, FSingle)
(
  LetIn foo (Lambda ([(x, FSingle)])
                (InfixOp Plus FSingle (Var x) (Const(3.0f, FSingle)))
            )
  (
    Application foo ([Var a])
  )

)
```

```
-- example 1
(79i32, 42.0f32)

-- example 2
let tuple = (79i32, 42.0f32) in
let a = tuple.1 in
let b = tuple.2 in
a i32.+ b

-- example 3
let a = 2.0f32 in
let foo = (\(x : f32) -> x i32.+ 3.0f32) in
in foo a
```

Figure 7.4: Three `FSharkExprs` expressions translated to Futhark by `FSharkWriter`

The `FSharkCompiler` creates a temporary folder at the temporary root folder[1] that was passed as an argument when initializing the `FSharkWrapper`, and writes the finished string of Futhark source code to a Futhark file in this directory. The name of the Futhark file is taken from the `libName` variable we passed to the `FSharkWrapper` at its initialization.

[1] see the `tmpRoot` variable in sec 6.1.2

7.4.1. The Futhark-to-C# compiler

The FSharkWrapper now calls the Futhark-to-C# compiler `futhark-cs`. If we named our FShark module "MyLib", the command used be like shown below:

```
1 $ futhark-cs --library -o MyLib.dll MyLib.fut
```

`futhark-cs` compiles a C# dynamically linked library and writes it to `MyLib.dll`, and also writes the C# source code from the Futhark compilation to `MyLib.cs`.

This marks the end of the FShark compilation process.

7.5. Design choices in writing the FShark Compiler

To implement the `FSharkCompiler`, we had the choice between implementing a lexer and parser manually, or using F#'s own compiler as a parser library. Furthermore, F#'s Typed Abstract Syntax Trees shows the types of any functions and operators used in an F# program. This means, that even for the program shown in 7.5, the F# compiler's type inference tells us that the plus operator used in the expression on line 3 is the `int` plus operator.

```
1 let x = 7
2 let y = 9
3 in x + y
```

Figure 7.5: A short F# program that uses type inference to decide the types of the plus-operator's operands.

If we were to implement our own F# parser, we would not only have to implement the lexer and parser ourselves, but also type inference.

I chose to use F#'s own compiler as it gave me all this functionality for free, out of the box.

7.6. Figures

$$
\begin{array}{rcl}
\mathit{Decl} & := & \texttt{FSharkRecord}([(\mathit{field}_1, \mathit{decl}_1), \ldots, (\mathit{field}_n, \mathit{decl}_n)]) \\
& | & \texttt{FSharkTypeAlias}(\mathit{name}, \tau) \\
& | & \texttt{FSharkModule}(\mathit{name}, [\mathit{decl}_1, \ldots, \mathit{decl}_n]) \\
& | & \texttt{FSharkVal}(\mathit{name}, [\tau_1, \ldots, \tau_n], [\mathit{arg}_1, \ldots, \mathit{arg}_n], \tau_{\mathit{return}}, e)
\end{array}
$$

Figure 7.6: The four possible `FSharkDecls`.

$$
\begin{array}{rcl}
\tau & = & \texttt{System.Int8} \\
& | & \texttt{System.Int16} \\
& | & \texttt{System.Int32} \\
& | & \texttt{System.Int64} \\
& | & \texttt{System.UInt8} \\
& | & \texttt{System.UInt16} \\
& | & \texttt{System.UInt32} \\
& | & \texttt{System.UInt64} \\
& | & \texttt{System.Single} \\
& | & \texttt{System.Double} \\
& | & \texttt{System.Boolean} \\
& | & \texttt{System.Array}\ \tau \\
& | & \texttt{System.Tuple}\ (\tau_1 \times \ldots \times \tau_n)
\end{array}
$$

Figure 7.7: The .NET types used in the `FShark`Compiler

$$
\begin{array}{rcl}
\text{e} & = & \text{Const(obj, } \tau) \\
& | & \text{Value(v)} \\
& | & \text{AddressOf(v)} \\
& | & \text{NewTuple(\_, } [e_0, ..., e_n]) \\
& | & \text{NewRecord}([(v_0, e_0),\ \ldots,\ (v_n, e_n)]) \\
& | & \text{NewArray}(\tau, [e_0, ..., e_n]) \\
& | & \text{TupleGet(\_, i, e)} \\
& | & \text{FSharpFieldGet}(e, \_, field) \\
& | & \text{Call}(\_, \texttt{GetArray}, \_, nil, [e_0, e_1]) \\
& | & \text{Call}(\_, name, \_, nil, [e_0, \ldots, e_n]) \\
& | & \text{Call}(\_, name, \_, \tau, [e_0, \ldots, e_n]) \\
& | & \text{Call}(\_, infixOp, \_, \tau, [e_0, e_1]) \\
& | & \text{Call}(\_, unaryOp, \_, \tau, [e_0]) \\
& | & \text{Let}(p, e_0, e_1) \\
& | & \text{IfThenElse}(e_0, e_1, e_2) \\
& | & \text{Lambda}([(v_1 : \tau_1), \ldots, (v_n : \tau_n)], e) \\
& | & \text{Application}(func, \_, [e_0, \ldots, e_n]) \\
& | & \text{TypeLambda(e)} \\
& | & \text{DecisionTree(\_, \_)} \\
& | & \text{DecisionTreeSuccess(\_, \_)}
\end{array}
$$

Figure 7.8: The `FSharpExprs` used in the `FShark`Compiler

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

Figure 7.9: Translation rules for .NET-types to FSharkIL-types translations.

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

Figure 7.10: Translation rules for FSharp expressions to FSharkIL expressions

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

Figure 7.11: Translation rules from `FSharkIL` types to Futhark types

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

Figure 7.12: Translation rules from `FSharkIL` expressions to Futhark source code

8.0

Current limitations

In the chapter, we describe the current known limitations of both our code generator, the `FShark` language design and of the `FShark` compiler. The limitations are divided into two categories; those caused by our design choices, and those caused by a lacking implementation.

8.1. The **C#** code generator

Both of the code generator limitations listed below are caused by lack of implementation.

8.1.1. Errors in the implementation

Using Futhark's own test suite, we have tested and evaluated the current implementation of the code generator. These tests have made it apparent that the `C#` code generator does not yet generate fully correct Futhark programs.

The Futhark test suite contains 961 tests, which tests everything from the Futhark compiler itself (for example whether type aliases, higher order modules et al. handled correctly?), to whether the individual mathematical and bitwise operations are correctly translated from Futhark to desired target language (`C#` in our case), and whether types, be they array types or scalar types, are retained throughout the entire program.
It also tests the `stdin/stdout` functionality of the generated programs.
The current implementation does not pass all tests correctly: for the Futhark `C#` compiler for non-OpenCL programs, we are passing 840 out of 961 tests.
For the Futhark `C#` compiler for OpenCL programs, we are passing just 833 out of 961 tests.

For many of these tests, we can immediately see what the issue is. In example, we our stdin reader for `C#` programs does not currently read empty arrays[1] correctly. Such errors are caused by lack of implementation in the stdin reader, and can be corrected with relative ease by implementing the missing features. The test suite also reveals various off-by-one errors in edge cases of the implementation, which should be simple to debug and fix.

[1] such as example `empty(i32)` or `empty(f32)`.

Despite of these temporary bad test results, we are still confident that our code generator is mostly correct. After all, the current implementation does pass more than 86% of the test suite.
More importantly, although we have 128 failing tests, the test results also show that many of these tests have the same point of failure.

For example, we have 16 failing tests caused by the missing reader features, and 8 `off-by-one` errors stemming from a certain helper function in the `C#` class. If this is representative for the other failing tests as well, it is likely that the remaining number of (known) bugs in the implementation is closer to ten, than to a hundred.

8.1.2. Errors in the benchmarking functionality

The current implementation contains an error which hinders us in reliably testing all benchmarks in the Futhark benchmark suite. In rare cases, a benchmark program will both compile and execute correctly, but return the wrong runtime measurements.

In principle, this bug should disqualify us from being able to report any reliable benchmarks from our solution at all, but repeated testing shows that the bug is indeed only present for certain benchmark programs.

For example, the `Crystal` benchmark executes correctly for both Futhark `C` and Futhark `C#`, but the runtimes reported are obviously wrong for the Futhark `C#` case:

```
$ futhark-bench --compiler=futhark-opencl crystal.fut
Compiling ./crystal.fut...
Results for ./crystal.fut:
dataset #0 ("200i32 30.0f32 5i32 1i32 1.0f32"):        32.00us
  ↪  (avg. of 10 runs; RSD: 0.37)
dataset #1 ("20i32 30.0f32 5i32 50i32 0.5f32"):        21.20us
  ↪  (avg. of 10 runs; RSD: 0.04)
dataset #2 ("40i32 30.0f32 5i32 50i32 0.5f32"):        40.70us
  ↪  (avg. of 10 runs; RSD: 0.02)
dataset #3 ("40i32 30.0f32 50i32 50i32 0.5f32"):      263.40us
  ↪  (avg. of 10 runs; RSD: 0.27)
dataset #4 ("2000i32 30.0f32 50i32 1i32 1.0f32"):   10800.40us
  ↪  (avg. of 10 runs; RSD: 0.04)

$ futhark-bench --compiler=futhark-csopencl crystal.fut
Compiling ./crystal.fut...
Results for ./crystal.fut:
dataset #0 ("200i32 30.0f32 5i32 1i32 1.0f32"):        32.20us
  ↪  (avg. of 10 runs; RSD: 0.54)
dataset #1 ("20i32 30.0f32 5i32 50i32 0.5f32"):        31.90us
  ↪  (avg. of 10 runs; RSD: 0.58)
dataset #2 ("40i32 30.0f32 5i32 50i32 0.5f32"):        29.30us
  ↪  (avg. of 10 runs; RSD: 0.57)
dataset #3 ("40i32 30.0f32 50i32 50i32 0.5f32"):       31.60us
  ↪  (avg. of 10 runs; RSD: 0.65)
```

```
dataset #4 ("2000i32 30.0f32 50i32 1i32 1.0f32"):      28.70us
↪  (avg. of 10 runs; RSD: 0.62)
```

On the other hand, the Hotspot benchmark works as expected:

```
$ futhark-bench --compiler=futhark-opencl hotspot.fut
Compiling ./hotspot.fut...
Results for ./hotspot.fut:
dataset data/64.in:      1686.00us (avg. of 10 runs; RSD: 0.07)
dataset data/512.in:    13064.60us (avg. of 10 runs; RSD: 0.01)
dataset data/1024.in:   42276.30us (avg. of 10 runs; RSD: 0.01)
$ futhark-bench --compiler=futhark-csopencl hotspot.fut
Compiling ./hotspot.fut...
Results for ./hotspot.fut:
dataset data/64.in:      2172.10us (avg. of 10 runs; RSD: 0.06)
dataset data/512.in:     9172.20us (avg. of 10 runs; RSD: 0.42)
dataset data/1024.in:   33946.10us (avg. of 10 runs; RSD: 0.46)
```

8.1.3. Cumbersome array entry functions in Futhark libraries

As described in section 4.3.5, we currently have to flatten our jagged arrays before we can pass them to our Futhark library functions. Likewise, we have to unflatten the results if we want to use them as jagged arrays again afterwards.

In sec. 5.8, we presented a solution for both flattening and unflattening such arrays, and thus solving this limitation is merely a question of porting and implementing these algorithms in the Futhark generated `C#` libraries.

8.1.4. Unnecessary memory allocations in chained Futhark function calls

The current implementation of the code generator causes significant overhead when chaining together GPU function calls, as discussed in sec. 6.2.2. Whilst not being a functional limitation, implementing an opaque return type for Futhark GPU calls would increase runtime performance in any programs that chain together such calls.

The Python code generator for Futhark already has such an opaque data type implemented, and one could look to this implementation for inspiration on how to design a similar data type for Futhark `C#`.

8.2. The **`FShark`** language

8.2.1. Scatter

The current implementation of the `FShark` language allows users to use the SOAC `Scatter`. However, the actual `scatter` SOAC from Futhark has cer-

tain usage constraints concerning uniqueness[2], which we currently do not enforce in FShark.

This means that users can unknowingly write valid FShark programs, that are not valid Futhark programs, if they accidentally break the rules of scatter usage. This is not a nice position for the FShark language to be in, so we should in principle either remove the Scatter SOAC from the FShark standard library, or enforce the uniqueness constraint in the FShark compiler. This might take some time.

8.3. The **FShark** compiler

8.3.1. Disallowing certain types of **FShark** entry functions

As the FShark wrapper relies on the flattening algorithms shown in sec. 5.8 to make F#s jagged arrays compatible with Futhark's flat arrays (sec. 4.3.5), we currently prohibit FShark entry functions have return types that are either ('a[] * int64[]) tuples, or tuples or arrays that contains such tuples. This is described in detail in subsec. 5.8.3.

This could be solved by moving the array flattening into the generated Futhark C# libraries as described in subsec. 8.1.3, solving two limitations at the same time.

8.3.2. Allow compiler usage outside of **FShark** wrapper

As discussed in section 6.2.2, we would like to be able to use just the FShark compiler, without having to go through the FShark wrapper. Our goal is to be able to compile FShark programs directly from the command-line like so:

```
1 $ fshark -o MyModule.dll MyModule.fs
```

This should be relatively easy to achieve, as the compiler architecture already exists within the FShark project.

8.4. The **FShark** validation

The current FShark test suite lacks accompanying tests for the FShark type conversion functions. These must be implemented to assure us that FShark correctly translate type convertion functions to Futhark.

[2] See https://futhark.readthedocs.io/en/latest/language-reference.html#in-place-updates

9.0

Evaluation and benchmarks

In this chapter we first evaluate both the correctness of our code generator, and the performance of the `C#` programs that our code generator generates. For testing correctness, we use the Futhark compiler's existing test suite with our new code generator. For the performance evaluation, we run and compare benchmark results between programs generated by the Futhark `C#`-, `C`- and `Python` code generators.

We then evaluate whether the `FShark` language succeeds in letting us write complex GPU benchmarks in an idiomatic `F#` style.

Finally, we evaluate the `FShark` compiler itself. First we test whether the `FShark` compiler correctly translates `FShark` programs to Futhark, so that they are functionally equivalent. We then compile and compare the performance of GPU benchmarks written in `FShark` with equivalent benchmarks written in Futhark.

Specifications for benchmark

For all benchmarks in this section, we have run the benchmarks on a system with these attributes:

- 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

9.1. Correctness of the Futhark csharp generator

To show that the Futhark `C#` code generator correctly translates Futhark to `C#` programs, we have chosen to test our solution using the already existing Futhark test suite, as described in section 8.1.1.

Although the `C#` code generator currently does not pass all the tests in the test suite, we believe that the Futhark-to-`C#` translation itself is correct.

However, there are still parts of the generated Futhark C# programs which contains errors. Therefore we cannot say that the current implementation is completely correct.

9.2. The performance of Futhark C# programs

To determine whether Futhark C# programs have similar performance to Futhark C and Futhark Python programs, we have used Futhark's own benchmark suite. We have measured the runtime of 22 benchmarks compiled with the Futhark C# code generator, and used the performance of the Futhark C-OpenCL-generated kernels as a reference point.
The results are shown in table 9.2.

We have sorted the benchmarks by their runtimes. Short benchmarks are easily influenced by "background noise" from the operating system, whereas small fluctutations in running time has much less effect on longer running benchmarks.

Our benchmarks doesn't show that either the C# or the C GPU kernels are the better choice to any significant degree. For the two longest benchmarks, we see that the C# kernels do run faster than the reference, but not more than $\sim 3\%$. For the remaining benchmarks, the two versions are mostly comparable. We see a large speed difference for the Hotspot benchmark, but it also has a high relative standard deviation.

As we go further down the list we also see that BFS Iterative Partitioning and NN have large speed differences, without much relative standard deviation. However, at this point we are running benchmarks that lasts for few milliseconds, so we will not investigate the difference further.

On the grounds of these benchmarks, we are confident that our C# code generator is able to build C# programs that are comparable in performance to corresponding C versions.

However, as described in 8.1.2, we are also aware that the current implemention Futhark C# implementation contains errors, which for some benchmarks makes it impossible to reliably measure their runtimes. We still have more benchmarks to run when these errors are fixed, but we are confident that future benchmarks will support our assertation, rather than disprove it.

Benchmark	C#	Ref.	Speedup in percentage	RSD
LocVolCalib	1879302	1940669	+3.16%	0.00
Particle Filter	396207	412421	+3.93%	0.01
Myocyte	395485	396866	+0.35%	0.00
Stencil	158911	156194	−1.74%	0.01
OptionPricing	153315	154702	+0.9%	0.00
LUD clean	131038	132479	+1.09%	0.00
NW	62752	62995	+0.39%	0.01
BFS Flattened	40708	39384	−3.36%	0.01
Hotspot	34073	44038	+22.63%	0.46
BFS Padded	30924	33047	+6.42%	0.06
High Pass Filter (FFT)	27156	22252	−22.04%	0.02
Radix-sort	21805	22462	+2.92%	0.02
SRAD	13813	13456	−2.65%	0.04
Backprop	12576	12887	+2.41%	0.07
Pagerank	8987	8796	−2.17%	0.01
BFS Iterative Partitioning	7193	6062	−18.66%	0.04
NN	7127	8197	+13.05%	0.02
BFS Heuristic	6468	6275	−3.08%	0.02
Fluid	1316	1060	−24.15%	0.06
LUD	570	532	−7.13%	0.13
Canny Edge Detection	241	202	−19.31%	0.07
Raytracer	218	351	+37.89%	0.17

Figure 9.1: Average benchmark runtime in microseconds

Benchmark	Dataset
LocVolCalib	large dataset
Particle Filter	128_128_10_image_400000_particles.in
Myocyte	medium dataset
Stencil	default dataset
OptionPricing	large dataset
LUD Clean	2048×2048
NW	medium dataset
BFS Flattened	64kn_32e-var-1-256-skew.in
Hotspot	1024×1024
BFS Padded	64kn_32e-var-1-256-skew.in
High Pass Filter (FFT)	1024×1024
Radix-sort	$N = 10^6$
SRAD	image.in
Backprop	medium dataset
Pagerank	random_medium.in
BFS Iterative Partitioning	64kn_32e-var-1-256-skew.in
NN	medium dataset
BFS Heuristic	64kn_32e-var-1-256-skew.in
Fluid	medium dataset
LUD	64×64
Canny Edge Detection	Lena (512×512)
Raytracer	dataset #0

Figure 9.2: Datasets used for benchmarks

9.3. Futhark **C#** integration in **C#** programs

Our implemented code generator allows us to use Futhark libraries in C#, as expected. The example below is taken from the file Program.cs in the CSharpTest folder in the FShark repository. In this folder, we find a compiled C# .dll file which is the result of compiling the LocVolCalib benchmark with the futhark-csopencl compiler.

```
using System;
using LocVolCalib;
namespace CSharpTest
{
    internal class Program
    {
        public static void Main(string[] args)
        {
            var lvc = new LocVolCalib.LocVolCalib(args);
            var res = lvc.main(256, 256, 256, 64, 0.03f, 5.0f, 0.6f, 0.5f,
            ↪  0.2f);
            var result_array = res.Item1;
            Console.WriteLine(result_array);
        }
    }
}
```

First we instantiate an instance of the LocVolCalib class from the Futhark C# class with the arguments that we passed to the main method of our Program class. This lets us pass flags to the LocVolCalib class, which for example lets us

specify the number of runs we want our benchmark to execute.

Then we run the benchmark by calling the `lvc.main` method as shown. The program behaves as expected, and prints the benchmark results after the benchmark run.

As discussed in section 4.3.5, the current return type is cumbersome, but such is the current state of affairs.

9.4. The design of the **FShark** language

One of the goals of `FShark` is to enable users to write complex GPU programs using idiomatic `F#` code.

To test this, we have taken two already existing benchmark implementations from Futhark, and manually translated them to `FShark`. We will not elaborate on the functionality of the benchmarks in these two examples. Instead we will focus on the code style used in `FShark` programs.

9.4.1. LocVolCalib

First, we present the `FShark` version of the LocVolCalib benchmark (respectively FSharpTests/Benchmarks/LocVolCalib.fs and finpar/LocVolCalib.fut in the `FShark` and Futhark benchmarks repository.) from the Finpar[1] benchmark suite.

We have chosen this benchmark as it is features plenty of nested SOACs, and is in general a structurally complex program.

Below, we show multiple snippets of the `FShark` and Futhark version of the LocVolCalib benchmark to demonstrate

```
;; Snippet from FShark's LocVolCalib
let explicitMethod (myD:    float32 [] []) (myDD: float32 [] [])
                   (myMu:   float32 [] []) (myVar: float32 [] [])
                   (result: float32 [] [])
                  : float32 [] [] =
  // 0 <= i < m AND 0 <= j < n
  let m = Length myDD
  Map3 (fun (mu_row : float32 []) (var_row : float32 []) (result_row : float32
  ↪  []) ->
    Map5 (fun (dx : float32 []) (dxx : float32 []) (mu : float32) (var : float32)
    ↪  (j : int) ->
      let c1 = if 0 < j
               then (mu*dx.[0] + 0.5f*var*dxx.[0]) * result_row.[j-1]
               else 0.0f
      let c3 = if j < (m-1)
               then (mu*dx.[2] + 0.5f*var*dxx.[2]) * result_row.[j+1]
               else 0.0f
      let c2 = (mu*dx.[1] + 0.5f*var*dxx.[1]) * result_row.[j]
      in  c1 + c2 + c3) myD myDD mu_row var_row <| (Iota m)
   ) myMu myVar result

// for implicitY: should be called with transpose(u) instead of u
let implicitMethod (myD:  float32 [] [])  (myDD:  float32 [] [])
                   (myMu: float32 [] [])  (myVar: float32 [] [])
                   (u:    float32 [] [])  (dtInv: float32)
                    : float32 [] [] =
  Map3 (fun (mu_row : float32 []) (var_row : float32 []) (u_row : float32 []) ->
    let (a,b,c) = Unzip3 (
      Map4 (fun (mu : float32) (var : float32) (d : float32 []) (dd : float32 [])
      ↪  ->
```

```
28          (0.0f - 0.5f*(mu*d.[0] + 0.5f*var*dd.[0]), dtInv - 0.5f*(mu*d.[1] +
            ↪  0.5f*var*dd.[1]),
29          0.0f   - 0.5f*(mu*d.[2] + 0.5f*var*dd.[2])
30          )
31        ) mu_row var_row myD myDD
32      )
33      in tridagPar a b c u_row
34    ) myMu myVar u
```

```
1   -- Snippet from Futhark's LocVolCalib
2   let explicitMethod [m][n] (myD:    [m][3]f32,  myDD: [m][3]f32,
3                              myMu:   [n][m]f32,  myVar: [n][m]f32,
4                              result: [n][m]f32)
5                     : *[n][m]f32 =
6     -- 0 <= i < m AND 0 <= j < n
7     map3 (\mu_row var_row result_row ->
8       map5 (\dx dxx mu var j ->
9         let c1 = if 0 < j
10                then (mu*dx[0] + 0.5*var*dxx[0]) * unsafe result_row[j-1]
11                else 0.0
12        let c3 = if j < (m-1)
13                then (mu*dx[2] + 0.5*var*dxx[2]) * unsafe result_row[j+1]
14                else 0.0
15        let c2 =       (mu*dx[1] + 0.5*var*dxx[1]) * unsafe result_row[j  ]
16        in  c1 + c2 + c3) myD myDD mu_row var_row (iota m)
17    ) myMu myVar result
18
19  -- for implicitY: should be called with transpose(u) instead of u
20  let implicitMethod [n][m] (myD:  [m][3]f32,  myDD:  [m][3]f32,
21                             myMu: [n][m]f32,  myVar: [n][m]f32,
22                             u:   *[n][m]f32,  dtInv: f32)
23                    : *[n][m]f32 =
24    map3 (\mu_row var_row u_row  ->
25      let (a,b,c) = unzip3
26        (map4 (\mu var d dd ->
27          ( 0.0   - 0.5*(mu*d[0] + 0.5*var*dd[0])
28          , dtInv - 0.5*(mu*d[1] + 0.5*var*dd[1])
29          , 0.0   - 0.5*(mu*d[2] + 0.5*var*dd[2]))
30        ) mu_row var_row myD myDD)
31      in tridagSeq (a, copy b, c, copy u_row )
32    myMu myVar u
```

There are a couple of differences. The Futhark version can define the lengths of the dimensions of its arrays in the function definition. These lengths can then be used as variables in the function body. An example of this is shown in the Futhark example: At line 2 we define that our input arrays have $n * m$ elements, or in some cases $3 * m$, and we can then use these lengths, such as in the if-expression on line 12.

The second difference is on line 31 of the Futhark version. Here, we copy the array used for our function call, which is a feature used for Futhark's uniqueness types[18]. As we don't have uniqueness types in `FShark`, we can leave out this copy in the `FShark` version, as shown in line 33.

The third difference is the usage of the `unsafe` expression in Futhark's example at line 10, 13 and 15. We need them in Futhark to circumvent Futhark's boundary checks for array indexing, but we can leave them out in the `FShark` version, as `FShark` doesn't have that kind of boundary checks.

The second pair of snippets are shown below.

```
let value (numX: int) (numY: int) (numT: int) (s0: float32) (strike: float32)
          (t: float32) (alpha: float32) (nu: float32) (beta: float32): float32 =
  let (myXindex, myYindex, myX, myY, myTimeline) = initGrid s0 alpha nu t numX
  ↪  numY numT
  let (myDx, myDxx) = initOperator myX
  let (myDy, myDyy) = initOperator myY
  let myResult = setPayoff strike myX myY
  let myTimeline_neighbours = Reverse (Zip (Init myTimeline) (Tail myTimeline))

  let myResult' =
    Foldl (fun oldResult (tnow,tnext) ->
      let (myMuX, myVarX, myMuY, myVarY) =
        updateParams myX myY tnow alpha beta nu
      in rollback tnow tnext oldResult
            myMuX myDx myDxx myVarX
            myMuY myDy myDyy myVarY
      ) myResult myTimeline_neighbours
  in myResult'.[myYindex].[myXindex]
```

```
let value(numX: i32, numY: i32, numT: i32, s0: f32, strike: f32,
          t: f32, alpha: f32, nu: f32, beta: f32): f32 =
  let (myXindex, myYindex, myX, myY, myTimeline) =
    initGrid(s0, alpha, nu, t, numX, numY, numT)
  let (myDx, myDxx) = initOperator(myX)
  let (myDy, myDyy) = initOperator(myY)
  let myResult = setPayoff(strike, myX, myY)
  let myTimeline_neighbours = reverse (zip (init myTimeline) (tail myTimeline))

  let myResult = loop (myResult) for (tnow,tnext) in myTimeline_neighbours do
      let (myMuX, myVarX, myMuY, myVarY) =
        updateParams(myX, myY, tnow, alpha, beta, nu)
      let myResult = rollback(tnow, tnext, myResult,
                              myMuX, myDx, myDxx, myVarX,
                              myMuY, myDy, myDyy, myVarY)

      in myResult
  in myResult[myYindex,myXindex]
```

There are two major differences: First, the `FShark` version doesn't take one single tuple as the function argument, but does instead need to take the tuple elements as arguments individually. This is because of the `F#` compiler's currying, as described in section 5.3.1.

Second, we have translated the Futhark version's for loop on line 9 into a `Foldl` on line 9 of the `FShark` version. This is because the current version of `FShark` doesn't support `for-loops`. Otherwise, the two expressions are equivalent.

The complete `FShark` version is available in appendix D, with the Futhark version available for reference.

9.4.2. nbody

Below, we present the `FShark` version of the nbody benchmark (respectively FSharpTests/Benchmarks/nbody.fs and accelerate/nbody/nbody.fut in the `FShark` and Futhark benchmarks repository.) from the Accelerate[5] benchmark suite.

We have chosen this benchmark to show how matrix manipulation functions can be implemented in `FShark`, and how `FShark` have support for type aliases and records similar to Futhark.

The examples from the nbody benchmark are shown below:

```
// Snippet from FShark's nbody

module Vec3 =
    type Vec3single = {x:single ; y:single ; z:single}
    let plus (a : Vec3single) (b : Vec3single) : Vec3single =
        {x=a.x+b.x; y=a.y+b.y; z=a.z+b.z}

    let minus (a : Vec3single) (b : Vec3single) : Vec3single =
        {x=a.x-b.x; y=a.y-b.y; z=a.z-b.z}

    let dot (a : Vec3single) (b : Vec3single) : single =
        a.x*b.x + a.y*b.y + a.z*b.z

    let scale (a : single) (b : Vec3single) : Vec3single =
        {x=a*b.x; y=a*b.y; z=a*b.z}

    let norm (a : Vec3single) : single =
        sqrt <| a.x*a.x + a.y*a.y + a.z*a.z

    let normalise (v : Vec3single) : Vec3single =
        let l = norm v
        in scale (1.0f / l) v

type vec3 = Vec3.Vec3single
type mass = single
type position = vec3
type acceleration = vec3
type velocity = vec3

type body = { position: position;
              mass: mass;
              velocity: velocity;
              acceleration: acceleration
            }

// ...

let advance_body (time_step: single) (body: body) (acc: acceleration): body =
  let acceleration = Vec3.scale body.mass acc
  let position = Vec3.plus body.position <| Vec3.scale time_step body.velocity
  let velocity = Vec3.plus body.velocity <| Vec3.scale time_step acceleration
  in {position=position; mass=body.mass; velocity=velocity;
  ↪  acceleration=acceleration}

let advance_bodies (epsilon: single) (time_step: single) (bodies: body []): body
↪  [] =
  let accels = calc_accels epsilon bodies
  in Map2 (fun body acc -> advance_body time_step body acc) bodies accels

let advance_bodies_steps (n_steps: int) (epsilon: single) (time_step: single)
                         (bodies: body []): body [] =
  let steps = Iota n_steps
  in Foldr (fun _ bodies' -> advance_bodies epsilon time_step bodies') bodies
  ↪  steps

let wrap_body (posx : single) (posy : single) (posz : single) (mass : single)
↪  (velx : single)
              (vely : single) (velz : single) (accx : single) (accy : single)
              ↪  (accz : single)
              : body =
  {position={x=posx; y=posy; z=posz};
   mass=mass;
   velocity={x=velx; y=vely; z=velz};
   acceleration={x=accx; y=accy; z=accz}}

```

```
let unwrap_body (b: body): (single * single * single * single * single * single *
↪  single * single * single * single) =
  (b.position.x, b.position.y, b.position.z,
   b.mass,
   b.velocity.x, b.velocity.y, b.velocity.z,
   b.acceleration.x, b.acceleration.z, b.acceleration.z)

// ...

let rotatePointByMatrix (rotation: single [] []) (p: position): position =
  let x = p.x
  let y = p.y
  let z = p.z
  {x= x*rotation.[0].[0] + y*rotation.[1].[0] + z*rotation.[2].[0];
   y= x*rotation.[0].[1] + y*rotation.[1].[1] + z*rotation.[2].[1];
   z= x*rotation.[0].[2] + y*rotation.[1].[2] + z*rotation.[2].[2]}

let rotatePointsByMatrix (rotation: single [] []) (ps: position []): position []
↪  =
  Map (rotatePointByMatrix rotation) ps

let rotateXMatrix (angle: single): single [] [] =
  [|
    [|1.0f;       0.0f;          0.0f|];
    [|0.0f;  cos angle;   - sin angle|];
    [|0.0f;  sin angle;     cos angle|]
  |]

let rotateYMatrix (angle: single): single [] [] =
  [|
    [|cos angle  ; 0.0f; sin angle|];
    [|0.0f       ; 1.0f; 0.0f     |];
    [|- sin angle; 0.0f; cos angle|]
  |]

let matmult (x: single [] []) (y: single [] []) : single [] [] =
  let Sum = Reduce (+) 0.0f
  Map (fun xr ->
        Map (fun yc -> Sum (Map2 (fun x y -> x*y) xr yc)) (Transpose y)
  ) x
```

```
-- Snippet from Futhark's nbody
module vec3 = mk_vec3 f32

type mass = f32
type position = vec3.vec
type acceleration = vec3.vec
type velocity = vec3.vec
type body = {position: position,
             mass: mass,
             velocity: velocity,
             acceleration: acceleration}

let advance_body (time_step: f32) (body: body) (acc: acceleration): body =
  let acceleration = vec3.scale body.mass acc
  let position = vec3.(body.position + scale time_step body.velocity)
  let velocity = vec3.(body.velocity + scale time_step acceleration)
  in {position, mass=body.mass, velocity, acceleration}

let advance_bodies [n] (epsilon: f32) (time_step: f32) (bodies: [n]body): [n]body =
  let accels = calc_accels epsilon bodies
  in map2 (advance_body time_step) bodies accels

let advance_bodies_steps [n] (n_steps: i32) (epsilon: f32) (time_step: f32)
                             (bodies: [n]body): [n]body =
  loop bodies for _i < n_steps do
```

```
26      advance_bodies epsilon time_step bodies
27
28  let wrap_body (posx: f32, posy: f32, posz: f32)
29                (mass: f32)
30                (velx: f32, vely: f32, velz: f32)
31                (accx: f32, accy: f32, accz: f32): body =
32    {position={x=posx, y=posy, z=posz},
33     mass,
34     velocity={x=velx, y=vely, z=velz},
35     acceleration={x=accx, y=accy, z=accz}}
36
37  let unwrap_body (b: body): ((f32, f32, f32), f32, (f32, f32, f32), (f32, f32, f32)) =
38    ((b.position.x, b.position.y, b.position.z),
39     b.mass,
40     (b.velocity.x, b.velocity.y, b.velocity.z),
41     (b.acceleration.x, b.acceleration.y, b.acceleration.z))
42
43  //...
44
45  let rotatePointByMatrix (rotation: [3][3]f32) ({x,y,z}: position): position =
46    {x= x*rotation[0,0] + y*rotation[1,0] + z*rotation[2,0],
47     y= x*rotation[0,1] + y*rotation[1,1] + z*rotation[2,1],
48     z= x*rotation[0,2] + y*rotation[1,2] + z*rotation[2,2]}
49
50  let rotatePointsByMatrix [n] (rotation: [3][3]f32)(ps: [n]position): [n]position =
51    map (rotatePointByMatrix rotation) ps
52
53  let rotateXMatrix (angle: f32): [3][3]f32 =
54    [[1f32,        0f32,         0f32],
55     [0f32, f32.cos angle, -f32.sin angle],
56     [0f32, f32.sin angle,  f32.cos angle]]
57
58  let rotateYMatrix (angle: f32): [3][3]f32 =
59    [[f32.cos angle,  0f32, f32.sin angle],
60     [0f32,           1f32, 0f32],
61     [-f32.sin angle, 0f32, f32.cos angle]]
62
63  let matmult [n][m][p] (x: [n][m]f32) (y: [m][p]f32): [n][p]f32 =
64    map (\xr ->
65          map (\yc -> f32.sum (map2 (*) xr yc))
66              (transpose y))
67        x
```

There are several differences between the `FShark` and the Futhark versions here. The primary difference is, that the Futhark version supports higher-order modules. This is shown in line 2 of the Futhark example, where we instantiate the higher-order 3D vector module `vec3` with the single precision floating point type `f32` module.

`FShark` does not at this point support similar higher-order modules, so we fake it by implementing a simple 3D vector module manually. Then, for both the `FShark` and the Futhark version, we state a list of type aliases for the rest of the program.

We then see five functions `advance_body`, `advance_bodies`, `advance_bodies_steps`, `wrap_body` and `unwrap_body`. The differences between the `FShark` and Futhark versions are negliable, with one except. As in the LocVolCalib example, we don't support for-loops in `FShark`. Therefore we have replaced the for-loop with a fold instead, obtaining the same functional-

ity.

Finally, we see five matrix helper functions. For both of the rotation functions `rotateXMatrix` and `rotateYMatrix`, the `FShark` version infers the types of the overloaded `cos` and `sin` functions automatically, whereas the Futhark developer must access those two functions through their containing modules instead.

Also, `FShark` doesn't have a `sum` operator in it's standard library like Futhark has, so we implement it ourselves by defining it in the `FShark` version at line 95.

The complete `FShark` version is available in appendix E, with the Futhark version available for reference.

9.4.3. Conclusion on **FShark** language design

As shown in the two example benchmarks above, we can now write complex GPU benchmarks in `F#` using purely functional idiomatic `F#`. We can use nested SOACs as we like, and use records and type aliases to improve the readibility of the code.

9.5. The correctness of the **FShark** subset.

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, we have written a comprehensive test suite with unit tests for each operator, standard library function and SOAC in the `FShark` language.

An `FShark` test is an `FShark` program, 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 below. It is a unit test for the division operator, and contains six test cases, namely division for signed integers of (8, 16, 32, 64 bits) and single- and double precision floating points

```
1  module Div
2  open FSharkPrelude.FSharkPrelude
3  open FShark.TestTypes.TestTypes
4  open System
5
6  [<FSharkEntry>]
7  let div (fiveByte : int8) (fiveShort : int16) (five : int)
8          (fiveLong : int64) (fiveSingle : single) (fiveDouble : double)
9          : (int8 * int16 * int * int64 * single * double) =
10
11     (fiveByte / 2y, fiveShort / 2s, five / 2,
12      fiveLong / 2L, fiveSingle / 2.0f, fiveDouble / 2.0)
13
14 [<FSharkInput>]
15 let value = [|5y; 5s; 5; 5L; 5.0f; 5.0|] : obj array
16
17 [<FSharkOutput>]
```

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

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.

We also test that edge cases, such as dividing floating point numbers by zero, has the correct results (namely `infinity` or `-infinity`.) We also test that `FShark` inlined functions such (see section 5.5.1 and 5.5.2) behaves as expected. The rewritten functions are tested in `FSharkTests/UnitTests/InlinedFuns`

At the moment, the `FShark` test suite (`UnitTests` + `FSharkPreludeTests`) contains 128 unit tests, spread across 248 test cases.

9.5.1. Testing the **FShark** standard library

For the `FShark` standard library `FSharkPrelude`, we supply multiple test suites.

UnitTests/FSharkSOACS

The first test suite is located in `FSharkTests/UnitTests/FSharkSOACS`. Here, we test various SOAC integrations. The SOACs are not tested systematically, but are instead mixed and matched with other `FShark` language features such as lambdas, to see whether they break when used in more complex ways.

In example, we use the test `Filter3`[1] to test whether Zip, Map, Filter and Unzip, anonymous functions and tuple outputs work well together, which they do. The test is shown below.

```
module Filter3
open FSharkPrelude.FSharkPrelude
open FShark.TestTypes.TestTypes

[<FSharkEntry>] // original test uses doubles but my CPU doesn't support f64s
let zip1a (xs1 : int array) (xs2 : bool array) : (bool array * int array) =
    let tmp = Filter (fun (x: (int * bool)) ->
                          let (i,b) = x in b
                      ) (Zip xs1 xs2) in
      Unzip(Map (fun (x: (int*bool))  ->
                  let (i,b) = x in (b,i)
                ) tmp)

[<FSharkInput>]
let value = [|[|0;1;-2;5;42|];[|false;true;true;false;true|]|] : obj array

[<FSharkOutput>]
let outvalue = ([|true;true;true|] , [|1;-2;42|]) : (bool [] * int [])
```

We do not test all of the Map and Unzip functions, as they are essentially the same. For example, we have tested Map and Map2, but as Map3 is designed just like Map2, but with an extra argument (shown below), we have decided that the design difference is too small to warrant an extra test case.

[1] `FSharkTests/UnitTests/FSharkSOACS/Filter/Filter3.fs`

```
let Map2 f aa bb =
    let curry f (a,b) = f a b
    let xs = Zip aa bb
    in Array.map (curry f) xs

let Map3 f aa bb cc =
    let curry f (a,b,c) = f a b c
    let xs = Zip3 aa bb cc
    in Array.map (curry f) xs
```

FSharkPreludeTests

Besides the `FSharkSOACS` tests in the `FSharkTests` folder, the `FSharkPrelude` also comes with a test suite on it's own. We have implemented this test suite to verify that our `FShark` SOACs have the same behaviour as their Futhark counterparts.

This test suite contains 53 unit tests of the SOACs in the `FSharkPrelude`. As opposed to the unit tests in the `FSharkTests`, the unit tests here doesn't compare their results with the Futhark results, but does instead only compare their results with a predefined expected result.

Nevertheless, this suite contains 53 test cases spread across all the SOACs, except for near-identical cases, like in the Map2/Map3 example from before.

9.5.2. Conclusion on **FShark** language correctness

We have written a comprehensive test suite which covers all parts of the `FShark` language. All our 248 test cases passes our tests, which makes us confident that the `FShark` compiler does indeed translate all parts of the `FShark` language correctly to Futhark. More importantly, our test cases tells us that we can trust that an expression written in `FShark` will evaluate to the same result, no matter whether it's executed in `F#` or in Futhark.

We still have one blind spot though; namely that we haven't written many test cases for unsigned integers. These have not been written due to time constraints, but shouldn't be inherently difficult to add to the existing test suite.

9.6. The performance of **FShark** generated GPU kernels

In the following section, we used the `FShark` compiler to compile complex GPU benchmarks written in `FShark`, to valid Futhark source code.

We have then used the `futhark-bench` program to run benchmarks on the `FShark` generated Futhark code, to compare the `FShark`-generated Futhark programs with equivalent handwritten Futhark programs.

The first benchmark tested is the LocVolCalib benchmark.

9.6.1. The **LocVolCalib** benchmark

In figure 9.3 we see the benchmark results of the LocVolCalib benchmark, located in `FSharkTests/Benchmarks/LocVolCalib.fs`.
For the `F#` results in this benchmark, we have used the `FSharkTests` program in the `FShark` repository to first run the compilation, and pass the dataset arguments to the resulting loaded library.

The first two columns represent the compiled `FShark` code being executed as `C#` libraries, both with and without OpenCL support. The third column represents the `FShark` code being executed natively as `F#` code. The three last columns show the benchmark results of the handwritten Futhark version.

Based on the benchmarks, we can make the following conclusions:

1. `FShark` code is significantly faster when compiled to OpenCL than when compiled to sequential `C#` code. The speed increase is $\times 420$, $\times 571$ and $\times 483$ for the small, medium and large dataset respectively.

2. Running `FShark` code natively in `F#` is slower than compiling the `FShark` code to sequential `C#` code. This is not surprising, as the Futhark compiler makes various optimizations like fusing SOAC calls during compilation, whilst `F#` executes the code as it is written.
   The speed increase from native `FShark` to compiled `FShark` is $\times 1.06$, $\times 1.02$ and $\times 1.51$ for the small, medium and large dataset respectively.

   It is surprising that the `F#` version is not that much slower than the compiled `C#` version.

3. The `FShark` version of the LocVolCalib benchmark is significantly slower than the already existing Futhark version (upto $\times 2.5$ slower.) This is strange, as the two versions are almost the same. We are currently not sure why, as the only major difference is the rewriting of Futhark's loop expression into a foldl expression (section 9.4.1). We will have to investigate this.

4. The sequential `C#` version of the benchmark is up to $\times 37\%$ slower than the sequential `C#` version. We do not currently know where precisely where this slow down appears, but investigating the fact is definitely worthwhile. It is our best guess that the speed difference comes from the sequential version's array reading and writing functions, as these are currently call-by-value, instead of call-by-reference.

9.6.2. The **nbody** benchmark

Whereas the LocVolCalib benchmark takes nine scalars as input to run, the `nbody` benchmark takes multiple arrays of inputs, each containing thousands of scalars. We have not implemented a nice way of handling inputs of this size in the `FShark` architecture yet.

However, we can still run accurate benchmarks on the `FShark` version of nbody, by taking the `FShark` program out of the `F#` context.

First, we compile the `FShark` version of nbody to Futhark source code by using the `FShark` compiler. Then we add the benchmark annotations (shown

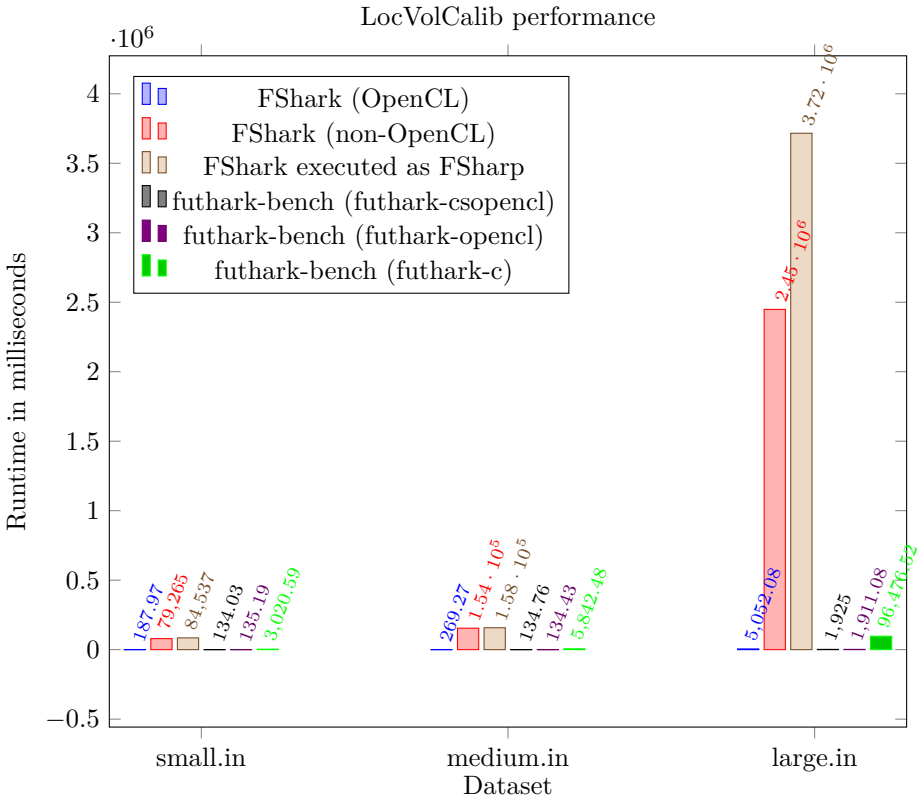


Figure 9.3: Comparison of LocVolCalib benchmark for multiple versions

in figure 9.5) from the Futhark version of nbody, to the `FShark` generated version. Finally, we run the Futhark benchmark program with our code by calling the command:

```
$ futhark-bench --compiler=futhark-csopencl fsharkNbody.fut
```

Unfortunately, the `nbody` benchmark is one of the benchmarks that are affected by the measurement described in section 8.1.2. Therefore we cannot present a representative comparison between the `FShark/C#` GPU kernel, and it's Futhark/`C` counterpart in this report.

However, we can show a runtime comparison between the original Futhark nbody implementation and the translated `FShark` version, by running them both with the Futhark `C` OpenCL compiler. We see the results of this benchmark in figure 9.4.

We see that the `FShark` version compiles to a Futhark program, that has performance very close to the original Futhark program.

9.6.3. Conclusion on performance of **FShark** generated GPU kernels

Based on our two benchmarks, we conclude that `FShark` is indeed a viable GPU programming language. Although we lose a lot of performance in the LocVolCalib benchmark, we see no loss of performance in the nbody benchmark. At this point we cannot safely conclude whether `FShark` generated Futhark programs are inherently inferior to Futhark programs. LocVolCalib says that `FShark` versions are inferior, and nbody says that they aren't, so we will have to write more benchmarks before we can make a conclusion either way.

```
-- ==
-- tags { futhark-opencl futhark-c }
--
-- input @ data/nbody-acc-t0.in
-- output @ data/nbody-acc-t0.out
--
-- input @ data/nbody-acc-t10.in
-- output @ data/nbody-acc-t10.out
--
-- input @ data/100-bodies.in
-- input @ data/1000-bodies.in
-- input @ data/10000-bodies.in
-- input @ data/100000-bodies.in

-- "data/N-bodies.in" all have the other attributes n_steps=1, timestep=1.0, and
-- epsilon=50.0.

-- rest of source code goes here
-- ..
```

Figure 9.5: Annotations like these shows `futhark-bench` where it can find datasets for its benchmarks

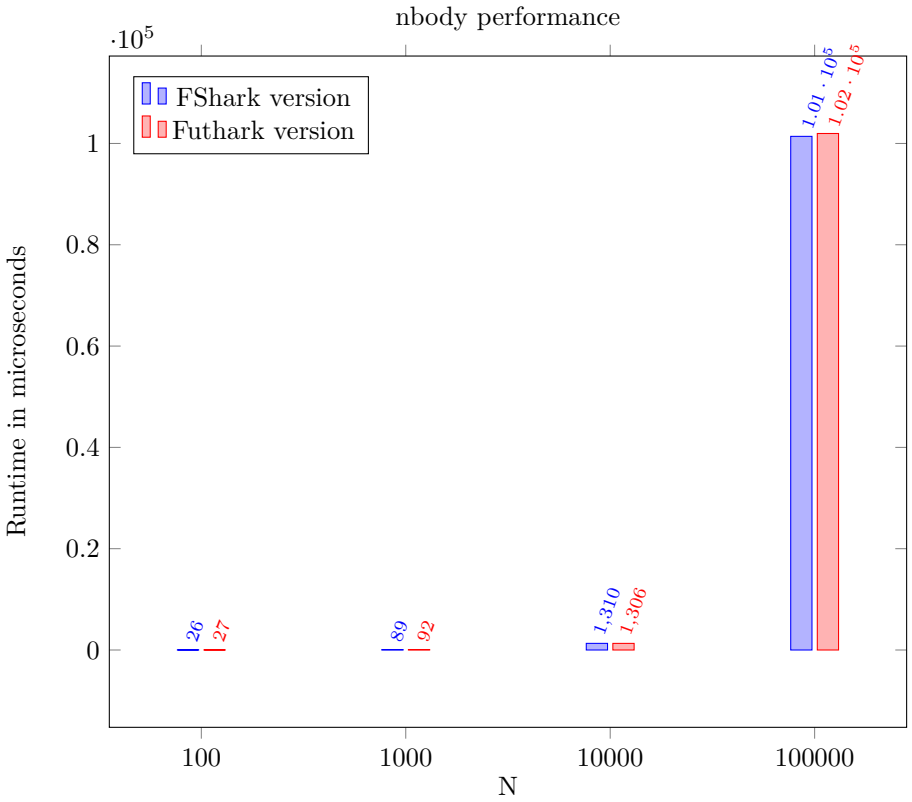


Figure 9.4: Two versions of the nbody benchmark compared

10.0

Conclusion and future work

We have presented a code generator which generates GPU accelerated computational libraries that are readily integrable in both `C#` and `F#` programs. We have also presented a programming language `FShark`, which allows us to prototype, compile and execute GPU kernels in an `F#` environment.

Our benchmarks shows that our `C#` code generator generates GPU kernels which has performance comparable to the GPU kernels generated by the already existing `C` code generator, with average runtimes being within $+0 - 3\%$ for large datasets. However, multiple test cases in the Futhark test suite are still failing. This indicates that the current version of the `C#` code generator can not completely Futhark code to `C#` programs.

We have validated the `FShark` language and it's translations to Futhark on a comprehensive test suite of over 120 unit tests and 190 unique test cases, which covers the entire `F#` language and it's standard library.
Furthermore we have demonstrated that `FShark` is suitable for developing efficient GPU kernels, by porting multiple existing GPU benchmarks to `FShark` – although currently with severe ($\times 2.5$) speed reductions for large datasets in some benchmarks.

Future work

As described in section 8, the current implementations of our code generator and `FShark` suite is limited in several ways. The following prioritized list shows the future work of the `FShark` project.

- The `C#` code generator for Futhark is not finished. Although we can succesfully compile and run a large part of the Futhark benchmark suite, there are still several benchmarks that fails to run on the current implementation. They either fail due to errors in the current implementation, or because they rely on features that have not been implemented yet. We plainly need to identify and fix these issues.
- There are still plenty of SOACs and array functions in the Futhark standard library, that haven't been implemented in `FShark` yet. We want to port more of these functions to `FShark`, which should be relatively easy, as they are just functions, and not language constructs.

- As described in section 6.2.2, chaining GPU kernel calls currently adds an unnecessary overhead. We want to develop an opaque array type which merely contains references to already allocated GPU buffers, and which aren't copied back from GPU to RAM until strictly necessary.
- The current version of `FShark` prohibits certain function types and reserves them for the `FShark` wrapper, as the wrapper needs to handle jagged arrays before using them with `C#` Futhark functions. We are interested in finding a better way of handling this, so we can remove this seemingly arbitrary language restriction.
- Section 9.6.3 shows that the sequential `C#` backend is significantly slower than the sequential `C` backend. We want to determine whether the current implementation contains any easy fixes for better performance. This is not a top priority though, as Futhark is primarily intended for OpenCL use anyhow.
- The current iteration of the `FShark` language lacks several of Futhark's language features. For example, as seen in section 9.4.1, we currently have to emulate `for`−loops by rewriting them into folds instead. It should be possible to add equivalent loop semantics to the `FShark` language, but it does require more time and experiments with the `F#` language and -compiler.
- Currently, we only use modules as nested namespaces in the outer scope of the `FShark` programs. It would be interesting to investigate whether it is feasible to implement higher-modules in `FShark`.
- `FShark` currently uses jagged arrays to emulate multidimensional arrays, even though jagged arrays themselves bring multiple design problems to `FShark`. The primary reason for using the jagged arrays is that they are easy to develop SOACs for. In the future we want to develop a library of SOACS for the multidimensional array type in `F#`, so we eventually will be able to exclude jagged arrays from `FShark` completely.
- The current implementation of the `C#` code generator generates `C#` libraries that takes flat array/integer array pairs as arguments, when multidimensional arrays are needed.
  It is unpleasant for programmers to manually convert their multidimensional (or jagged) arrays before calling Futhark library functions.
  We want to change the entry functions for Futhark `C#` libraries, so they can be called with multidimensional or jagged arrays.
  Alternatively, we can alleviate the problem by making the flattening algorithms from section 5.8 available through the Futhark generated `C#` programs.

Appendices

A.0

Implementation

Code generator

The C# code generator is part of the Futhark project. The Futhark implementation is located on Github: `https://github.com/diku-dk/futhark`

The code generator is stored in the Futhark branch `mknu/csharp`, and this thesis is based on the commit `0d363e9572b1c32299332a30cadfc31d5427817a`, see

`https://github.com/diku-dk/futhark/tree/0d363e9572b1c32299332a30cadfc31d5427817a`

Our implementation is located in the folder `src/Futhark/CodeGen/Backends`.

We have been testing the code generator using the test suite located in the `tests` directory in Futhark's root directory.

FShark language and compiler

The `FShark` language, compiler and test suite is all located on Github: `https://github.com/diku-dk/fshark`. It is structured as a single `F#` solution containing five `F#` projects.
This thesis is based on the commit `cc223733703f20a313fb5fddef51b697619504d3`, see

`https://github.com/diku-dk/fshark/tree/cc223733703f20a313fb5fddef51b697619504d3`

The `FShark` test suite can be found in the `FSharkTests` folder.
An executable example is shown in `Examples/Program.fs` folder. We recommend opening and using the `FShark` solution through an IDE such as Jetbrains' `Rider`.

B.0

FShark standard library

The `FShark` standard library is available in the `FShark` repository in the file `FSharkPrelude/FSharkPrelude.fs`.
See `https://github.com/diku-dk/fshark/blob/cc223733703f20a313fb5fddef51b697619504d3/FSharkPrelude/FSharkPrelude.fs`

C.0

Program for benchmarking byte memory writes in **C#**

```
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)
        {
```

```
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

D.0

LocVolCalib benchmark written in **FShark** and Futhark

To avoid adding hundreds of lines of source code to the appendices, we instead link to the two different versions of the LocVolCalib benchmark:

The `FShark` version is available on:
`https://github.com/diku-dk/fshark/blob/d41e5d99f37dc6c77b565ec89ee58533bb264232/FSharkTests/Benchmarks/LocVolCalib.fs`
The Futhark version used is available on:
`https://github.com/diku-dk/futhark-benchmarks/blob/fd4dec357bb51d8109fed67c2e14bc5da9b20179/finpar/LocVolCalib.fut`

E.0

nbody benchmark written in **FShark** and Futhark

The FShark version is available on
https://github.com/diku-dk/fshark/blob/d41e5d99f37dc6c77b565ec89ee58533bb264232/FSharkTests/Benchmarks/Nbody.fs

The Futhark version used is available on:
https://github.com/diku-dk/futhark-benchmarks/blob/fd4dec357bb51d8109fed67c2e14bc5da9b20179/accelerate/nbody/nbody.fut

Bibliography

[1] C. Andreetta, V. Bégot, J. Berthold, M. Elsman, F. Henglein, T. Henriksen, M.-B. Nordfang, and C. E. Oancea. Finpar: A parallel financial benchmark. ACM Trans. Archit. Code Optim., 13(2):18:1–18:27, June 2016.

[2] L. Bergstrom and J. Reppy. Nested data-parallelism on the GPU. SIGPLAN Not., 47(9):247–258, Sept. 2012.

[3] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a Portable Nested Data-Parallel Language. Journal of parallel and distributed computing, 21(1):4–14, 1994.

[4] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell array codes with multicore GPUs. In Proc. of the sixth workshop on Declarative aspects of multicore programming, pages 3–14. ACM, 2011.

[5] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM.

[6] Y. Chicha, M. Lloyd, C. Oancea, and S. M. Watt. Parametric Polymorphism for Computer Algebra Software Components. In Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Comput., pages 119–130. Mirton Publishing House, 2004.

[7] R. Clifton-Everest, T. L. McDonell, M. M. T. Chakravarty, and G. Keller. Streaming irregular arrays. In Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Haskell 2017, pages 174–185, New York, NY, USA, 2017. ACM.

[8] P. J. Denning and T. G. Lewis. Exponential laws of computing growth. Commun. ACM, 60(1):54–65, Dec. 2016.

[9] M. Dubois, M. Annavaram, and P. Stenstrm. Parallel Computer Organization and Design. Cambridge University Press, New York, NY, USA, 2012.

[10] M. Elsman, T. Henriksen, D. Annenkov, and C. E. Oancea. Static interpretation of higher-order modules in futhark: Functional gpu programming in the large. In Proceedings of the ACM on Programming Languages, Volume 2, Number ICFP, ICFP 2018, New York, NY, USA, 2018. ACM.

[11] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming (JFP), 15(3):353–401, 2005.

[12] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. Int. Journal of Parallel Programming, 34(4):383–427, 2006.

[13] J. Hamilton. Language integration in the common language runtime. SIGPLAN Not., 38(2):19–28, Feb. 2003.

[14] M. R. Hansen and H. Rischel. Functional Programming Using F#. Cambridge University Press, New York, NY, USA, 2013.

[15] A. Hejlsberg, S. Wiltamuth, and P. Golde. C# Language Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[16] T. Henriksen, M. Dybdal, H. Urms, A. S. Kiehn, D. Gavin, H. Abelskov, M. Elsman, and C. Oancea. APL on GPUs: A TAIL from the Past, Scribbled in Futhark. In Procs. of the 5th Int. Workshop on Functional High-Performance Computing, FHPC'16, pages 38–43, New York, NY, USA, 2016. ACM.

[17] T. Henriksen, K. F. Larsen, and C. E. Oancea. Design and GPGPU performance of futhark's redomap construct. In Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY 2016, pages 17–24, New York, NY, USA, 2016. ACM.

[18] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[19] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. Compiling and optimizing java 8 programs for gpu execution. In Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT), PACT '15, pages 419–431, Washington, DC, USA, 2015. IEEE Computer Society.

[20] B. Legrand. Mastering Dyalog APL. Dyalog Limited, November 2009.

[21] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. Maple 9 Advanced Programming Guide. Maplesoft, 2003.

[22] G. Mudalige, M. Giles, J. Thiyagalingam, I. Reguly, C. Bertolli, P. Kelly, and A. Trefethen. Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems. Parallel Comput., 39(11):669–692, Nov. 2013.

[23] C. E. Oancea and S. M. Watt. Domains and expressions: An interface between two approaches to computer algebra. In Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation, ISSAC '05, pages 261–268, New York, NY, USA, 2005. ACM.

[24] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[25] A. Shaikhha, A. Fitzgibbon, S. Peyton Jones, and D. Vytiniotis. Destination-passing style for efficient memory management. In Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing, FHPC 2017, pages 12–23, New York, NY, USA, 2017. ACM.

[26] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, pages 205–217, New York, NY, USA, 2015. ACM.

[27] M. Steuwer, T. Remmelg, and C. Dubach. Lift: A Functional Data-parallel IR for High-performance GPU Code Generation. In Procs. of Int. Symp. on Code Generation and Optimization, CGO'17, pages 74–85, Piscataway, NJ, USA, 2017. IEEE Press.

[28] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans. Embed. Comput. Syst., 13(4s):134:1–134:25, Apr. 2014.

[29] J. Svensson. Obsidian: GPU Kernel Programming in Haskell. PhD thesis, Chalmers University of Technology, 2011.

[30] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures, pages 117–128. ACM, 2011.

[31] E. Totoni, T. A. Anderson, and T. Shpeisman. Hpat: High performance analytics with scripting ease-of-use. In Proceedings of the International Conference on Supercomputing, ICS '17, pages 9:1–9:10, New York, NY, USA, 2017. ACM.

[32] S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach, and R. S. Sutor. AXIOM Library Compiler User Guide. Numerical Algorithms Group (ISBN 1-85206-106-5), 1994.