FuGL

Functional GPU Language

Group DPW108F18

Christian Lundtofte Sørensen Henrik Djernes Thomsen

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg Ø

Title:
FuGL - Functional GPU Language

Theme:
Programming Technology

Project period:
01/02/2018 -
08/06/2018

Project group:
DPW902E17

Members:
Christian Lundtofte Sørensen
Henrik Djernes Thomsen

Supervisor:
Lone Leth Thomsen

No. of Pages: 95
No. of Appendix Pages: 11
Total no. of pages: 106

Abstract:

Developing software that utilizes the GPU often requires using low-level languages, and the developer must be aware of the underlying architecture and execution model, to fully utilize the available resources found on a GPU. This may require the developer to tweak and tune many details in an implementation in order to gain full performance, also requiring a lot of knowledge on the topic.
In this project we introduce *FuGL*, a statically typed functional GPU programming language. *FuGL* features high level abstractions, requiring only a few keywords to parallelize an implementation. *FuGL* uses *LLVM* as backend, as it features architectural support for both CPUs and GPUs.
The *FuGL* compiler is developed using *.Net Core* with *C#*, to support both *Windows*, *macOS* and *Linux*. The compiler implements the *LLVM* C API using *LLVMSharp*, to emit *LLVM Assembly*.
FuGL features a simple syntax with only a few constructs, as with *Lisp*. With only the keywords `mutable` and `gpu`, the developer is able to decide how and when data is allocated and copied to the GPU, to remove unwanted memory copying. Kernels are specified using only the `gpu` keyword.
FuGL has been tested on three developers to determine the programmability, which is satisfying for the current version. Furthermore *FuGL* is performance tested, showing that the performance is comparable to other high level languages. The performance tests are performed by implementing three algorithms in *FuGL* and comparing the performance with other languages for GPU development.

Master Thesis Summary

This report documents the master thesis work performed by group DPW108F18 during the 10th semester of Software Engineering. We describe the design and implementation of *FuGL*; a Functional GPU Language. *FuGL* is a statically typed, high-level functional programming language with primitives for GPU development. The purpose of *FuGL* is to allow developers to utilize the GPU at a higher level than traditionally seen in *CUDA* and *OpenCL*, and maintaining a high level of programmability, as described by Sebesta. We base this project on the assumption that for many developers, programmability is more important than performance.

The work presented in this report is based on the work performed during the 9th semester, in which we analyzed multiple languages and frameworks for GPU development by implementing three algorithms in each language. The algorithms we implement are generating permutations, K-Means, and Radix sort. We ranked the programmability and performance and compared the languages and frameworks on these parameters. We concluded that the language is not the most difficult part of GPU development, but instead figuring out how to parallelize an algorithm and how to make the implementation run well on the GPU. We also found that low-level languages, while faster, are much more difficult to develop in and have lower programmability than the higher-level languages. Furthermore we found that high-level frameworks for GPU development, such as *Numba* for *Python*, require using syntax or elements that are not native to the languages, and thus often do not fit into the languages.

FuGL is designed with a low amount of constructs and keywords, as well as a simple syntax, in order to raise the *simplicity*. The constructs are inspired by *Lisp*, and the syntactical elements are based on *C*-like languages, *Haskell*, and *Swift*. The GPU utilization is based on frameworks such as *Alea GPU* and *Thrust*, and *FuGL* allows developers to use the built-in types on both the CPU and the GPU, but still retain control over memory using keywords to describe how the memory is used.

The *FuGL* compiler is implemented in *C#* and uses *LLVM* as the backend. *C#* was chosen as it is high-level, cross-platform, and allows interfacing with *LLVM* using the *LLVMSharp* framework. Solutions such as *Alea GPU*, *Rust*, and *Julia* use *LLVM* as the backend, and *LLVM* allows compilation to both *CUDA* and *OpenCL* code. The compiler emits a single *LLVM* assembly file, containing both the CPU and GPU code of the program which can then be compiled and executed. The compiler is not complete, and there are many features that are designed but not implemented. There are various issues in the current implementation, the largest being that *FuGL* does not allow many levels of recursion, due to some *LLVM* issues during the design. The recursion issue can be avoided by utilizing `for`-loops.

The level of programmability and the performance of *FuGL* was tested. The performance was tested by implementing the three algorithms from the 9th semester

project in *FuGL* and comparing the execution time to the execution times of the implementations previously developed. The performance of *FuGL* is comparable to other high-level frameworks for GPU development, such as *Numba*, but worse than the low-level languages, as expected. The programmability was tested by performing interviews with three testers, inquiring about the general syntax and layout of the code, as well as the GPU utilization. The testers found *FuGL* to have a high level of readability, and the GPU primitives seemed easy to use and high level compared to the languages that are traditionally used for GPU development.

We are satisfied with the outcome of this project. *FuGL* is simple, with few constructs, and interviews indicate that the level of programmability is higher than traditionally used GPU languages. It is easy to convert sequential *FuGL* code to parallel code that can be executed on the GPU, and tests showed that by just adding the `gpu` keyword, the execution time was halved. There are some issues with the compiler, but due to the timeframe and scale of this project we find this acceptable.

Preface

This report documents the work performed by group DPW108 on the 10th semester master thesis project at the Department of Computer Science at Aalborg University. References to programming languages and frameworks refers to what is available at the time of writing this report. The work is based on the work performed during the 9th semester[1]. The bibliography can be seen at the end of the report.

We would like to thank our supervisor, Lone Leth Thomsen from the Department of Computer Science at Aalborg University for the guidance in this project. The feedback from Lone significantly raised the quality of the project.

The report is best read in sequential order to follow the five phases the report is split into: Analysis, design, implementation, testing, and summary.

The report structure is based on the initial brainstorm performed in this project, which is seen in Figure 1. The major discussions are the type of solution to implement and the backend for the solution. These are discussed throughout the analysis and design. An initial assumption of this project, is that programmability is more important than performance, based on the work performed in [1]. Programmability refers to multiple criteria, such as writability and readability, as defined by *Sebesta*[2] and the GPU criteria found in [1]. This assumption is used as the basis of this project, and we attempt to implement a solution that makes GPU development easier and faster for developers, but possibly at the cost of performance.

Analysis
: The analysis contains the initial thoughts about the project, a consideration of which type of solution to implement, as well as an analysis of various libraries and frameworks for GPU and parallel development.

Design
: The design phase contains the design choices for *FuGL* and the *FuGL* compiler. We describe the syntax and thoughts behind *FuGL*, as well as how the compiler is designed.

Implementation
: The compiler is implemented, and the compiler phases are described.

Testing
: This part contains a description on how testing is performed. We test how well the compiler works, the programmability of *FuGL*, and the performance of *FuGL*.

Summary
: In this part we summarise and finish the project. This includes reflections, conclusion, and future work.

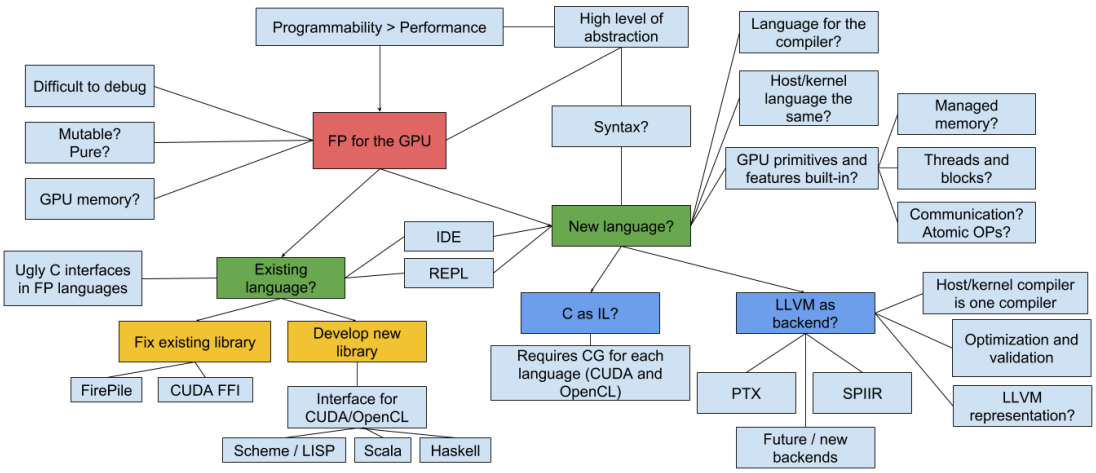

Figure 1: The initial brainstorm for the project, showing the major questions and discussions in the project.

Contents

1. Motivation

In recent years, functional programming has been introduced in many modern and popular languages[3]. These languages, many of them object-oriented, have adapted functional features, to accomplish simple-to-express, yet powerful operations, such as first-class functions[4], higher-order functions[5], anonymous functions[6], and immutable objects[7]. This indicates that the functional paradigm and higher level of abstraction is becoming something that developers want to use.

Another topic gaining popularity in the recent years is GPU programming. As GPUs are often used for gaming or graphical purposes, a GPU may be left idle during general purpose computing, leaving computations to the CPU. For complex computations this might not be desirable, and may take a longer time than what the GPU can accomplish[8]. A modern strategy is therefore to execute parts of an algorithm on the GPU, to increase performance[8]. This is seen in various applications in various fields, such as scientific computing and image manipulation.

Developing software for a GPU requires a different mindset, as the architecture of a GPU differs from a CPU. Where a CPU often has a limited set of cores, a GPU can have thousands of cores[8]. This requires software introducing massive parallelism to utilize each core on a GPU. As we concluced in [1], determining how to parallelize an algorithm is difficult, and often requires massive rewriting of an algorithm in order to utilize the GPU correctly. In conclusion, programming GPU software can be challenging and time consuming.

In our previous project[1] we concluded that functional languages either do not work for GPUs or require an imperative approach. Based on this conclusion we want to build a language that takes these problems into account, and enables GPU programming in a more functional manner. Because we want to build GPU functionality directly into a language, we develop a new language instead of building on top of an existing language.

One of the focus points of [1] is the programmability of languages. We define programmability as readability, writability, and reliability, as defined by *Sebesta*[2]. These criteria are not focused on performance. In [1] we found that low-level languages are better performing than high-level languages for GPU development, but the programmability of these languages are lower than the high-level languages. Furthermore we found that even in high-level languages the developer still has to consider low-level details, such as thread and memory management, in order to utilize the GPU fully. Therefore, the goal of this project is to design and implement a solution that allows developers to utilize the GPU while developing at a higher level of abstraction, such as seen in functional programming languages and modern GPU libraries. The project is built upon the assumption that programmability is more important than performance. This means that we are willing to sacrifice some performance in order to gain programmability and to make it easier to utilize the GPU.

2. Problem Statement

This chapter outlines the problem statement for this project. The problem statement is further split into subtasks that have to be completed during the project. We also describe the development process of this project.

As described in Chapter 1, GPU development is a daunting task, and as we have previously experienced, the level of abstraction when developing for the GPU is very low. In [1] we found that low-level languages are very efficient, but have lower programmability than their high-level counterparts. In an attempt to help developers utilize the GPU at a higher level of abstraction and supporting multiple platforms, we have the following problem statement:

How can we develop a solution that allows developers to write efficient GPU code at a higher level of abstraction than OpenCL or CUDA, while maintaining the possibilities that a low-level language provides?

2.1 Tasks

In order to solve this problem we have to perform a number of tasks, that will help us learn about the problem domain, potential solutions, and problems which should help us fulfil the problem statement.

Analyze Possible Solutions
The first step is to analyze which kinds of solutions we can develop that increase the level of abstraction for GPU development. This includes considering whether the best solution is a library or a language.

Determining a Backend
Determining a backend for the library or compiler is an important task, as it will will impact all parts of the development. The backend can be developed to target *NVIDIA* and *AMD* platforms, or utilize a shared backend such as *LLVM*, that compiles for both platforms.

Designing the Solution
Whether the solution is a library or a language, the developer interface has to be designed. In the case of a library, the APIs that are exposed to the developer have to be designed carefully, to allow ease of use, while maintaining functionality. In the case of a programming language, the grammar and functionality have to be designed. A development language, in which the compiler is implemented, has to be chosen in this task as well.

Implement the Solution
The process of implementing the designed solution with the chosen backend.

Testing the Solution
In order to test the claim that the solution is easy to use and provides a

high level of abstraction, the solution has to be tested. This entails finding developers that have an interest in GPU development, and have knowledge about how GPU development is currently performed, in order to provide feedback on our solution in comparison to the 'traditional' way. Furthermore we test the performance of the solution. Although one of the assumptions of this project is that performance is less important than programmability, it is still interesting to see how the solution compares to traditional GPU development languages. We use the knowledge and performance tests from [1] for this comparison.

2.2 Development Process

The development of this project follows the waterfall model, meaning that we analyze, design, implement, and test in a linear way. Each phase is finished before starting the next. The implementation of the language or library is performed iteratively, meaning that we implement a few features, then reconsider, implement more features and so on. The reason for this choice of process is that we wish to analyze and design the solution, but we may not be able to implement all parts of the solution, and as such, we implement a small number of features in each iteration. The testing is performed at the end.

Part I

Analysis

3. Analysis Introduction

The analysis contains discussions regarding the type of solution to implement in this project, and inspiration for the design and implementation. There are multiple options for the solution that we consider; implement a library, modify a compiler, or develop a new language. The advantages and disadvantages of these approaches, along with our choice, is described in Chapter 4. We then analyze various libraries and languages for parallel and GPU development. We describe how these solutions handle memory and threads, which we have previously seen in [1] is one of the difficult parts of GPU development.

4. Choice of Solution

In this chapter we discuss and choose what type of solution to implement. The options are to develop a library, further develop an existing language and compiler, or to develop a new language. This decision influences how the solution is utilized by end users. All options are viable choices, and each have strengths and weaknesses.

4.1 Library

The first approach is to implement a library that allows developers to write high-level GPU code in a language by using a library. The library should offer high-level abstractions, such as the ones available in *Alea GPU* for *C#*[9]. These abstractions allow `for`-loops to be parallelized, automatic memory management, etc. Other similar implementations are *Firepile*[10] for *Scala* and the *Rust* GPU framework[11] for *Rust*.

Other libraries offer varying levels of abstraction for GPU development, but they all have the same problem; they are added to a project and a language, and not an integral part of it. This means that developers will use only what the developers of the library expose, and it might not be possible to extend the library for more use cases.

Libraries exist that solve many types of problems for GPU development, such as *cuBLAS*[12] for vector and matrix operations, but many of these libraries are for very specific operations. The *Thrust*[13] library is more general-purpose oriented, but still only expose the functionality that the developers of the library wish to expose. Many of these libraries will only target one platform, such as *CUDA* or *OpenCL*, which can make it unsuitable for cross-platform development.

One of the conclusions we found in [1] is that functional programming languages are not, at the time of writing, well suited for GPU development, and the libraries that can be used in functional programming languages are immature, difficult to install, or require the developer to write imperative code instead of functional code. The advantage of a library is, that it is based on an existing compiler infrastructure, which allows developers to focus on implementing the functionality of the library, without having to consider how the compiler works.

Other issues with this approach is that the framework has to update with the language, and changes in the compiler can break the framework. This is the case with both *Firepile* and the *Rust* GPU implementation. A few years after development the frameworks are no longer usable, as the languages or runtimes have changed since their implementation.

4.2 Modify a Compiler

The next possible solution is to modify or add phases to an existing compiler. We have previously seen this approach used in [14], where a new phase was added to the *Julia* compiler that allows run-time compilation and execution of GPU code. In the *Julia* library, the framework performs a specialized *CUDA* GPU compilation, using interfaces in the main compiler, when functions are called with a `@cuda` keyword. This approach allows more control of how the compiler handles the GPU code, as the project directly interacts with the compiler.

While language changes can break this implementation, the risk of such is greatly reduced, as the library relies mostly on the main compiler. As long as the changes do not affect the compiler interfaces, or introduce functionality not supported on the GPU, the library will not break on language changes. Modifying a compiler requires the developer to follow the development of the compiler.

4.3 New Language

The third possibility is to design and implement a new language, which features built-in support for GPU development, as well as a high level of abstraction. This approach allows full control of how the GPU is handled, and how the host code is implemented. This approach allows us to develop a language that has the high level of abstraction that we are looking for, and the amount of low-level control that GPU development requires.

The main disadvantage of this approach is the massive amount of work that implementing a compiler requires. The language has to be designed, a syntax has to be made, and an implementation has to be made including testing. A suitable intermediate language will have to be chosen, which should work on multiple platforms, to support both *CUDA* and *OpenCL*.

This approach requires developers to learn a new language, which might discourage developers looking to do GPU development. This approach will furthermore allow us to implement a functional programming language with the features needed for GPU development, which we have seen in [1] does not currently exist.

4.4 Summary

The choice of solution to implement is important, and this chapter provided three options; a library, a language or compiler extension, or a new language and compiler. We described advantages and disadvantages for each of the approaches. Despite the large amount of work that it will require, we choose to implement a new programming language, with high-level abstractions and built-in GPU primitives. This language is called *FuGL*, meaning *Functional GPU Language*.

As previously stated, functional programming is not yet a viable choice for GPU development, despite many of the functional programming languages being built for array and number operations, which the GPU is well suited for. Therefore it is interesting to explore how a functional programming language can be implemented to work well for GPU development. In our opinion, neither a library or a compiler extension will make the GPU development appear built-in, which is one of our priorities. Furthermore we will not rely on other developers of a language potentially breaking our library or extension, which we have previously seen[10] [11].

5. State of the Art

In this chapter we analyze some existing high-level languages and libraries for parallel and GPU development before designing and implementing a new language. We consider libraries and languages that are general purpose, and not libraries such as *cuBLAS* which are aimed at one specific type of usage. We cover multiple types of solutions, both frameworks, libraries, and compiler extensions. The libraries are for both low-level and high-level languages. The list of libraries we examine is not exhaustive, but we have chosen the solutions based on our work on [1], knowledge from a previous course, and some interesting solutions found while analysing the GPU development field.

For each of the solutions, we analyze how memory is handled, including allocations, deallocations, memory transfers to and from the GPU, and how they handle block and thread dimensions. We focus on these features as they are what make the libraries differ from low-level GPU development, and as seen in [1], these parts are difficult in GPU development. We further consider whether these solutions rely on compile-time or run-time code generation for GPU code. These solutions are used to aid in the design of the language.

5.1 Julia

In December 2017, the *Julia*[15] language introduced interfaces for extending the main compiler and potentially targeting new platforms. An example of such an implementation can be seen in [14], where a JIT compiler for the *CUDA* platform is implemented. Both the GPU functionality and compiler structure is very interesting in the context of our project.

Julia is a dynamically typed language utilizing type inference and aggressive specialization on runtime variables, to infer dynamic types into a much more statically typed *Julia* Intermediate Representation(IR). *Julia* IR, which is "mostly statically-typed"[14], and "is a good fit for the *LLVM* compiler framework"[14] which is fully statically typed, "commonly used as a basis for industrialstrength compilers"[14], and furthermore features support for several architectures including *CUDA*.

As seen on Figure 5.1, the main Julia compiler provides a set of interfaces, highlighted with the numbers 1 to 5, so that a specific compiler phase can be invoked, or an IR can be modified. In addition, specific hooks or parameters can be set to restrict specific functionality. For example, exceptions or dynamic memory is a bad fit[14] on a GPU, and as such, exceptions can be restricted through the compiler interfaces.

In the *CUDA* library for *Julia*, kernels can be programmed just as in *CUDA*. Most of the low-level operations are performed automatically leaving out the need of explicit memory allocation and copying. Kernel functions are not marked with

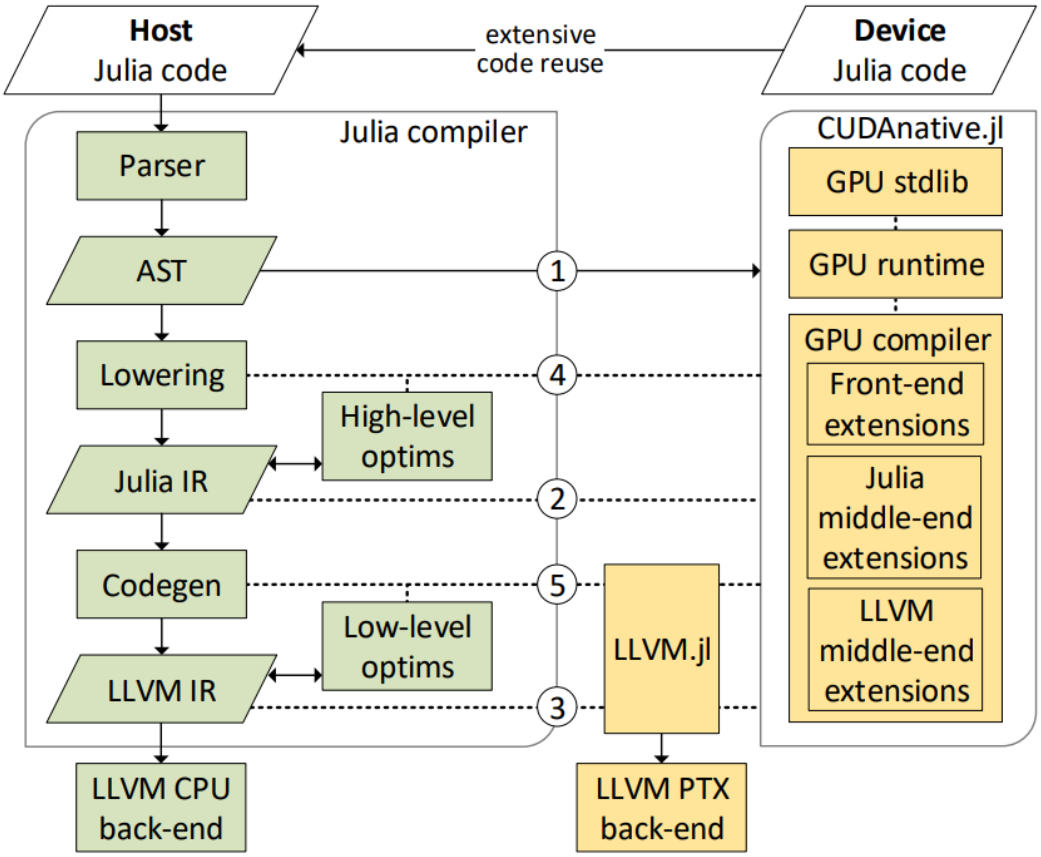

Figure 5.1: The structure of the main *Julia* compiler and GPU *Julia* compiler[14]. The numbers 1 to 5 highlight the compiler interfaces.

anything hence acting like host functions, but to work on a GPU usually a global or local block id must be retrieved. This is done using a `blockIdx`, `blockDim` or `threadIdx` function which can only be used inside kernels. Marking a function as a kernel function is done through the function call, where a `@cuda` must be prefixed.

CUDA arrays can be allocated in a more abstract way using a `CUDAAdrv.Array` function. Functions can be broadcasted to these arrays, which will make the GPU execute the function on each element in the array in parallel, leaving out the need for a kernel function.

A problem in *Julia* GPU is the compile time, upon first execution of a kernel. The compile time for a kernel is measured in [14] as being between 2 and 6 seconds, depending on the complexity of the kernel. As this compile time is several times higher than the typical kernel execution time experienced in [1], ranging from 0.35 seconds to 7.98 seconds, this is devastating for the overall performance.

5.2 Rust

Rust[16] is another language where GPU programming can be achieved. As seen in [11], a separate compiler is developed that allows *NVIDIA* GPU programming. Like in *Julia*, both the main and kernel compiler uses *LLVM* as compiler backend.

Unlike the *Julia* compiler, the *Rust* compiler does not provide any interfaces in the compile process. The kernel compiler and main compiler are separated, as seen on Figure 5.2, and as such the kernel compiler duplicates some functionality from the main compiler. This also means that if even minor changes are made to the language, the kernel compiler must be updated with the same changes.

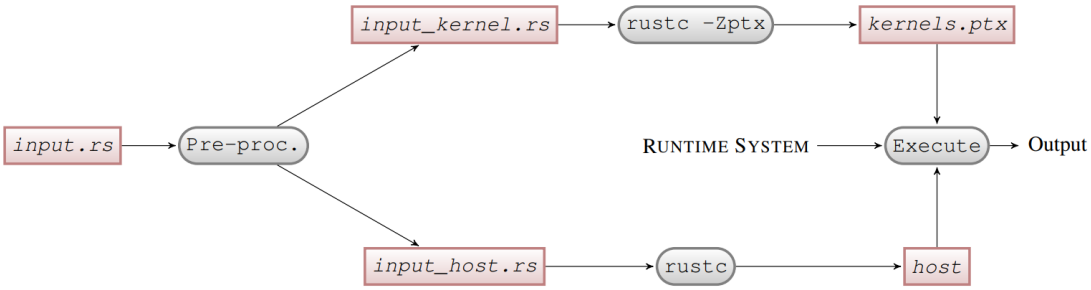

Figure 5.2: The structure of the main and kernel *Rust* compiler[11].

In the *Rust* language, the keyword `mutable` exists. This keyword marks that a variable can be modified, which is convenient in a functional language for GPU development, as mutable values are often not allowed in functional languages, as seen in *Haskell* and *Lisp*. Because kernels will often modify the contents of an array, immutable values are problematic in GPU development.

The abstractions of *Rust GPU framework* are in the form of built-in functions to be called from kernels. No GPU abstractions can be invoked or called directly from a host function. In our opinion this limits what can be accomplished by the *Rust GPU framework*, which could be improved.

To execute the compiled kernels, the *Rust GPU framework* utilizes *OpenCL* bindings. The reason for this is not explained in [11], but they point out some challenges in using these bindings. As *FuGL* is supposed to target multiple platforms, using *OpenCL* bindings to start kernels is a possible choice.

5.3 Accelerate

Accelerate[17] is a *Haskell*[18] library for GPU development, that uses run-time code generation for the GPU code. It uses *LLVM* as the backend for GPU code generation. As described in [1], we were unable to install *Accelerate*. The documentation for *Accelerate*[17] describes how the framework is used, which we will base the analysis on. As *Accelerate* is a high-level library for a high-level language, it is interesting to analyze as it attempts to solve the same issues as *FuGL*; offering high-level abstractions in a high-level language for GPU development.

Based on the examples from the *Accelerate* website[19], we see that *Accelerate* allows normal list functions, such as `fold` and `map`, but requires that the lists are of type `Acc`. Listing 5.1 shows a dot product function using *Accelerate* that can be run on the GPU. This code is best read from right to left. The parameters for the function indicate that the input must be floating point vectors of the type `Acc`. First the code multiplies the two vectors, `xs` and `ys`, using `zipWith` and the multiply function. Then the multiplied vector is 'folded' using the plus operator starting from 0, meaning that all the values are summed.

```
1 import Data.Array.Accelerate as A
2
3 dotp :: Acc (Vector Float) -> Acc (Vector Float) -> Acc (Scalar Float)
4 dotp xs ys = A.fold (+) 0 (A.zipWith (*) xs ys)
```

Listing 5.1: Dot product in *Haskell* using *Accelerate*.

Accelerate uses regular *Haskell* syntax, and the library contains many standard library list functions, making it easy to integrate into existing *Haskell* applications. Some special types are used and there are constraints on how these types can be used on the GPU. For example, `Acc` lists can not be appended or created, which is a GPU limitation[20]. An `Acc` list can be created from a 'regular' *Haskell* list by using a keyword, and the `Acc` keywords can be reused for GPU computations without being allocated on the GPU again.

5.4 Futhark

Futhark[21] is a statically typed, pure functional programming language designed for GPGPU development created by researchers at University of Copenhagen. *Futhark* is intended to help in developing the computing intensive parts of a program, and then use the generated code from *Futhark* in other applications. It is not intended to replace general-purpose languages[21]. *Futhark* generates *OpenCL* code when compiled, which can be integrated into other frameworks,

such as *PyOpenCL* or used in a regular *OpenCL* application. According to the *Futhark* developers, *Futhark* produces GPU code that is fast[22].

Futhark allows common functional operations, such as `map` and `reduce` to be executed on the GPU, with a high level of abstraction for developers. An example of a *Futhark* program can be seen in Listing 5.2. This function computes the factorial of a given integer `n`, by reducing the numbers using a multiplication function from 1 to `n`.

```
1 let fact (n: i32): i32 = reduce (*) 1 (1...n)
```

Listing 5.2: Factorial implemented in *Futhark*.

5.5 Alea GPU

Alea GPU[9] is a library for GPU development in *C#*[23] and *F#*[24]. It uses *LLVM* as the backend for GPU code generation. In [1] we used *Alea GPU* in *F#*, where we found *Alea GPU* to have a high level of programmability and good performance, but the framework required the developer to write imperative code in a functional language. While this is unfortunate, it shows that it is possible to utilize the GPU in functional programming languages.

When using *Alea GPU* for *C#*, the level of abstraction is actually higher than when using the library in *F#*. The *C#* version of the library allows automatic memory handling whereas in *F#*, memory has to be allocated and deallocated explicitly.

Listing 5.3 shows the signature of a for-loop that is executed on the GPU in *Alea GPU* for *C#*. This function takes a start index, end index, and a lambda that describes the action for each iteration. Other than the `for`-function, there are functions for multiple types of aggregates. Besides using these high-level abstractions, the developer can access memory and thread block dimensions. Memory allocated on the GPU can be accessed using *.NET* types instead of pointers, allowing developers to perform the same actions on both the CPU and GPU objects.

```
1 void Gpu.For(int start, int end, Action<int> op);
```

Listing 5.3: For-loop in *Alea GPU* in *C#*

5.6 Thrust

Thrust[13] is a general purpose GPU library for *C++*. It adds a number of abstractions that allows the developer to perform actions without considering memory, and thread / block dimensions. In *Thrust*, once a `thrust::device_vector` is constructed with a vector, the memory required is allocated on the GPU and the array is copied. This variable can then be used for multiple function calls, such as `thrust::sort` or `thrust::reduce`, which performs these actions on the GPU. Once the actions have been performed, the array can be copied back into a `host_vector` using the `thrust::copy` function.

```
1  // Generate random numbers
2  thrust::host_vector<int> h_vec(32 << 20);
3  std::generate(h_vec.begin(), h_vec.end(), rand);
4
5  // Transfer to GPU
6  thrust::device_vector<int> d_vec = h_vec;
7
8  // Sort data on the GPU
9  thrust::sort(d_vec.begin(), d_vec.end());
10
11 // Transfer back to host
12 thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```

Listing 5.4: Sorting an array of numbers in *Thrust*. Taken from [25].

An example of using *Thrust* is seen in Listing 5.4. This listing shows that a `host_vector` is filled with random data. Then a new vector of type `device_vector` is initialized with the `host_vector`, which automatically allocates and moves the content to the GPU. A function then sorts the content, and the sorted array is then copied back to the host.

5.7 Sisal

Sisal is a statically typed functional programming language made for parallel programming[26]. *Sisal* is based on mathematic principles and safety, and the compiler ensures that all *Sisal* programs are deterministic on all platforms and environments and levels of parallelity. As with many other functional programming languages, everything in *Sisal* is an expression. This makes it possible to use, for example, an if-expression as a parameter to a function. *Sisal* allows returning multiple values from expressions and functions.

Parallelity in *Sisal* is achieved using loop constructs, where each computation is independent. A loop consists of a range, an optional body, and a return value[27]. Loops return values, as they are expressions. The body of the loop is optional,

but can be used to calculate and set values for the return clause. An example of a loop in *Sisal* is seen in Listing 5.5. This loop creates a new value each iteration, `new_count`, which is the value of `counters` and `increment` added, based on the current index in the loop. The `new_count` array is returned after the loops has finished.

```
1 for i in 1, num_counters
2    new_count := counters[i] + increment[i]
3 returns array of new_count
4 end for
```

Listing 5.5: Loop in *Sisal*. From [27].

This looping construct and the usage of it, where each calculation is independent, matches well with how GPU kernels are often implemented. Dependencies in calculations on the GPU makes operations slow, and might not be possible due to the lack of global locks on the GPU. The promise of deterministic parallel code in *Sisal* is also interesting, and could make development faster for GPU code, as programs that are 'wrong' are simply not compiled.

5.8 Summary

In this chapter we have analyzed a number libraries and languages with a high level of abstraction for parallel and GPU development. These solutions are used in the design of the language, when designing how developers utilize the GPU. We are confident that the libraries analyzed are representative of high-level GPU development, even though the list is not exhaustive.

Part II

Design

6. Design Introduction

In this part we introduce the *FuGL* language, including the design choices and syntax and the impact on programmability. We first describe how *FuGL* is designed and used, along with which languages that are used for inspiration, the delimitations, and finally the EBNF. This serves as an introduction to *FuGL*.

We then consider the compiler architecture, which includes the backend used in the compiler, the compiler development language, and the phases of the compiler. We then describe the backend *LLVM*, and how this is utilized in the compiler. How the GPU is utilized in *FuGL* is then described. These design choices are based on the analysis performed in Chapter 5.

It should be noted that this part of the report describes many design choices, and not all of these are present in the final implementation of *FuGL*. In the summary of the part, we describe which considerations and design choices that are delimited from the implementation, and why they are not present in the implementation.

7. Language Design

This chapter contains the design choices taken in regards to *FuGL* and presents the *FuGL* syntax. We describe the reasoning for these choices, which languages or frameworks inspired them, and the impact the choices have on programmability. As described in Chapter 1, an important part of *FuGL* is the high level of readability and writability, as defined by *Sebesta*[2]. This means that *FuGL* should be easy to understand, have a low number of concepts and constructs, and it should be easy to get started using *FuGL*. The syntax and constructs in *FuGL* are inspired by many other languages, such as *Haskell*[18], *Swift*[28], and *Lisp*[29], but also frameworks for GPU development as described in Chapter 5. During this chapter different language criteria will be presented, which are described by *Sebesta*[2] or the GPU criteria from [1].

7.1 Syntax

Instead of starting from scratch with the syntax we decide to look at the *Expressions* language[30], which is a hobby project by one of the authors. We do not follow the syntax exactly, but it is used for inspiration. *Expressions* is a statically typed functional programming language. The use of compile-time type-checks increases the *reliability* of the language, and the low amount of constructs in the language increases the *simplicity*. In the following sections the syntactic elements are described.

Basic Syntax

The general structure of *FuGL* syntax is based mainly *C* and *Lisp*, but with elements from *Haskell* and *Swift*. The syntactic constructs such as blocks and function calls, are taken from the family of *C* languages.

Blocks are started with curly brackets and function calls and parameters are started with parentheses. Blocks can contain one or more expressions, where the result of the last expression will be returned. Most blocks should only contain a single expression, but multiple are allowed and necessary for performing IO actions, such as printing values. It is not always necessary to return a value from a block, as `Void` is allowed as a return type. Comments match the *C* syntax, allowing both single- and multiline comments.

The syntax for function- and lambda declarations and their parameters, as well as function types are taken from *Swift*. Many of the constructs in *FuGL* are taken from *Lisp*, due to its simplicity. The different constructs in *FuGL* are functions, lambdas, if-expressions, let-expressions, arrays, and we also add the ability to create records. The low amount of built-in constructs and reserved keywords is intended to raise the *simplicity* of the language.

Values are, by default, immutable in *FuGL*. In order to make a variable mutable, the `mutable` keyword is used when declaring the type of the variable. This design choice is taken from *F#* and *Rust*, which were described in Chapter 5. We decide on allowing mutable values even though it is against what normal functional programming languages allow. Mutable values are necessary on the GPU to modify lists, which we have seen previously is something that is often done in kernels.

We take the syntax for arrays and generic types from *Haskell*. This means that arrays are denoted by square brackets, both array types and array literals. Types that are not known by the compiler we assume to be a generic type. This means that a function that takes parameters which are not one of the default types, or a user defined record, is assumed to be generic.

Functions

Functions are defined using a *Swift*-like syntax. A keyword is used to start the function declaration, then a number of parameters, containing a type and a name, and then finally the return type is defined. All types can be used as parameters and return types. An example of a function declaration is seen in Listing 7.1. This function takes two integer parameters, and returns an integer.

```
1 func myFunction (Int a, Int b -> Int) {
2   // Code here
3 }
```

Listing 7.1: Function declaration in *FuGL*.

Function calls in *FuGL* are much like function calls in *C*. An example is seen in Listing 7.2. This code calls the previously declared function with two integer arguments.

```
1 myFunction(1, 2)
```

Listing 7.2: Function call in *FuGL*.

FuGL contains standard functions for performing actions on arrays and numbers, as well as some miscellaneous helpers. A complete list of standard functions is found in Appendix B.

Types

There are few builtin types, and these are taken from *C*-like languages. The types are `Bool`, `Char`, `Int`, `Int64`, `Float`, `Double`, and `String`. A string should be con-

sidered an array of characters, and can be used in functions that accept arrays, such as `map`.

When used in an expression, a value of one type can be casted implicitly to a type higher in the hierarchy. In order to downcast a type, an explicit cast must be used. Functions that allow casting to each of the builtin types can be used, such as `toInt('a')`, which returns an integer containing 97, the ASCII value of `a`. The type hierarchy is as such:

`Bool` < `Char` < `Int` < `Int64` < `Float` < `Double`.

Strings are not in the type hiearchy, as they should be considered an array of characters instead of a special type.

Records

Records in *FuGL* behave much like structs in *C*. We take the syntax from the *struct* syntax in *C*. Records can not be mutated. They allow all types of variables, including function types. To create a record a constructor is used, which must contain all the values to fill the struct with. An example of a record declaration and creation of a record is seen in Listing 7.3.

```
1 // Record decinition
2 type MyType {
3   Int a,
4   Bool b,
5   Char c
6 }
7
8 // Record initialization
9 MyType(1, false, 'c')
```

Listing 7.3: Declaration and initialization of a record.

Generic Types

Generic types is a feature that allows any type to be used in a function or construct. It is widely used in functional programming languages that works on arrays, as arrays can often contain all types. In *FuGL*, generics can be used for parameters and return values in functions, which can then be called with any type. An example of a function that is best implemented using generics is the `map` function. Without generics, there would have to be a `map` implementation for each type, even developer defined structs, which would require much development time. With generics, one `map` implementation can be used on all types, meaning that it does not have to be rewritten for new types or records thus saving developer time. Using generics, the definition of `map` can be seen in Listing

7.4. In this example, the types `A` and `B` are generic, and any type can be used for these.

```
1 map([A] lst, (A -> B) fnc -> [B])
```

Listing 7.4: Definition of `map` in *FuGL*.

Three approaches are considered for implementing generics:

1. Generate functions for each known type when a generic function is defined.
2. Let the developer tell the compiler which types are used in a generic function and generate functions for the defined types.
3. Analyze parameters and generate functions based on the parameters provided to a generic function.

The first option, which is to generate a function for each known type, for each generic function, is easy to implement and understand. The large drawback with this approach is the massive amount of code generated. *FuGL* has 7 built-in types, and for every record a developer defines, a new function has to be generated. Furthermore, if multiple generics are used in a function, there has to be generated one with each type being one of the known types. This leads to many functions, and most likely many unused functions, and as such this approach is not viable.

The next approach is based on how *C++* handles templates. This approach would have developers define which type is used in a generic type, making it easy for the compiler to know which types are used, and generate functions for each of the used types. *Simplicity* and *writability* are important in *FuGL* , and therefore we do not use this approach. This would require developers to know more syntax and write more code, which we want to avoid.

The third approach is to analyze which types are used when calling a generic function, and generate a function that works on the provided types. This approach is similar to how *Haskell* allows generic function calls. This would have the same advantage as the second approach, that it only generates the necessary functions. Furthermore it does not require developers to declare which types are used, as the compiler deduces this, making it simpler for the developer.

Expressions

As with many other functional programming languages, the majority of the constructs in *FuGL* are expressions, meaning that they must return a value when evaluated. As stated previously, the type of expressions in *FuGL* closely matches those of *Lisp*. A list containing the types of expressions is seen below.

If-elseif-else

The `if-elseif-else`-expression is a standard if expression known from many languages. It must contain an `if` part, can contain multiple `elseif` parts, and must contain an `else` part, as to make sure that it does return a value. An example of an `if-elseif-else`-expression is seen in Listing 7.5.

```
1 if      a > b   { 1 }
2 elseif  a < b   { -1 }
3 else            { 0 }
```

Listing 7.5: `If-elseif-else` expression.

Let

The `let`-expression allows developers to declare variables by type and name, which can then be used inside the block of the expression. Multiple variables can be declared, and modifiers, such as `mutable`, can be used on the types. An example of a `let`-expression is seen in Listing 7.6.

```
1 let Int a = 1, Bool b = true, Char c = 'a' {
2   // a, b, and c are defined and usable here
3 }
```

Listing 7.6: Example of let-expression.

As seen on Listing 7.7, a `let`-expression defines its own block, where a declared variable is accessible with a given value. If the mutable variable is redefined in a new let expression, the variable will have the newly given value.

```
1 let mutable Int a = 2 {
2   //Variable a is 2
3   let a = 10 {
4     //Variable a is 10
5   }
6   //Variable a is 10
7 }
```

Listing 7.7: Scope rule example.

Lambdas

Lambdas, or anonymous functions, are a way of defining a function without a name. These can be used instead of defining functions with a name. They can take arguments and return values, just like a regular function. An example of a lambda-expression is seen in Listing 7.8. In the example, the `map` function is called, which applies a function to every member in an array. The lambda takes an integer, `a`, as parameter, adds one to the

integer, and returns the integer. The result of the call to `map` is a new array containing `[2, 3, 4, 5]`. The syntax of lambdas match the syntax of functions and function types, increasing the *orthogonality*.

```
1 map([1, 2, 3, 4] func (Int a -> Int) { a + 1 })
```

Listing 7.8: Example of using a lambda as a parameter to `map`.

Literals

There are multiple types of literals; booleans, strings, characters, numbers, and arrays. Boolean literals can be `true` or `false`. Numbers can be either integer or floating point numbers. Strings can contain multiple characters. The character type can contain a single character. Array literals can contain any type of expression, as long as they have the correct type. Examples of literals are seen in Listing 7.9.

```
1 1 // Int
2 13.37 // Float
3 "string literal!" // String, which is [Char]
4 'c' // Char
5 '\n' // Also Char
6 true // Bool
7 false // Bool
8 [1, 2, 3, 4] // [Int]
```

Listing 7.9: Example literals.

Function calls

A call to a function, as seen in Listing 7.2.

Binary operations

Binary operations, such as `1 + 1`.

Unary operations

Unary operations, such as `-9` or `!boolValue`.

Variables

Named variables are expressions. These can come from the parameters of a function, lambda, or declared in a `let`-expression.

These expressions can be used anywhere an expression is expected. For example, an `if-elseif-else`-expression can be used as an argument to a function, or in another *if-elseif-else*-expression.

GPU

GPU usage is built into *FuGL*, and it should be easy to use the GPU for developers. Inspired by *Julia*, we decide to have a keyword that defines which code

is run on the GPU. In *Julia*, the keyword is applied on the function call, whereas in *FuGL* the keyword is applied both on the function declaration and function call. The `gpu` keyword on the function declaration makes the function compile to GPU code instead of host code, and also handles memory and threads for the GPU function. The same types that are used on the host code can be used on the GPU, but there are some restrictions. For example, an array can be appended on the CPU, but this is not possible on the GPU. The GPU requires all memory to be allocated before execution, which makes it difficult to append or remove objects from arrays. Stack-based arrays can be allocated on the GPU, but this requires that the size is known on compile-time. Stack-based arrays can not be appended or have elements removed. Though this limitation lowers *orthogonality* in the language, it is a necessary restriction. The technical details about how the GPU is utilized is described in Chapter 9.

An example of vector addition in *FuGL* that is run on the GPU is seen in Listing 7.10. The function is declared using the `gpu` keyword, meaning that the function is compiled for GPU usage, and the thread and block dimensions are implicitly declared inside the function. The global thread ID is precalculated and can be accessed using `gpu.threadID`. The body of the let modifies the `c` array, setting the element at index `tid` to the sum of `a` and `b` at index `tid`.

```
1 gpu func vectorAddition ([Int] a, [Int] b, mutable [Int] c -> Void) {
2   set(c, gpu.threadID, get(a, gpu.threadID) + get(b, gpu.threadID))
3 }
```

Listing 7.10: Vector addition in *FuGL*.

To start a kernel, the `gpu` keyword is used again. The different meanings of the `gpu` keyword based on context somewhat lowers *orthogonality* and *simplicity*, but at the same time makes it clear what is executed on the GPU. An example of starting a kernel is seen on Listing 7.11. The first action that is performed is a call to `gpu` with the number of threads as parameter, which in the particular example is the length of `A`. A GPU object can be saved in a `let`-expression and used for multiple kernels that needs the same number of threads, instead of creating a new object for each kernel.

The arrays used in Listing 7.11 each have different behaviour, because they are defined with different keywords. `A` is defined with the `gpu` keyword, which implies that the array remains on the GPU. To be accessible on the host, it must be copied into a host array in a let assignment. The `B` array does not have any modifiers. This defines a behaviour, where the array is only copied to the GPU when required in a kernel, but not copied back, as the array is not marked as mutable. This leads to the behaviour of array `C`, which is marked as mutable, meaning that the array is both copied to the GPU before kernel execution and from the GPU after the execution of the kernel.

```
1 let gpu [Int] A = [1, 2], [Int] B = [3, 4], mutable [Int] C = [0, 0] {
2   gpu(length(A)).vecAdd(A, B, C)
3 }
```

Listing 7.11: Starting a kernel.

The functions `get`, `set`, and `length` are used on the arrays. These functions fetch or change a value in an array or returns the number of elements in an array. The `set` function is only allowed on arrays that are marked mutable. These functions are usable on both the CPU and the GPU.

7.2 Delimitation

Due to the time constraints of this project, there are some features that we do not include. These are features that we deem not needed for GPU development in *FuGL*, and are too time consuming considering design and implementation effort.

Pattern matching, which exists in *Haskell* and *F#*, is not featured in *FuGL*. This feature could be designed to work as in *F#* where it is a construct that can be used if needed, instead of it being a requirement as in *Haskell*. We do not see this functionality as necessary, and as such, it is not designed or implemented.

Partial application, meaning to create a function where not all parameters are filled in but can be at a later stage[31], is another feature that is not implemented in *FuGL*. This functionality is useful in functional programming languages where first class functions are often used. *FuGL* does not contain this feature due to the amount of work it takes to design and implement this functionality, mainly due to the code generation phase. Partial application can be designed in multiple ways, such as either saving the provided arguments and use them when calling the new function, or generating a new function with the provided parameters already declared in the body of the function. Some initial research shows that this feature is not easy to recreate in *LLVM*, and as such we have decided not to implement it in this version of *FuGL*. We argue that this functionality is not strictly needed on the GPU and therefore is not necessary.

Type classes, or interfaces, on types, such as seen in *Rust* and *Haskell* is a feature that allows developers to implement some functionality on a type, which in turn makes the type usable in functions that requires this trait to exist. Some simple examples are `Eq`(Equal) and `Ord`(Ordering), which allows objects to be compared and sorted. This feature is not implemented, as we do not see the need for it currently.

7.3 EBNF

The EBNF for *FuGL* can be seen in Appendix A. These rules describe the syntax of the language, and which constructs can be used in which locations. The scanner and parser implementation follows these rules closely, and a node type is created for each of the rules, making the abstract syntax tree (AST) follow the grammar. The grammar is an LL(1) grammar[32], meaning that the compiler can decide on the next rule to follow based on the next token found, and no further lookahead is necessary. Due to the *simplicity* of the language, it is only necessary with one token lookahead.

The class of expressions and operators is large. Whether the expressions or usage of the operators are valid, is covered in the validation phase, and not the scanner and parser phase. The grammar does not describe the operator precedence in order to keep it simple, but the parser does handle the precedence. We follow the operator precedence rules from *C++*[33], where applicable. Lambda nodes are declared as an expression, even though they do not return a value directly. They are the only expression type that do not return a value, and as such, is a special case. As they are declared as variables, and can be used in the same places as other expressions, they are defined as an expression.

8. Compiler Architecture

This chapter contains a description of the choices taken in regards to the compiler development. This includes the language that is used for implementing the compiler, the intermediate representation(IR) that the compiler generates and the compiler backend. Finally the phases of the compiler are explained. An overview of the compiler can be seen in Figure 8.1.

8.1 Choice of Backend

Regarding the choice of backend and IR, we consider multiple possibilities: *CUDA* and *OpenCL* as IR, *PTX* and *SPIR* as IR, or *LLVM* as backend. We want a backend that can work on multiple GPU platforms, which these options allow. These platforms also optimize and perform some validation of the code, which could make the compiler development faster. Due to the disadvantages that emitting two types of *C* or platform specific intermediate code has, we will use the *LLVM* backend for the *FuGL* compiler.

LLVM is used in *Julia*[14], *Rust*[11], *Alea GPU*[9] and various other libraries and compilers. *LLVM* is a complete backend for compiler development, it is cross-platform, and can compile an IR to native code for many platforms, including *CUDA*[34] and *OpenCL*[35]. Multiple options exists for using *LLVM* as backend[36]: Use the *LLVM C Library* to generate *LLVM* IR, emit *LLVM Assembly* from the compiler, or emit *LLVM Bitcode* from the compiler. Of these three options, we only consider the first two, which is using the *LLVM C library* or generating *LLVM Assembly*.

The advantage of generating *LLVM assembly* is that it is easy to get started, as the compiler only has to emit assembly strings[36]. *LLVM Assembly* is a relatively high-level assembly language, which features structs, arrays, and functions as first-class-citizens. The main disadvantage of this approach is that if *LLVM* changes the assembly structure, the code generator has to be updated.

In order to use the *LLVM C library*, the language in which we implement the compiler must support calling *C* functions, which limits the amount of languages that the compiler can be developed in. When using a high-level language there will most likely be lots of glue code that wraps the *C* calls to the higher level language[36]. A few advantages of using the library is that *LLVM* optimizes the code while it is being generated, and the library will automatically update the IR when new versions of the library are released[36].

Emitting *LLVM assembly* is easy to get started on, but we would like to have a more stable code generator phase, as well as allowing *LLVM* to optimize our code, and as such we use the *LLVM C library*. While this somewhat restricts which languages we can develop the compiler in, this approach seems to be the most stable, as the *LLVM C library* will not break upon updating *LLVM*[36].

8.2 Choice of Compiler Language

Which language to use for implementing the compiler is the next decision that is taken. As mentioned in Section 8.1, in order to use the *LLVM C Library* the language that the compiler is implemented in, must support calling *C* functions. We want the compiler to be cross-platform, meaning that the language the compiler is developed in, must be cross-platform as well. Furthermore we want to use an object-oriented programming language, which allows us to make classes and subclasses for the node types in the AST, and to subclass a treewalker class which makes it easier to traverse the AST and perform different actions in the different phases. Even with these restrictions a large amount of languages are still viable.

In order to make interfacing with the *LLVM C library* easier, *C++* could be used for the compiler. Using *C++* would allow using the library without having to wrap the function calls. We chose not to use *C++* though, due to the low-level nature of the language. We do not want to spend the development time debugging memory and pointer errors.

There are many high-level languages that offer what we need for the compiler language, such as *Java*, *C#*, and *Swift*. These are object-oriented, with managed memory, and are cross-platform. We have previous experience with both *Java* and *C#*, and have used these languages for various projects. Due to mainly personal preferences, we implement the compiler in *C#*. This has the added bonus that a library that wraps the *LLVM C library* has been implemented in *C#* by Microsoft[37], meaning that we do not have to implement the middle-layer that connects *C* to *C#* ourselves. This API is nearly identical to the *C* library, and supports all functionality exposed by the *C* library[37].

8.3 Compiler Phases

This section contains a description of the compiler phases. The phases describe the actions from the source is inputted, to code is generated. The complete architecture is seen on Figure 8.1. The phases are explained below.

Preprocessing
: The *FuGL* source code is passed to the compiler. The preprocessor loads the imported modules and inserts these into the source code given to the compiler. Once all imports, and their imports are imported, the source is passed to the scanner and parser phase.

Scanner and Parser
: The scanning and parsing phase converts the source code into an AST. The

scanner searches for words, numbers, and symbols, and in turn passes these to the parser as tokens. The parser uses the tokens for building nodes that are used in the AST. When the parser receives a token, it can peek at the next token to decide which type of rule to follow. Due to the syntax of the language, a lookahead of one is enough[32]. Once the AST is built, the compiler can begin working on the tree, which is used in the rest of the phases.

Post Parsing
After scanning and parsing, some early modifications are made to the tree. An example is creating a constructor function for each of the record types defined by the developer.

Validation
This phase validates that the program satisfies a number of constraints set by the rules of the language. The scope checker makes sure that variables are not used when they do not exist, that the functions that are called exists and so on. The type checker attempts to check that types are not used in illegal operations, such as multiplying a boolean value with a number. Initial validation is performed by the *FuGL* compiler, but we also rely on *LLVM* to validate the code.

Optimization
In this phase the code is optimized. We rely on *LLVM* to optimize the code.

Pre Code Generation
This phase makes some necessary modifications to the AST before the code generation. Some operations might need to be restructured in the AST in order for the code generator to perform less work. An example is specializing generic functions.

Code Generation
The *LLVM* intermediate representation is generated using the *LLVM C* library[38].

Once these phases are completed the *LLVM* IR can be compiled for the target platforms using the backends that are provided by the vendors. For *NVIDIA* GPUs we use the *CUDA LLVM Compiler*[34], and for *AMD* we use the *AMDGPU* backend[35]. For the host code we rely on *LLVM* to compile working machine code for the target platforms.

8.4 Summary

In this chapter we chose the backend for the compiler, and the language for implementing the compiler. We chose to implement the compiler in *C#*. We

have previous experience with this language, and it satisfies the requirements for what we need for the compiler, mainly it being object-oriented and managed, allowing us to focus on the implementation instead of pointers and memory. Using an object-oriented language allows us to implement the node types of the AST by subclassing a common node class, making the classes resemble the actual AST.

We have decided to use *LLVM* as the backend for the language. This choice is made not only because many other libraries and frameworks depend on *LLVM*, but also due to the large amount of backends that allows the code compiled by *LLVM* to run on many platforms, including *NVIDIA* and *AMD* GPUs. We chose to use the *LLVM C library* instead of emitting *LLVM assembly*. This makes us less vulnerable to changes in the internal representation in *LLVM* and allows optimizations while the code is being generated by *LLVM*.

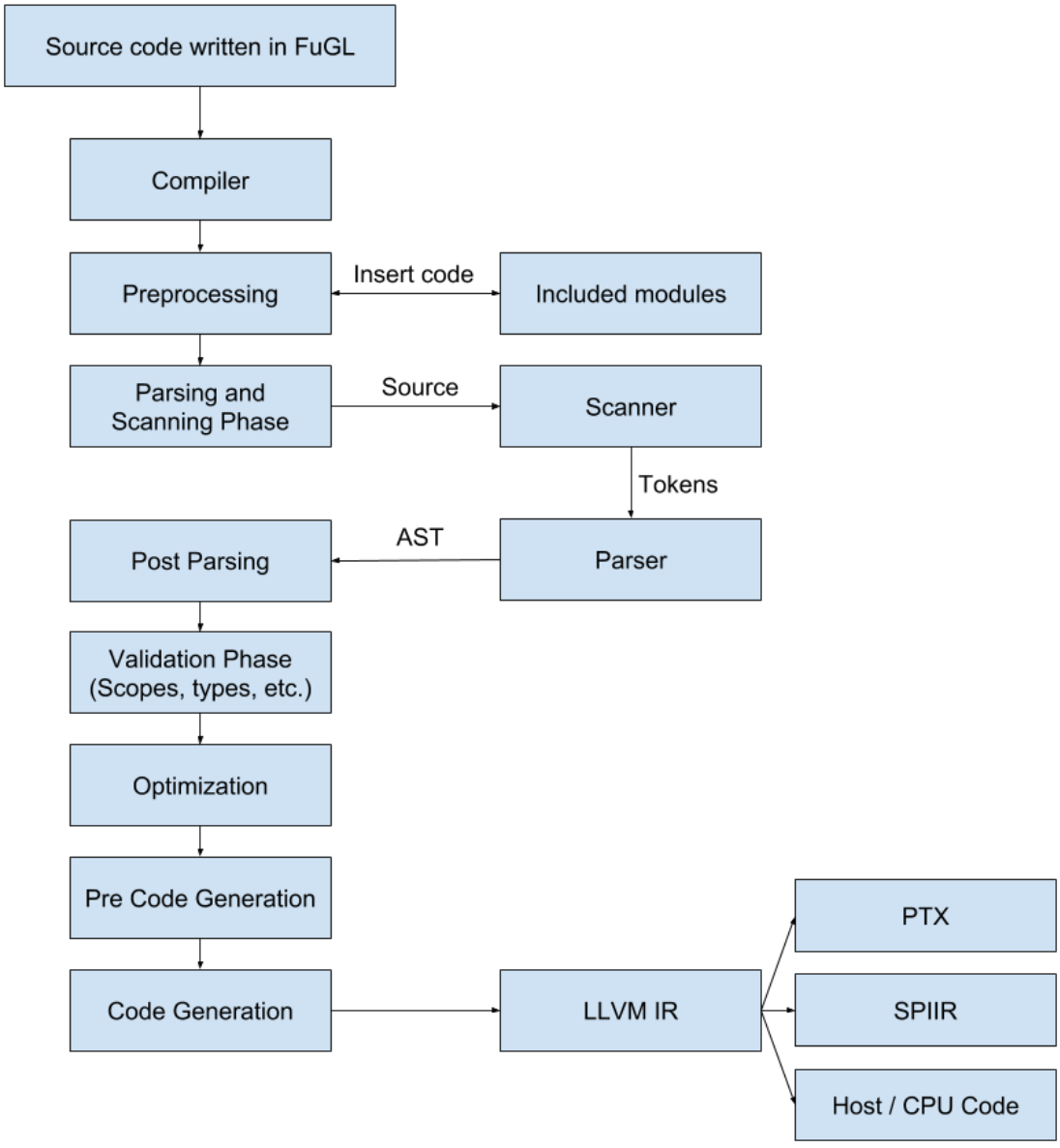

Figure 8.1: The structure of the compiler.

9. Utilizing the GPU

This chapter contains a discussion about how *FuGL* utilizes the GPU, and how some low-level parts of the GPU can be abstracted away from the developer. This includes how memory is handled, how arrays work on the CPU and GPU, and thread and block dimensions. The decisions described in this chapter is based on the analysis performed in Chapter 5.

9.1 Memory Handling

Memory handling is an important and difficult part of GPU development, which requires consideration into how memory is accessed and used, in order to use the GPU efficiently[39][40][41]. Memory can not be allocated by code being executed on the GPU, meaning that the host program has to allocate and deallocate all memory that is used on the GPU. This part of GPU development is problematic in functional programming languages, which often utilize dynamic memory handling for operations such as appending arrays or creating new objects. The host program can make use of dynamic memory handling, as it is possible to allocate, reallocate, and deallocate freely while on the host. In kernels, however, we can not allocate, reallocate, or deallocate memory, which limits the amount of possible operations. This is seen in multiple of the previously discussed solutions, such as the *Julia GPU framework* and *Alea GPU*.

FuGL allows developers to create arrays and objects while on the CPU, just like other functional programming languages. On the GPU we limit these actions, and only allow stack based variables to be allocated and disallow appending or removing objects from arrays. This is necessary as the host program needs to know the size of the arrays or objects that are used on the GPU in order to allocate the required memory.

Our design of handling GPU restrictions is to use a keyword to define when an array is used on the GPU. This keyword determines what actions are possible, but the arrays are otherwise the same way as CPU arrays. Furthermore this keyword determines when to allocate and copy the array to the GPU, which allows the developers to reuse the memory that is allocated for GPU usage. If the host program declares a GPU array, and then calls multiple functions that accepts GPU arrays, the allocated data will be reused. This approach is based on how *Thrust* and *Alea GPU* handles arrays on the host and GPU. Arrays that are not marked for GPU usage and are used in a GPU function will be allocated and copied to the GPU when necessary, but also copied back immediately after usage, even though multiple kernels may need the array. This makes it possible to both reuse allocated data, but also allows the developer to implement algorithms that require data to be moved to and from the device without handling this themselves.

Another issue with memory management is the various types of memory avail-

able on the GPU, such as shared, global, and texture[42], which impacts the performance of an application. *FuGL* has a high level of abstraction, and having to know how the various types of memory works, and what they are well suited for, lowers the level of abstraction. In addition it increases the difficulty of utilizing the GPU for developers that are new at GPU development.

The various types of memory is hidden from the developer, and the compiler chooses which type of memory to utilize. For example, the compiler could analyze the kernel to determine whether it modifies values, and if it does not, it can use constant memory. These optimizations and abstractions are too complicated and out of scope for this project, and are not implemented. *FuGL* therefore only uses global memory.

9.2 Threads, Blocks and Dimensions

In the analyzed libraries and languages, thread and block ids exist as either implicit function calls or variables available from within a kernel. In the *Julia GPU framework*, the thread and block variables can be accessed inside functions that are used on the GPU. In *Alea GPU* the dimensions are implicitly declared for the developer. In *FuGL*, they are also implicitly declared in functions that use the GPU keyword. In addition to this, the developer can access the global thread id, which usually has to be calculated by the developer.

To raise the level of abstraction, the developer is not required to specify block count or thread and block dimensions. *FuGL* uses a default setting that performs well on many algorithms, though not optimal.

Reaching an effective configuration on threads and blocks is a science in itself, and multiple papers and articles have been presented on this topic [39] [43] [44]. To achieve an effective configuration, all GPU cores must be utilized, and to utilize memory hiding, the full amount of supported concurrent threads per core should be started, according to [39]. On modern GPUs this is 1536 concurrent threads per SM[45]. A rule of thumb[39] is that block size should be a multiple of 32 or 64, which fits into *Wavefronts* on *AMD*, and *Warps* on *NVIDIA*.

According to [44], the best performance on a GTX 280 is achieved when running between 4 and 8 warps per SM, which is between 128 and 256 threads. As the GTX 280 was released in 2008, these numbers may be outdated, and GPUs today may perform better with a higher number of threads. But as a reference point these numbers are taken into account. *NVIDIA*[43] indirectly confirms this number stating that blocks should be a multiple of warp size, and sizes between 128 and 256 threads.

According to NVIDIA[43], multidimensional aspects do not play a role in per-

formance, but instead help mapping multidimensional problems onto *CUDA*. [39] does not agree, as they have shown that for specific problems typically including data reuse, performance can be increased by adjusting block dimensions. Thread and block dimensions are not utilized, as we are investigating an efficient default setting for all solutions.

Based on the knowledge from [39], [43] and [44] we strive for the max amount of 1536 concurrent threads per SM, to fully utilize the resources available on a GPU, as described in [39]. According to Intel[44] and NVIDIA[43] the block size should be between 128 and 256, which we choose as a default range.

9.3 Offline or Run-time GPU Code Compilation

To execute code on a GPU, we need to consider how and when to compile the GPU kernels. The two strategies are *offline compilation* and *run-time compilation*.

In offline compilation we compile to machine code, which can be executed directly on a GPU. Choosing this approach can reduce startup time, as everything is precompiled. The negative side of this approach is that we can only compile for a specific architecture at a time. While it is possible to repeat this procedure for each architecture, we eliminate the possibility of supporting architectures developed in the future and future optimizations.

In run-time compilation we compile during run-time, as the name suggests. This allows the compiler to compile and optimize the solution specifically to the present architecture, and removes the need of compiling for any other platform. Furthermore, future optimizations can be applied to the solution because the solution is compiled during run-time. The negative side of this approach is that it may take some time to do these compilations, which can affect the execution time.

An extreme case of the run-time JIT compiling is seen with the *Julia* GPU compiler described in Chapter 5.1. With *Julia GPU*, kernels are JIT compiled all the way from dynamically typed *Julia* code to device code. This provides great flexibility, as a kernel can accept any type of argument. The negative side is that the JIT compilation takes between 2 and 6 seconds[14], depending on the kernel complexity, which is a long time to start a kernel. As *FuGL* does not have dynamic types, JIT compiling all the way from *FuGL* source code to machine code is unnecessary as the types are known at compile-time. An advantage of run-time code generation is that kernels can be specialized, which may increase performance[14].

CUDA

CUDA has two different terms[46] for compiling to the architectures either with speed in mind, or compatibility in mind. These are actual architectures and virtual architectures.

Compiling for an actual architecture reflects the offline compiling approach. This means that device code is compiled, and can be run on the specific architecture. This reduces startup time, as code can be deployed directly on a GPU, but has no support for other architectures.

Compiling for a virtual architecture follows the JIT approach in some way. The source code is offline compiled to PTX code which is a low level IR, but still requires JIT compilation to a specific architecture. When compiling for a virtual architecture, the minimum supported architecture must be specified, as any architecture supports its predecessors. This may, however, reduce optimization or performance possibilities.

As a hybrid approach, *CUDA* supports combining different compilations into binaries, called fat binaries[46]. This allows compiling both for actual architectures as well as for virtual architectures to support future architectures. It also allows the supported architectures to start executing without compilation, and thus fast.

OpenCL

As with *CUDA* and *PTX*, *OpenCL* can be compiled to a low level representation called *SPIR*[47]. When being executed on a system, the driver supporting *OpenCL* compiles the *SPIR* code to device code to be executed on the GPU.

Offline compilation in *OpenCL* requires much more work than in *CUDA*, as it depends on the system to execute the binaries. Many different vendors release drivers that support *OpenCL*[48], but each of them has their own way of implementing *OpenCL*. Furthermore, only some vendors have released an offline compiler[48] to compile device code, but each of these must be used to compile for the hardware released by the vendor.

As offline compilation in *OpenCL* is very complicated we compile to *SPIR*, and let each device driver JIT compile the *SPIR* code to device code.

An interesting project is available at *Github*[49], that allows bi-directional translation between *SPIR* and *LLVM*. As we are already building a compiler with *LLVM* as the backend, the translation project can aid in supporting *OpenCL* devices.

9.4 Summary

In this chapter we described how the GPU is utilized by *FuGL*, how we handle memory allocations and deallocations, thread and block dimensions, and the actual compilation process for GPU code.

Memory is handled for the developer, and the process of allocating and deallocating is abstracted away from the developer. *FuGL* allocates, copies, and deallocates memory based on the usage of variables and modifiers used on the variables. The keywords `gpu` and `mutable` are used to indicate how variables are handled on the GPU, when they are copied back.

Thread and block dimensions are decided for the developer, but the thread id and dimensions can be accessed in kernels. We have decided on a number of threads and blocks that should work well for a wide variety of algorithms. We choose the block dimensions of 256 threads per block, as that number is efficient for many use cases, as described in Section 9.2. We allow developers to choose the number of threads to run.

The compilation for *CUDA* devices happens both at compile-time and at run-time. At compile-time we create a fat binary which contains the code for various architectures, as well as *PTX* code that can be compiled for specific architectures in case the fat binary does not contain the necessary code. For *OpenCL* devices we compile to *SPIR*, and trust the device vendors to compile this to usable code at run-time.

As described in Section 8.1 we use *LLVM* as the backend for the compiler. This chapter contains a description of how we utilize the *LLVM* toolchain, how the *LLVM C library* is used in the *FuGL* compiler, and how *FuGL* constructs are mapped to the *LLVM* IR.

10.1 LLVM Usage

LLVM contains a large number of tools, which can be used by the *Clang* compiler. *Clang* can emit *LLVM assembly* from *C* and *C++* code using the `-S -emit-llvm` command line arguments. This makes it possible to implement a feature of the compiler in *C*, compile it to *LLVM assembly* and reverse engineer the code to determine how to implement the feature in the *FuGL* compiler. This approach is used for determining both how *LLVM* works, and how elements in *FuGL* are mapped to *LLVM*.

We use the *LLVM C library*[38], as described in Section 8.1, with bindings to *C#* using *LLVMSharp*[50] to generate *LLVM assembly*. Using either the *LLVM compiler (llc)* or the *LLVM interpreter (lli)* the compiler can compile to native code or run the generated code directly. GPU code is generated using the *NVPTX*[51] backend for *NVIDIA* GPUs, and the *AMDGPU*[35] backend for *AMD* GPUs. These backends are used as they are developed by *NVIDIA* and *AMD*, and as such, we assume they are working correctly and are well documented.

The *LLVM C library* generates errors during the code generation phase if anything goes wrong, making it possible to skip many validation steps in the *FuGL* compiler and rely on *LLVM*. We furthermore rely on *LLVM* to optimize the generated code.

To learn how *LLVM* works, including the *C* library and assembly, we use a number of tutorials[50] [52] [53], answers from forums, and documentation pages[54] [38] [51]. These sources aid in designing the mapping between *FuGL* constructs and *LLVM* constructs.

10.2 Mapping *FuGL* to *LLVM*

The code generation phase of the compiler consists of converting the *FuGL* IR into *LLVM* IR. In this section we describe how the constructs in *FuGL* are converted into *LLVM* constructs. We show some *LLVM assembly* for each of the constructs, to make it easier to understand the mapping. This section describes the mappings from *FuGL* constructs to *LLVM* constructs.

Functions

Functions are mapped to *LLVM* functions. *LLVM* functions are defined in assembly as such: `define retType @name(pars) { code }`. Here, `retType` is the return type of the function, `name` is the name of the function, always prefixed by an 'at' sign, `pars` are the parameters, which must be defined by a type and can be given a name though it is optional.

An example of a function in *LLVM assembly* is seen in Listing 10.1. This function is called `add`, and accepts two unnamed 32-bit integers and returns an 32-bit integer. As the parameters are unnamed, they can be accessed in the block by using `%0` and `%1`. Named parameters are instead accessed by using their name. All variables are prefixed by a percentage sign.

```
1 define i32 @add(i32, i32) {
2 entry:
3   %tmp = add i32 %0, %1
4   ret i32 %tmp
5 }
```

Listing 10.1: `add` function definition in *LLVM assembly*.

Functions can have multiple *basic blocks*, which are like labels in regular assembly. In Listing 10.1, a basic block called `entry` is seen, which acts as the entry point for the function. More basic blocks can be created, and these can be used for conditional jumps, loops, and such. In *FuGL* , basic blocks are used for *if* -expressions and each function has an entry block.

Records

Records are implemented by using the *LLVM struct aggregate type*[54]. *LLVM* supports two types of struct; one saved in registers which are constant, and one in memory. We first tried to implement records as struct types in memory. This, apparently, is not always possible, as returning structs has to adhere to the platform ABIs[55] [56] [57]. Instead we discovered, through testing and the developer forums, that by using the constant aggregate struct type it is possible to return constant structs from a function by value.

An example of a function creating and returning an aggregate struct type is seen in Listing 10.2. The return function of this type is a structure type containing two 32-bit integers. In order to create and fill a structure type, the `insertvalue` instruction is used. This instruction requires a structure type, a 'current' value, the value to insert, and an index. In order to fill all the values in the structure, each value must be inserted by using the `insertvalue` instruction. Because the structure contains two values, two `insertvalue` instructions must be used.

The first `insertvalue` builds on an `undef` value, whereas the second instruction builds upon the first one, by providing the first struct as an argument.

```
%Point = type { i32, i32 }

define %Point @Point(i32, i32) {
entry:
  %tmp =  insertvalue %Point undef, i32 %0, 0
  %tmp2 = insertvalue %Point %tmp,  i32 %1, 1

  ret %Point %tmp2
}
```

Listing 10.2: Definition and constructor function for an aggregate type.

Types

Mapping types is relatively simple, as each of the types in *FuGL* corresponds to a type in *LLVM*. *LLVM* uses `i1` for booleans and `i8` for characters, as there are no 'native' types for these. The mappings are seen on Table 10.1. Records are mapped as described previously. Arrays are mapped using an aggregate value, which is explained in the next section.

FuGL	**LLVM**
`Bool`	`i1`
`Char`	`i8`
`Int`	`i32`
`Int64`	`i64`
`Float`	`float`
`Double`	`double`
`Void`	`void`
`String`	array of characters
`mutable` values	pointer to type

Table 10.1: Mapping of types.

Arrays

Arrays in *FuGL* are not implemented using *LLVM* array types, as *LLVM* arrays require a known length and type, and can not easily be reallocated. The instruction `alloca` can be used to implement variable arrays, but we need to know the size of an array, which is not saved on the object when using this method. Instead we implement arrays as an aggregate type. This type contains the number of elements, the element size, capacity of the array, and the data. The definition written in *C* can be seen in Listing 10.3. In *LLVM assembly* it is implemented

as an aggregate structure type with the definition: `{ i64, i64, i64, i8* }`. The order of variables is the same as in the *C* implementation.

```
typedef struct {
    size_t size;
    size_t objectSize;
    size_t capacity;
    char* contents;
} Array;
```

Listing 10.3: Array struct as implemented in *C*.

LLVM allows the use of certain *C* functions, such as `malloc` and `memcpy`. These are used when performing array operations. The data part of an array is allocated using `malloc`. The size of the data region is based on the number of elements needed and the size of elements. `memcpy` is used when setting an element in an array, which copies a value into a region of memory. Due to *LLVM* not containing a meta-type describing a type, the compiler has to remember what type a given array is, as it is not possible to have the array know its type. This requires that the compiler uses the `bitcast` instruction on values being inserted or fetched from the array. For example, when inserting an `i32`, this value is cast to an `i8*` to fit into the data region of the array. When fetching elements, the value is cast from `i8*` back to the `i32`.

When the developer uses an array literal, such as `[1, 2, 3, 4, 5]`, the compiler generates a call to `initializeArrayWithCapacity(4, 5)`, where 5 is the amount of elements, and 4 is the size of each element in bytes. Then the compiler sets each element, from 0 to the amount of elements in the literal, using the `set` method on the array with the current element as a parameter.

The append operator, ++, which can be used to append one array to another is replaced before the code generation phase with a call to `appendArrays` function, which allocates and creates a new array from two arrays.

Expressions

Each expression type in *FuGL* , except for a lambda, returns a value. This makes the expressions usable in many places, making it necessary for the compiler to keep track of when and where to set the values generated by expressions, and whether the value should be returned or if they are used in another expression. *FuGL* allows more than one expression in a block, and the last expression is the value that is returned. Multiple expressions might be necessary, for example when printing values for debugging.

Let

Let expressions are used to declare local variables in *FuGL* . These variables are immutable as standard, but can be declared as mutable values, which allows redefinition of the variable. In *LLVM*, a variable is declared with the contents of the expression provided in the let statement, for each of the variable assignments in the let-expression. Scopes do not exist in *LLVM*, and a variable that is declared and assigned can be accessed anywhere in the function as long as it exists, but the *FuGL* compiler can associate a name to a value.

If-elseif-else

`If-elseif-else`-expressions in *FuGL* are implemented in *LLVM*, as seen on Figure 10.1, using conditional jumps, regular jumps and blocks. As *LLVM* supports only a single boolean statement in a conditional jump, an `If-elseif-else`-expression in *FuGL* are implemented in *LLVM* as a nested conditional jump.

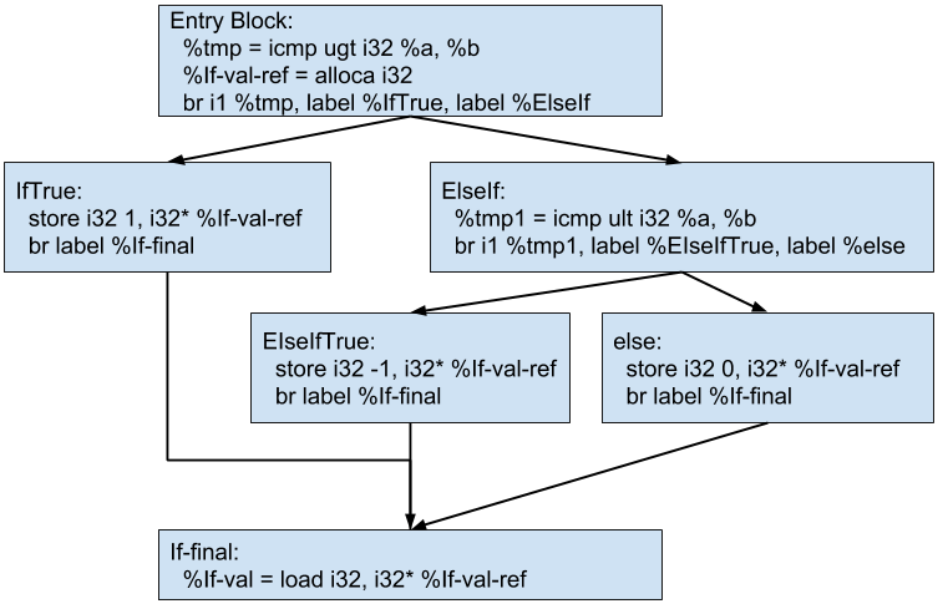

Figure 10.1: The structure of an if expression with one else-if condition in LLVM. This is the *LLVM* generated, when compiling Listing 7.5. For clarification some *LLVM* variables have been renamed manually.

`If-elseif-else`-expressions can return a value from their body. This is implemented in *LLVM*, using a pointer to store the value from any branch. The type of the `If-elseif-else`-expression, is determined by testing the value of the first branch in the `If-elseif-else`-expression.

To continue execution after the `If-elseif-else`-expression, a regular jump

is required to jump to a block containing the instructions to execute after the `If-elseif-else`-expression. Furthermore, the previously saved value from a chosen branch must be retrieved from memory.

Lambda

Lambda expressions are defined as global functions as *LLVM* does not allow functions inside functions. We declare the function with the parameters of the lambda node and return type. The function is declared with a random name to avoid name clashes, which *LLVM* validates for us. The location where the lambda is used is then replaced with a function pointer to the new function, making it callable.

10.3 Summary

In this chapter we described how we utilize *LLVM* in the *FuGL* compiler. For some of the constructs, we implement these in *C* and compile to *LLVM assembly* using *Clang*, allowing us to analyze and reverse engineer how to implement these features. The code is tested using *lli*, making it easy to run the generated code directly on the platform without compiling the code further.

The mapping of *FuGL* constructs to *LLVM* constructs is mostly easy, but some problems were discovered during debugging. For example, we can not return a struct pointer, which could improve performance, but instead have to return aggregate types as structs and pointers are platform dependent. Arrays are implemented as an aggregate type, though this is hidden from the developer. Strings are converted to character arrays, as we want developers to be able to use list functions on strings, and this is easier if the internal representation is the same.

11. Design Summary

In the design part we described how *FuGL* is designed, including the syntax, the inspiration for the language, how the GPU is utilized, and the features we delimit initially. We furthermore described how the compiler is designed, what language it is implemented in, the backend for the compiler, and what phases it contains. How the GPU is utilized in the compiler, was described, which mainly concerned how memory, threads and blocks, and kernel launches, are handled in *FuGL*.

Some of the described design choices are not implemented in this version of *FuGL* due to either time constraints or the features being of low priority. The first delimitation covers the compiler validation and optimization stages. The *LLVM* C library validates the code when generating *LLVM* code, together with the *Clang* compiler which is used to compile *LLVM* IR to native code. As such, there is little reason for the *FuGL* compiler to validate the code. *LLVM* and *Clang* also handles optimizations, making it unnecessary for the *FuGL* compiler to optimize the code. Some validation is performed by the *FuGL* compiler, mainly parts that the *LLVM C library* can not handle, such as scope rules.

As described in Section 9.3 we considered using the *AMDGPU* backend to support GPUs that are not *CUDA* compatible. We delimit us from the use of *AMDGPU* and *OpenCL*, as we do not have access to a non-*NVIDIA* GPU to test the implementation, and because the documentation for *AMDGPU* is poor. In order to implement working GPU code, we therefore only support *NVIDIA* GPUs in the code generation phase.

The last design delimitation is how GPU code is executed. In Section 9.3 we described how code can be run on the GPU for both *CUDA* and *OpenCL* devices. For *CUDA* we found that fat binaries allows for starting kernels fast for the architectures that the binary is compiled for, and for JIT compilation of PTX code for architectures that the binary is not compiled for. The usage of fat binaries is not documented well, and therefore we rely only on JIT compilation of PTX code. This means that GPU code is compiled into PTX code, which is then embedded into the executable and at runtime the PTX code is compiled for the correct architecture.

Part III

Implementation

12. Implementation Introduction

This part of the report documents the implementation of the compiler, including examples of the source code. We describe the scanner and parser, the precode generation, and the code generation implementation.

The optimization phase is not described, as this is not performed by the *FuGL* compiler, but instead we rely on *LLVM* to perform opimizations, as described in Section 8.3. We assume that the source code is correct once the compiler reaches the code generation phase. Some minor validation is performed by the *FuGL* compiler, such as scope checking, though this is not documented.

Each phase after parsing is implemented as treewalkers. These classes inherit from the `AbstractSyntaxTreeWalker` class, which can walk through the AST, following the EBNF rules. Each of the subclasses can override or extend functions from the base class and perform modifications or analysis of the tree and individual nodes. Each of the subclasses performs only one action thus they should be independent, though some phases perform work required by later phases. The treewalker subclasses used in the compiler are as follows:

PrettyPrinter
: Prints the AST.

PostParsing.GlobalFunctionTable
: Finds all functions, their parameters, and return type. These are saved into a global table as many phases need a list of functions.

PostParsing.GlobalTypeTable
: Finds all record types, as well as the contents of them. These are saved into a global table to be used in future phases.

PostParsing.Generics.SpecializeGenericFunctions
: Specializes calls to generic functions. The algorithm is explained in Section 14.1.

PostParsing.Generics.ReplaceTypeInFunctionWalker
: Helper class used by `SpecializeGenericFunctions` to replace all occurrences of a type in a function.

PostParsing.ArrayEqualityReplacement
: Replaces occurrences of `array1 == array2` with a call to `arEqs` with the two arrays.

Validation.FunctionCallScopeChecker
: Validates that all functions that are called exist.

Validation.VariableScopeChecker
: Validates that variables exist in the scope they are accessed.

Validation.ArgumentTypeCheck
: Arguments to function calls are validated, mainly illegal implicit casts.

Validation.ArrayFunctionCallChecker
Checks whether `set` is being called on a mutable value.

Validation.LetReassignmentMutableCheck
Validates whether reassignments are performed on mutable values in *let*-expressions.

PreCodeGen.AppendArrayConverter
Converts the ++ operator to function calls, to avoid this operator being a special case in the code generator.

PreCodeGen.CastSpecializer
Specializes calls to the casting functions, such as `toInt` and `toBool`.

PreCodeGen.FindCaptureVariables
Used for finding captures in lambdas. Walks the bodies of lambda expressions and determines which variables are declared outside the lambda and then saves these for usage in the code generation phase.

PreCodeGen.ConvertStringToArray
Converts string literals into arrays. This is performed as strings act as arrays of characters in *FuGL*, and the internal representation is easier to handle if they are implemented the same way.

CodeGen.GPUCodeGenerator
Generates *LLVM* assembly that is compiled to PTX code for the GPU.

CodeGen.LLVMCodeGenerator
Generates *LLVM* assembly for the CPU.

13. Scanner and Parser

This chapter contains a description of the scanning and parsing phase of the *FuGL* compiler. The first phase of the compiler is the preprocessor which reads imported files and inserts the source code into the main program file. After the imports are handled, the source code is passed to the scanner and parser.

The purpose of the scanner and parser is to create an AST from the source code. This AST is used throughout the rest of the compiler phases, and can be modified by each phase if necessary. The scanner provides the parser with tokens, and the parser then uses these tokens to build a tree by following the rules in the EBNF. The EBNF for *FuGL* is seen in Appendix A. For each of the rules in the EBNF, the parser contains a function that attempts to parse the given rule, such as `ParseFunctionDeclaration` and `ParseBinaryOperatorExpression`.

13.1 Scanner

The scanner reads the source code, one token at a time. In the *FuGL* compiler, tokens can be either keywords, symbols, names, or literals. These types of tokens can be combined to create the rules from the EBNF. The scanner also keeps track of the current line number.

Two functions are exposed in the scanner for the parser; `Get`, which gets a token, and `Peek` which peeks a token. The `Peek` function returns the current token, but does not find the next token. The `Get` function returns the current token and then finds the next token. When the parser needs to know which token comes next, but without consuming the token, `Peek` is used. When the correct rule is determined by the parser, the token can then be consumed and the parser can continue.

The scanner finds a token by iterating the source code, character by character, skipping whitespace and comments, until it reaches a character that must be handled. This might be a letter, a number, or a symbol. The type of a token is represented by the enum `TokenType`. The scanner knows the keywords, and creates a keyword token if the found letters match a keyword. Integer and floating point numbers are put into the same kind of token, as the type of number is not important in building the AST. The scanner knows the various symbols in the language, and the symbols are given a type before being returned to the parser. Pseudocode describing this is seen in Listing 13.1.

```
1 source is string containing the program source code
2 cursor is integer value describing the current location in the source code
3
4 while cursor < length of source:
5   move past whitespace characters
6
7   char = character at index cursor in source
8   increment char
```

```
9
10  if char is symbol:
11    return symbol token
12
13  else if char is letter:
14    name = read the rest of the letters
15
16    if name is keyword:
17      return keyword token
18    else
19      return letter token
20
21  else if char is number:
22    read the number
23    return number token
```

Listing 13.1: Pseudocode presenting the scanner.

13.2 Parser

The purpose of the parser is to build the AST based on the tokens provided by the scanner. The parser requests a token when necessary, using either `Peek` or `Get`. When a token is requested, the type of the token is checked and if the token type is usable in the current state of the parser, a rule from the EBNF is expanded and a node in the AST is created.

The nodes created by the parser all inherit from the `Node` class, which contains a reference to a parent node. The only node without a parent, is the program node. The expression nodes in the tree inherit from the `ExpressionNode` class, which makes it easier to create and use expressions as these can all be used in the same locations. The classes used in building the AST can be seen in Appendix C.

As seen in the EBNF, which is found in Appendix A, the root node in the tree is the program itself, which consists of multiple namespaces. Namespaces are named containers that contain functions and types. Before the parser is started, the source code given to the compiler is wrapped in a namespace called *program*. Namespaces are currently not used, but are implemented for usage in the future to avoid name clashes on functions and types. It is delimited in the current implementation, due to it not being that important and thus we have decided not to spend the time on implementing it.

After the scanning and parsing phase, all other phases inherit from the `AbstractSyntaxTreeWalker` class. This class 'walks' the program, recursively walking nodes in the tree, based on the rules of the EBNF. Each phase performs one action only, and inherits from the treewalker to perform this action, such as type or scope checking. Subclasses of the treewalker can modify which order to walk nodes, or skip nodes entirely if necessary.

Operator Precedence

LLVM assembly is structured as single operations, such as `'mul op1 op2'` and `'add op1 op2'`, and *LLVM* itself does not have any knowledge of operator precedence. This requires the *FuGL* compiler to handle operator precedence for mathematical correctness and correct variable access. *FuGL* follows the *C++* operator precedence rules[33] where applicable.

Operator precedence is not presented in the grammar of *FuGL* as this would complicate the grammar, and many different expression nodes would have to be introduced. Instead of a complicated grammar, the parser enforces operator precedence by rotating binary operator subtrees. The rule it actually enforces is, that no binary operator can have a binary operator child, with a higher precedence than itself. Pseudocode for this algorithm can be seen on Listing 13.2.

```
1  root is root node in subtree, of type BinaryOperatorExpressionNode
2  left is left child of root
3  right is right child of root
4
5  highestBinOpChild = node among root node and children with highest precedence and ↩
       of type BinaryOperatorExpressionNode
6  while highestBinOpChild != root:
7
8    if highestBinOpChild is left:
9      rotate subtree right
10
11   if highestBinOpChild is right:
12     rotate subtree left
```

Listing 13.2: Pseudocode describing the precedence algorithm.

A special case which is handled separately is when a unary operator is applied to a binary expression with a dot operator. The dot operator is used for accessing values on record types. According to precedence rules, the dot operator must be applied to the operands before applying the unary operator. The issue is, that the previously described algorithm only applies precedence to trees of type `BinaryOperatorExpressionNode`. To solve this problem we force a binary expression with a dot operator to become a child of the unary expression, during the parsing phase. This way the dot operator is captured in the right precedence level, and is not modified by the previously described algorithm. This is a special case, but is necessary in order to correctly handle unary operators on values being accessed from a record type.

13.3 Next Phases

The AST created by the scanning and parsing phase can be passed to the next phases in the compiler. As described in Chapter 11, we do not implement any

optimization but instead rely on *LLVM* to perform this, thus the next phases validation, preparation for code generation, and the code generation phase itself. We do not document the validation, as we mainly validate minor parts of the code, such as scope checking.

Some actions are performed just after the parser, and though they are not performed in the parser class, they should be considered part of the parsing phase. The classes `PostParsing.GlobalFunctionTable` and `PostParsing.GlobalTypeTable` are used before running other phases. These find all functions and types and save these in a global list that can be used by the other phases to find the definition of a function or a type without walking the tree again. These tables are used throughout the rest of the compiler.

14. Code Generation

This chapter contains a description of pre code generation and code generation. These are the final phases of the compiler, where the *FuGL* IR is prepared for code generation, and the *LLVM* code is generated. The output from these phases is *LLVM* assembly for both the CPU and the GPU.

14.1 Pre Code Generation

Before code generation is started, the pre code generation phase is executed. In this phase the compiler analyzes parts of the code to prepare for code generation, and performs some modifications to the AST. Most of the modifications could be performed during the code generation phase, but to make the code generation implementation more concise, these changes are made beforehand. This is done in order to make the code generator not having to consider the logic and details of *FuGL*, but instead simply emit code based on the AST. Multiple tree walkers are used, one for each modification or analysis phase.

All occurrences of the ++ operator, which append two arrays, are replaced with a function call. This occurs in the `AppendArrayConverter` class. The array function `appendArrays` appends two arrays and returns a new, appended array. We create a new `FunctionCallExpressionNode`, where the two parameters are the expressions from the `BinaryOperatorExpressionNode`.

Strings are converted into array literals to match the internal representation of strings to arrays, as we consider a string an array of characters. This substitution is performed in the `ConvertStringToArray` class.

Lambda captures are found by walking lambdas and finding variables that are defined outside the lambda. This is performed in the `PreCodeGen.FindCaptureVariables` class. The list of captures is used in the code generation phase for implementing the struct that is used to pass the captured variables. This is described further in Section 14.2.

Generic Types

As described in Chapter 7, generic types allow creating functions that can be called with any type of parameter. Any parameter that uses a type that is not known, is assumed to be a generic parameter. An example is the `map` function definition, which can be seen on Listing 14.1. This function can be called with an array of any type, a function that converts an object to another object, and then the `map` function returns an array of these objects. `A` and `B` can be of different types, but can also be the same type.

```
1 map([A] lst, (A -> B) fnc -> [B])
```

Listing 14.1: Definition of `map`.

Before the code generation phase, generic functions are specialized. Our approach to specializing functions is similar to *C++* templates, where a specialized function is created for each used type for a given function. In *FuGL*, the developer does not have to specify the types, as these are found by the compiler based on the arguments provided in the function calls. The specialization occurs in the `SpecializeGenericFunctions` and `ReplaceTypeInFunctionWalker` classes. `SpecializeGenericFunctions` is the main class for handling conversions. This class finds generic functions and calls to these, finds the provided types to the generic parameters, and then uses `ReplaceTypeInFunctionWalker` to clone and specialize the functions with the new types. After all calls to generic functions are specialized, the generic versions of the functions are removed from the AST. We have designed the following algorithm which consists of four phases:

1. Find and save functions that use generic parameters. Remember the index of each of the generic parameters in the function. Build a tree for each function type, which can contain nested types.
2. Walk the body of all non-generic functions, i.e. functions that do not have any generic parameters. While walking the bodies, find function calls to generic functions. Functions that have previously been walked are ignored.
3. Determine the types provided to the generic parameters, and determine whether this combination has been met previously. If not yet, clone the generic function and specialize it with the provided types and add it to the AST. Replace the function call with a call to the specialized function instead of the generic function.
4. If a new function was added to the AST during the algorithm, run the algorithm from step 2 again. If no new functions were added, meaning that no new generic combinations were met, the algorithm removes all generic functions from the AST.

Consider the function `f((A -> (B -> C)) fc -> C)`, which takes a function that returns a function as parameter, and returns a value of type `C`. In order to determine which types are provided to this function, we build a tree containing each of the types in a function, and for each nested function type we generate a new layer in the tree. The compiler finds three generic types in this call, and each of them are given an index in their parent tree. Return values are given the index `-1`. Each generic parameter knows its own index and its parent. A call to this function with a function as parameter of the type `(Int -> (Char -> Double))` allows us to build a tree, and find `A`, `B`, and `C` in the provided type. An

example of these trees is seen in Figure 14.1 and Figure 14.2.

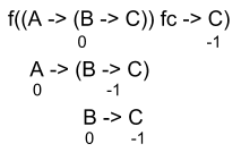

Figure 14.1: Generic types in the function.

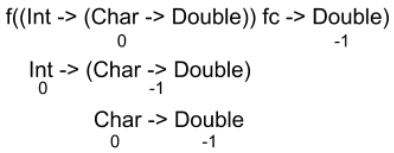

Figure 14.2: Provided types to the function.

Each type knows its index and its parent, and due to this we can find the direction from the top node to a specific type, by walking from the node and up the tree, saving each index we encounter and then reversing this list. To find the type of `A`, we walk from the top node into the function type, and find the type indexed with 0, giving us the `Int` type. To find `C`, we go into 0, then into -1, then -1 again. This is done for each given generic type and argument. In the end the compiler knows which types to replace `A`, `B`, and `C` with, and a function can be generated. The specialized function is named after the provided types, such as `Int-Char-Double-f` for the example function. After specializing a function it can then be walked in the next iteration of the algorithm. This iteration then specializes calls to generic functions inside the newly specialized function, thus handling calls to generic functions inside the previously generic functions.

14.2 Code Generation

The implementation of the code generation is performed in multiple steps, in a waterfall manner. This means that we implement the CPU code generation part first, then the middle-layer that connects the CPU to the GPU and then implement the GPU code generation. The work is split into smaller steps, as to not take on too much at once, and instead perform smaller incremental tasks. We furthermore follow the idea from the *Julia GPU* framework where the GPU code generation is the same as the CPU code generation, but with certain exceptions handled specifically[14].

14.2.1 LLVM

The code generation phase is contained in the `LLVMCodeGenerator` class and `GPUCodeGenerator` class. These classes walk the tree and use the *LLVM C library* to create *LLVM* functions, values, and types, which the library can then use to generate *LLVM assembly* or *LLVM bitcode*. The `GPUCodeGenerator` class inherits from `LLVMCodeGenerator`, allowing us to reuse the code generation implemented for the CPU, and then override specific functions when necessary.

An example is how some types are different on the CPU and GPU, specifically arrays. Arrays on the CPU are implemented using aggregate structure types containing the number of elements, the capacity, the element size, and the data, while on the GPU they are implemented as an aggregate type only containing the size of the array and the data region. By overriding the function that determines the *LLVM* type of a *FuGL* construct we can easily implement this difference. Figure 14.3 shows how the compiler generates and handles CPU and GPU specific code.

Most calls to the *LLVM C library* return either a `LLVMValueRef` or a `LLVMTypeRef`. Everything representing a value is of type `LLVMValueRef`, and type representations are of type `LLVMTypeRef`. Much of the work in the code generator consists of converting the *FuGL* representation into a fitting *LLVM* representation. The design for the mappings is described in Chapter 10. *LLVM* makes sure that the names we provide do not clash, and that the types are correct. Many functions in the *LLVM C library* accept references to values, and *LLVM* handles all naming problems.

The code generation class is a treewalker subclass, i.e. that the compiler walks the AST and generates *LLVM* instructions from the various *FuGL* constructs. This structure is recursive, but we 'flatten' this when generating code, to fit the *LLVM* structure as *LLVM* assembly does not contain blocks and nesting. For example, a *FuGL* function with a few levels of nesting contains no nesting when the *LLVM* code is generated. We use a stack-based approach to flatten the code, as stacks fit well when working with nesting. When an expression is generated, we push the `LLVMValueRef` generated from the expression to a stack of expression values, which can be used by parent expressions, and returned from the function.

14.2.2 Lambda Captures

Lambda captures is a feature that allows variables used in lambdas to come from another scope, and the variables are then 'captured' inside the lambda. This feature is important when using certain higher-order functions, such as `filter`. Without captured variables, the `filter` function can only filter lists based on hard coded constants or the input parameter of the lambda, which is not very useful. The values of captured variables are not always known at compile time, and as such has to be inserted on runtime. An example of a case where a lambda capture is necessary is seen in Listing 14.2. Without being able to use the variable `cp`, which is not a parameter to the lambda, `filter` would not be usable, and it would be necessary to implement a new version of `filter` with the necessary parameters.

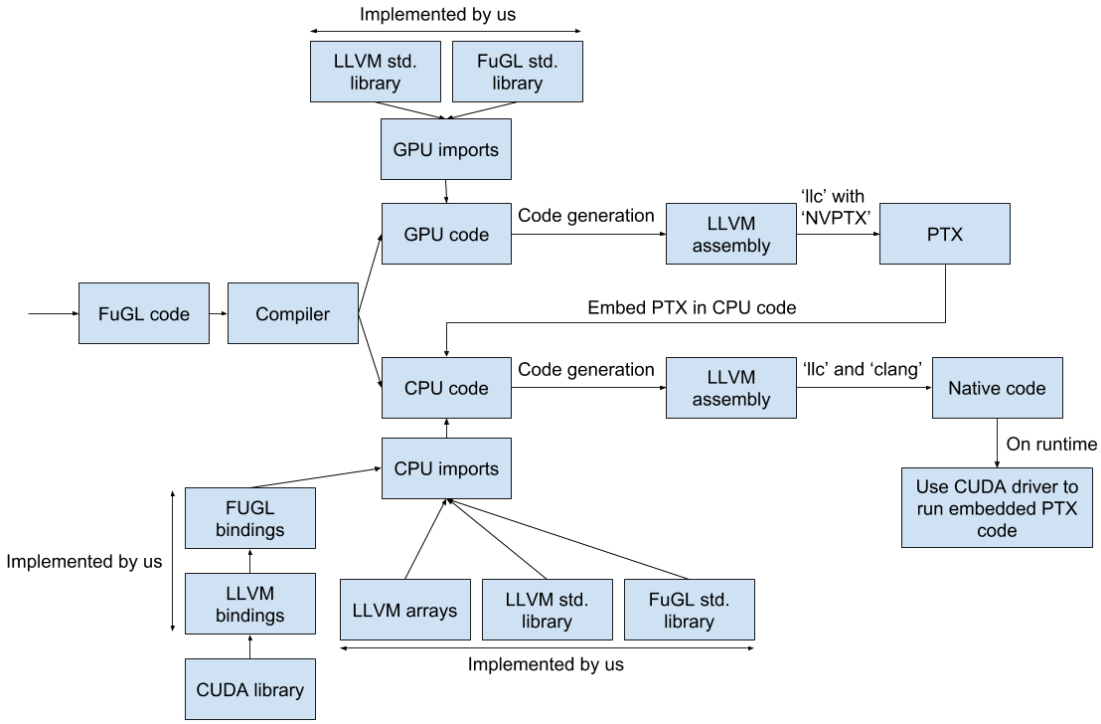

Figure 14.3: How CPU and GPU code generation is handled by the compiler.

```
1 func moveCentroid(Centroid cp, [DataPoint] dps -> Centroid) {
2   let [DataPoint] insideDPs = filter(dps, func(DataPoint dp -> Bool) { dp.label == ↩
        cp.label }) {
3       // Removed for clarity
4   }
5 }
```

Listing 14.2: Capturing a variable in a lambda.

Functions nested in functions are not possible in *LLVM*, which is why lambdas in *FuGL* are generated as regular functions during code generation. The parameters and return type are known from the lambda definition, and the lambda is easily converted into a regular function. We considered implementing lambdas inline, i.e. in the body of the declaring function as a block, which makes it easy to capture the variables in the block. Unfortunately lambdas are often called from another function where the lambda is provided as a parameter, which makes this approach unviable, as the captured variables do not exist inside the function that calls the lambda.

Instead, captured variables are saved in a struct that is passed to the lambda function packed into an `i8*`. Each value captured is represented in the struct, and the lambda function then unpacks the values from the struct. The values are packed into an `i8*` to make it 'generic', thus every function that the lambda is passed onto does not need to know the captured values. Instead functions can just pass the `i8*` to the next function. This approach requires that each function defined in *FuGL* must have a new parameter in the *LLVM* representation, the `i8*`.

Consider the code in Listing 14.3. The code shows the function `doSomething`, which accepts two numbers and a function as parameters. The functions `add` and `sub` matches the type of the function that `doSomething` accepts. The problem is that the lambda is declared as a function, the parameters are no longer `(Int, Int)`, as the `i8*` is added to the parameter list, to be able to capture the `c` variable at runtime. To correct this issue, we add an `i8*` to all functions that are defined in *FuGL*. This means that the `add` and `sub` functions are of the same type as the lambda. When these functions are called the `i8*` is empty, whereas it contains the captured variables in the lambda call. The compiler keeps track of which lambda captured which variables, and it is able to recreate the values from the struct inside the body of the lambda function.

```
1 // Definitions
2 func doSomething(Int a, Int b, (Int, Int -> Int) fnc -> Int) { fnc(a, b) }
3 func add(Int a, Int b) { a + b }
4 func sub(Int a, Int b) { a - b }
5
6 // Calls
7 doSomething(1, 2, add)
8 doSomething(1, 2, sub)
```

```
9 doSomething(1, 2, func (Int a, Int b -> Int) { a + b + c }) // Assume 'c' is ↩
      defined outside
```

Listing 14.3: Function definitions that should be usable.

While this is an unfortunate side-effect of how captures and lambdas are implemented, this approach works. Most of the functions do not need the `i8*` and it is mostly being passed as a null value when calling functions. It is only used on lambda capture values. The parameter could be used in future versions for data that has to be passed to functions, though currently it is only used for lambda captures.

Lambda captures are handled in an early phase of the compiler, in the `FindCaptureVariables` class. This class saves all captured variables in the `LambdaExpressionNode` object that contains the lambda which accesses the captured variables. All variables that are accessed within a lambda node, but are not declared inside the lambda, are assumed to be a captured variable.

14.2.3 CPU-GPU Code Generation

In order to bind GPU and CPU code together, and thereby to be able to execute GPU code, we need to implement some function calls to the *CUDA* driver API[58]. These calls provide the functionality required to allocate and manage resources and execution on the GPU. *LLVM* provides a user guide to get started with this task[51], but only provides an imperative implementation.

To support functional programming and increase the abstraction from the driver API, we wrap these API calls in two levels of abstraction. The first level of abstraction is implemented as a *CUDA* wrapper, whereas the second level is the designed abstractions provided by *FuGL*.

CUDA Wrapper

To implement the *CUDA* Driver API, and create abstraction over this API, we have created a *CUDA* wrapper in *LLVM*. Its main purpose is to handle some of the operations required to communicate with the API. These operations can be data extraction or construction in the form of *FuGL* internal types, pointer handling or error code handling. As not all *CUDA* functionality is supported by *FuGL*, some parameters can be left out. These parameters are hidden by the *CUDA* wrapper.

Pointers are heavily used in the *CUDA* Driver API. For example, they are required to retrieve any value from an API call, except from the call error code returned by any API call. As pointer types do not exist in *FuGL*, it is not possible to declare a pointer type required by the Driver API. Instead, the *CUDA* wrapper handles this declaration, which is then passed to an API call. After the API call the value of this

pointer is then returned from the *CUDA* wrapper, and the error code returned from the API call is stored in a global variable within the *CUDA* wrapper. This error code can be retrieved by calling `getCudaError`.

The *CUDA* Driver API uses a launch kernel call to start the execution of kernels. The launch kernel call is implemented in two versions in the *CUDA* wrapper. The first version is a regular launch call, where all thread and block dimensions can be specified. A second version, is implemented where only a thread count must be specified. The called function then handles the calculation of a block count, based on the total number of threads, and a number of threads per block. Determining a good amount of threads per block was discussed in Section 9.2.

FuGL Abstraction

As the *CUDA* wrapper supports *FuGL* types, it can be used directly in a *FuGL* implementation. The driver API can be called directly from *FuGL* code, but this approach is not recommended as the structure of this library is heavily imperative. Therefore a second level of abstraction is implemented, hiding all calls to the *CUDA* wrapper under the `gpu` keyword. The syntax design is previously described in Section 7.1.

As previously described, variables behave differently depending on their keywords, but the keywords hide many different calls to the *CUDA* wrapper. These keywords are translated to functionality in the compiler that builds the copy calls to the *CUDA* wrapper.

Depending on the keyword used on an array, there is some difference in where the copy call is built. If the `gpu` keyword is used on the array, the copy is built in the initialization of the array, whereas in other cases the copy call is built just before the kernel launch.

To start a kernel, a `gpu` struct must be declared where the number of threads is specified. This is shown in Listing 7.11. For this context, *gpu* is a struct that can be instantiated with a thread count. When launching a kernel on this struct, the thread count is retrieved from the struct and passed to the kernel call, which is a special struct behaviour.

To prepare all kernel parameters, a helper function is implemented in the compiler, that takes a kernel name, a thread count and a list of expression nodes. This helper function walks the expressions given to the kernel call, to build a pointer array which can be passed to the launch kernel call in the *CUDA* wrapper. During the preparation of arguments, the helper function determines whether the expression is an array that must be copied to the GPU and whether this array should also be copied back. In addition, the instructions for doing this are built for each array.

14.2.4 GPU Code Generation

Inspired by the compiler structure of *Julia*, as described in Chapter 5.1, the GPU code generator is built on top of the main code generator. We have implemented this structure using class inheritance, where *Julia GPU* utilizes interfaces for communicating with the main compiler. In both cases the outcome is the ability to reuse all functionality from the main compiler, and to override functionality not supported on the GPU. The GPU compiler subclass modifies a few operations from the main compiler, mainly how arrays are handled. For example, when getting and setting elements from an array, the `get` and `set` functions are overridden.

The compiler furthermore makes some 'macros' that allow developers to access GPU variables at runtime, such as the block dimensions and thread ID. These are accessed by developers by using `gpu.VARIABLE` where *VARIABLE* is the variable being accessed. For example: `gpu.threadID`, which returns the global thread ID for the current thread. These macros are converted into the *CUDA* calls seen in [51], as well as a custom wrapper function that calculates the global thread ID based on the block and thread dimensions.

As the host is responsible for starting a kernel, kernel code must be available to the host. This is done by embedding PTX code into the main program, storing it in a global string. When the compiled program is started, the PTX code is compiled at runtime to the specific architecture, and stored as a *CUDA* module. To start a kernel, the *CUDA* module is accessed and the required kernel function is extracted from the module. This process is shown on Figure 14.3.

In GPU code, functions can be either a kernel or a device function[51]. When marked as a kernel, the function can only be called from the host, and when not marked as a kernel, the function is a device function only callable by the GPU. Therefore kernel functions cannot call themselves in a recursive manner, which conflicts with the structure of a functional language. To accommodate this, we generate both a kernel function and a device function if necessary.

To minimize the size of the PTX string embedded in the program, we run some simple analysis on the GPU functions, to determine which functions are called by the GPU. The intention is to limit the amount of GPU functions required by the program. Additionally we only generate the kernel functions actually called by the host program. This removes all unused kernel and GPU functions.

15. Implementation Summary

This chapter concludes the implementation of the *FuGL* compiler. The code generation phases contained multiple interesting problems we had to solve, such as implementing templating for generics. Some decisions taken in the design phase were found to be problematic in the code generation phase, and we have changed some of our early decisions during this phase.

The largest issue we found is that recursion is not implemented properly in *FuGL*. We expected that *LLVM* was able to optimize and handle recursion, but this does not seem to be the case. Initial testing of *FuGL* showed that recursion breaks the stack much earlier than expected, making it difficult to implement many algorithms in a functional programming manner. We were unable to correct this issue and implement recursion in a way that does not break upon too many layers of recursion. Instead we implemented loops.

The looping construct was implemented as a function in *FuGL*, with the following definition: `for(Int64 start, Int64 end, (Int64 -> Void) fnc -> Void)`. The function accepts a function that takes the current iterator value as parameter. An example of using the `for` function is seen in Listing 15.1, which sets each element of an array to the value 0. This is a very imperative construct, and does not fit into the functional paradigm, but it is necessary in the current state of *FuGL*.

```
1 for(0, length(ar) - 1, func (Int64 i -> Void) {
2   set(ar, n, 0)
3 })
```

Listing 15.1: For-loop in *FuGL* that sets each element of an array to 0.

Although the `for` function does not fit into a functional language, it makes it easier to parallelize many implementations, as the loop construct and loop index can be replaced by the `gpu` keyword and `gpu.threadID` variable, and thereby execute in parallel.

The second largest issue we discovered is the lambda capture problem. Lambdas were not designed with captures, but after some initial testing we found that some problems were very difficult to solve without capturing variables, such as using `filter` to filter lists based on two variables. This problem was solved by adding a new parameter to all functions implemented in *FuGL* and passing the captured variables, at runtime, to the functions which can then pass them to the lambda when called. Lambda captures also allow the new `for`-loop construct, as it can modify arrays and variables from outside the lambda provided to the `for` construct.

This version of the *FuGL* compiler do not collect garbage on the CPU, and only frees memory allocations on the GPU, but does not free the context or kernel modules. It is possible to call `free` on arrays if necessary.

Other than the discovered issues in the implementation of *FuGL*, the implementation is working as intended with high abstractions and GPU usage in a functional manner. The compiler first scans and parses the input, and the resulting AST is then passed to multiple phases that modify or analyze the AST for usage in the code generation phase. The code generation phase outputs *LLVM assembly* for the CPU, with the GPU PTX code embedded as a string. This means that everything is contained in one file, making it easy to compile and run the program.

In order to compile *LLVM assembly*, *clang* and *llc* are required which handles compiling *LLVM assembly* to PTX code, and *LLVM assembly* to executables.

15.1 Implementation Future Work

In this section we describe some features that are planned and designed in *FuGL* but not implemented, and some features we would like to improve upon. We describe the technical details of how these features can be implemented, and why they are currently not implemented. This is not an exhaustive list, but it contains some specific features and corrections where we have considered, and designed the implementation.

CPU to GPU Lambdas

This feature covers the declaration of a lambda on the CPU, which is then executed on the GPU. It is currently only possible to use a lambda declared on the GPU, in kernel functions. This feature would be useful in many situations, such as implementing a `for`-loop function which can execute on the GPU, as seen in *Alea GPU*. An example of this can be seen in Listing 15.2, which implements vector addition on the GPU using a for-loop. We have two implementation ideas for this feature, both of which are untested.

```
1 gpu(length(C) - 1).for(func (Int64 n -> Void) {
2     set(C, n, get(A, n) + get(B, n))
3 })
```

Listing 15.2: For-loop using CPU to GPU lambda.

The first approach is to generate all known CPU to GPU lambdas as kernels, and then let the CPU access the pointer to these functions, so they can be provided as function pointers to the kernels that accept functions as parameters. Our idea is to implement a kernel that is executed before the program starts, which returns an array of function pointers of all known lambda functions. The function

pointers can then be passed to the kernels when they are executed, thus using the function as a lambda. This is only an idea, and we do not know whether this is possible. It would require extensive testing.

The second approach is much more simple and safe than the first one; we specialize kernels that take lambdas, inserting the lambda functions into the kernels. This approach provides more kernels thereby more generated code, as each use of a kernel requires a specialization, unless the same lambda is used more than once. This approach could provide better performance than using function pointers.

Improve Capture Performance

In order to implement anything useful in *FuGL* without having to reimplement higher-order functions such as `filter` each time they are used, lambda captures are implemented. The current implementation of captures is described in Chapter 14.2.2. Captures are used extensively when utilizing lambda functions, such as in the `for`-loops we implemented. Testing showed that `for`-loops are very performance draining due to the casting and unpacking that is performed due to captures, and the performance of captures should be improved upon.

Our approach for improving the performance of lambda captures is to specialize the functions that accept the lambdas with the captures. This means that we will compile more functions and create a larger executable. It should be faster than the current approach, as there is no longer a need for loading the values from the pointer and struct. This will furthermore remove the need for having the `i8*` as the last parameter on each function, as that is only used for lambda capture transport. The specialization approach matches our approach for implementing CPU to GPU lambdas, meaning that implementing one of these should make it easier to implement the other.

Partial Application

Partial application was delimited in the language design phase, as described in Section 7.2. This feature would allow developers to apply one or more parameters to a function, which in turn would create a new function with these parameters already set, thus decreasing the arity of the new function[31] and allowing it to be called with the remaining parameters. An example is seen in Listing 15.3, where the number 1 is applied to the `add` function, thereby creating a new function which now only accepts one integer and adds 1 to the given argument.

```
1  // The 'add' function adds two integers.
2
3  let (Int -> Int) addOne = add(1)         { // addOne is now a function that accepts↩
       one integer and adds one to this integer
4    addOne(2) // All parameters are applied, call the function with the values (1 and↩
         2)
5  }
```

Listing 15.3: Partial application example in *FuGL*.

Our approach to implementing partial application is to save the variables that are applied into a struct, if the number of parameters is less than the expected number. This struct can contain the function pointer, and each of the applied parameters. If more parameters are applied, a new struct with the new values is created. When the function is called with the required number of parameters, the struct can unpack all the values and call the function in the pointer in the struct. For the *FuGL* code seen in Listing 15.3, the corresponding pseudo-*LLVM* assembly can be seen in Listing 15.4.

```
1  ; Definition of 'add'
2  define i32 @add(i32, i32) {
3    %val = add i32 %0, %1
4    ret i32 %val
5  }
6
7  ; When developer calls add(1), we create the following struct, which contains a ↩
       pointer to the function, and the value applied:
8  ; Struct of type :
9  { i32 (i32, i32)*, i32 }
10
11 ; with contents:
12 %addOne = { @add, 1 }
13
14 ; When the developer calls the newly created 'addOne' function with a value, we can↩
       unpack the function pointer and the value and call the function at the ↩
       pointer.
```

Listing 15.4: Partial application in pseudo-*LLVM* assembly.

OpenCL Support

In this version the *FuGL* language only supports *NVIDIA* GPUs using the *NVPTX* backend for *LLVM*. In Chapter 11 we delimited *OpenCL* support due to lack of time and documentation. As described in Section 9.3, there is a project which attempts to translate *LLVM* to *SPIR*, available at [49]. Utilizing this project should be attempted in order to support *OpenCL*, possibly without having to rewrite the GPU code generator.

Tuples

Tuples are not implemented in *FuGL*, but were considered in the project. Tuples can be used to return or transport multiple values as one, much like a record type, but unnamed. There are generally two ways tuples work in most languages; either the values inside the tuple are named, and can be accessed by name, or they are unnamed, and can be accessed by using the index of the value. How they should be designed in *FuGL* was not considered, but both approaches can be easily implemented. We chose not to implement tuples, due to time constraints as well as requiring modification of the syntax and parser.

The first approach, meaning that values are named, is very much like the current record implementation in *FuGL* but without the record type having a name, meaning that the current implementation can be used for implementing tuples. In the code generation phase the tuples could be transferred as both named and unnamed aggregate types in *LLVM*[54], but we consider unnamed aggregate types the easiest approach, as names are not required for tuples.

The second approach, where values are unnamed and based in indices, can be implemented much like arrays. Arrays in *FuGL* require that all elements have the same type, which is not the case with tuples, meaning that some modifications are required. We consider the version of tuples with named values to be much nicer to work with, and thus would implement that version instead of the index-based tuples.

Part IV

Testing

16. Testing Introduction

In this part we document the testing phase of the project. The testing performed in this part is summarised at the end of the part. This part covers compiler testing, programmability testing, and performance testing.

As described in Section 2.1, one task in this project is to test *FuGL* to determine the programmability of *FuGL*. To test the programmability, we perform interviews that tests various aspects of the language. The interviews are performed with two software engineering students and an IT student.

We test the performance of *FuGL* by implementing three algorithms and comparing the execution time to the execution time of the algorithms implemented in other GPU languages and frameworks. More specifically, we compare the execution time to the languages described in [1]. The three algorithms are generating permutations, K-Means, and radix sort. These algorithms are used in [1], and are used in this project to compare the performances.

17. Compiler

In this chapter, testing of the compiler is documented. Different measures and strategies have been used to manually inspect the output, and thereby determine if the compiler gives the correct output. As we are only able to give the compiler input to the best of our knowledge, this is not a guarantee that the compiler gives the correct output for any input.

Starting from the first phases, the scanner and parser have been tested by printing the AST in the console, and inspect the output. This way, scanner or parser errors can be found. For example precedence errors can be spotted in the printed AST, which would be difficult to find using other strategies. Printing the AST is done by the `PrettyPrinter` class.

The compiler features simple validation, which has been tested by trying to compile code with scope errors or missing variables or functions. Again, this is a simple and manual test. Other errors not checked in the validator are handled by *LLVM*.

The code generation phase has been tested, by programming in *FuGL* and developing different kinds of programs. By testing these programs, potential errors in the code generation can be found, as the tested program behaves differently than expected. In addition, the developed programs are saved and recompiled each time changes are made in the compiler. This way we can test for unintended changes in the compiler.

18. Developers

One of the main goals of this project is to create a language that raises the level of programmability compared to low-level languages and make it easier to utilize the GPU for developers. In order to test whether *FuGL* is understandable and easy to use, we test *FuGL* on three testers; two software engineering students, and a student with some programming knowledge.

Testing programmability is difficult as the level of programmability can not be quantified and is subjective, and whether developers understand *FuGL* depends on their knowledge of the functional paradigm, the languages that we have gained inspiration from, and GPU programming. We perform interviews with the three testers, with questions about how the language is structured, if it is understandable, and if the GPU abstractions are easy to understand and useful.

First Tester

The first tester is a software engineering student on the tenth semester. This tester has knowledge of *CUDA* and limited knowledge of the functional paradigm, as well as having implemented GPU code in *Numba*[59], a *Python* framework for GPU development. The purpose of this interview is to gain insight from someone who has previously implemented high-level GPU code and is able to compare this to how *FuGL* utilizes the GPU.

This tester stated that programmability is more important than performance, and that high-level abstractions that makes it easier to utilize the GPU are important. He further stated that if performance is very important, then performance trumps programmability.

The tester was confused by some of the syntactical elements, mostly array definitions and function type definitions. The syntax for these is taken from *Haskell* and *Swift*, meaning that developers without knowledge of these languages might not have seen this syntax before.

Due to this testers previous knowledge of the functional paradigm, he mentioned that the language has imperative features, and does not exactly fit in the functional paradigm. This comment is to be expected from developers with knowledge of the functional paradigm which *FuGL* deviates from, due to the problems with recursion, as described in Chapter 15, as well as how kernels on the GPU do not return values, but instead modifies the values in-place and copies them back to the CPU. He stated that the lack of pure, functional programming is not a problem, though, and is understandable due to how the GPU works.

The tester was not confused by how the GPU is utilized. The tester stated that the conversion from CPU to GPU code seems easy, and he has no problems understanding how the keywords work or how the thread ID is accessed and used.

Finally, the tester stated that the language seems interesting, that he would like to use the language, and that it seems more interesting than GPU development in *Numba*.

Second Tester

The second tester is also a software engineering student on the tenth semester. This tester has knowledge of the functional paradigm and GPU development. The developer mainly develops in low-level languages, and states that performance is more important than programmability, especially when utilizing the GPU, as utilizing the GPU is chosen purely for performance. Though he furthermore stated that if a solution is already implemented in a high-level language and that GPU utilization could be gained by adding a keyword, then poorer performance than *CUDA* could be ignored.

The tester noted that there are some imperative parts of the language, such as the ability for a block to have more than one expression. This is necessary due to the recursion problems described in Chapter 15. We solved these issues by adding for-loops and `void` return type, making the language more imperative. The tester understood, though, that having no return values makes sense for GPU kernels. We furthermore described why it is necessary to be able to have more than one expression in a block due to the previous issues, which the tester understod and agreed with.

The tester commented on the array syntax and the syntax for the return value of a function. The array syntax, which has the type contained within square brackets, is not like the languages he usually develops in. The syntax for the return type is a little different from *Rust* which he expected it to be like. He stated that syntax can be learned, and that this was no major problem. The tester commented on the same syntactical elements and design choices that the first tester noticed.

Finally the tester seemed to be confused on how kernels are defined and started. He understood that kernels are defined with the GPU keyword, but did not understand the usage of this keyword as both a keyword and type, and possibly a variable, depending on the usage. The usage of this keyword used in multiple ways negatively effects *orthogonality*, but we are confident that how the keyword is used can be learned. The tester suggested to allow developers to put functions into namespaces with the same syntax. This would allow developers to create functions and call them the same way that GPU functions are defined and used, but in namespaces other than the GPU namespace. Some namespace usage was considered for *FuGL*, and some parts of the namespace design matches what this tester stated, though not completely. The tester also noted that utiliz-

ing the GPU in *FuGL* requires that the developer understands how threads are structured on the GPU. This is true, but can not be avoided completely. We are convinced that by having flattened the thread and block dimensions to only require the developer to provide one number should make this easier than *CUDA*.

Third Tester

The third tester is not a software engineering student, does not have knowledge of the functional paradigm, and does not have knowledge about the GPU development. The tester knows *Java* and the object-oriented paradigm, and the reason for interviewing this developer is to test if *FuGL* is understandable for someone who has no knowledge of the paradigm or GPU development. The tester stated that the level of programmability is more important than performance in most cases. In cases where performance is part of the solution and the solution must be performant, he would not use *FuGL*. But in cases where the code is executed on the CPU and speed can be gained by adding the GPU keyword, the GPU abstractions in *FuGL* seems useful.

The tester did not understand the purpose of arrows in function declarations and function types, as he is used to the return type in front of the function. Furthermore the syntax for array type declarations was unlike his usual languages. These problems can be attributed to the tester not having knowledge of *Swift* or *Haskell*, where the syntax for function types and arrays are taken from. The tester stated that the syntax is clean and the small amount of constructs makes it easy to understand. Due to this testers previous experience with *Java*, and lack of experience in the functional paradigm, it did not seem out of place to have for-loops and functions that return void. This indicates that for developers that are not used to the functional paradigm, the language provides features they know from imperative and object-oriented languages, making it easy to use.

The tester was shown some *CUDA C* code in order to understand how *CUDA* works. This includes how memory allocations are performed on the GPU, how threads and blocks are determined and used, and the limitations on the kernels. The same code was shown in *FuGL* for both the CPU and the GPU. The tester mentioned that the GPU capabilities in *FuGL* makes it more readable than *CUDA*. This is mostly due to arrays being the same on both the CPU and GPU, as well as the GPU allowing the same functions that can be used on the CPU, such as `length` on arrays.

19. Performance

As described in Chapter 1, programmability in *FuGL* is of higher priority than good performance. This also entails that high-level abstractions are more desirable than execution time. On the other hand, GPU programming is typically done to increase performance of a program. Therefore *FuGL* must still deliver better performance on the GPU than on the CPU, though we do not expect the same performance as low-level languages for GPU development, such as *CUDA*. We expect the performance of *FuGL* to be comparable to high-level languages and frameworks, such as *F#* and *Numba*.

In this chapter, performance refers to the performance of run-time execution. The performance of the compiler is not measured in this project. We deem it irrelevant, as the compiler is an offline compiler, hence compilation is only required once for a *FuGL* solution.

To document the balance between abstraction and performance we perform a set of performance tests on *FuGL*. As described in Section 2.1, the performance of *FuGL* is compared with the implementations from [1]. We implement the same algorithms in *FuGL* as in [1] as these are already implemented in multiple languages and frameworks, thus making it straightforward to perform the tests. We only compare the execution times for the frameworks that are *CUDA*-based. The algorithms from [1] are generating permutations, K-Means and Radix sort. Machine specifications and software versions can be found in Appendix D.

19.1 Algorithms

In this section the three algorithms from [1] are briefly described.

Generating permutations is used to generate all possible combinations of characters in a given alphabet, and word length. The generation of each word is completely independent, making this algorithm easy to parallelize. To test that the algorithm is correct, we attempt to find a specific string among the different permutations.

The K-Means algorithm[60] is used to find groups among data where no other information than coordinates is known. The algorithm works in iterations, and requires some synchronization between the steps in each iteration. To test the result of the algorithm, the image tool from [1] is used to illustrate each iteration.

The Radix sort algorithm[61] is used to sort a given array of integers each digit at a time. This algorithm is not directly parallelizable, as it requires a set of buckets per thread, and a different summation strategy. To test the result of the algorithm, the sorted array is iterated once, to test if the array is in increasing order.

As the described algorithms are all mostly memory bound, this set of algorithms

is not ideal to fully test the performance of a language. Instead, an algorithm such as N-body[62] should be considered, which is an algorithm to compute gravitational force of each particle in a grid of particles. We consider the N-body problem, because the algorithm is compute bound, and can potentially run for a long time, without any memory transfer.

The reason that we do not implement N-body is that it is a complex and time consuming algorithm to build. Especially because we need to implement it in each of the languages we test. Another challenge is that in order to test the result of each implementation, we must also develop a tool that can handle this, or at least visualize the output like the implemented K-Means tool. As this task is not realistic to complete in this project, given the timeframe, this is something we choose not to implement.

19.2 Results

In this section, the results of the performance testing are presented. As testing in [1] was done on a Windows platform, and the platform for this project is Linux, all algorithms are tested again on the Linux platform. Instead of handling time measurement within each implementation, the time command in Linux is utilized, to measure a consistent and complete execution time of each implementation.

From [1] we implemented the parallel algorithms in *F#* using *Alea GPU*. Since we have changed to a Linux platform, we need to use the *.NET Core* version of *F#*. Though, we are not able to make *Alea GPU* work on the Linux platform, as there are some compatibility issues with *.NET Core* and *Alea*. Therefore we are only able to test the sequential implementations in pure *F#* on the Linux platform.

19.2.1 Parallel performance results

In this subsection the results of the parallel performance test are presented. The languages tested in parallel are *CUDA C++*, *FuGL*, *OpenACC* and *Numba*.

The first algorithm tested is the generating permutations algorithm. For this algorithm we are unable to performance test the *OpenACC* implementation from the previous project, because of the compiler being unable to parallelize the implementation. This can be attributed to either the Linux platform, or the new version of the PGI compiler. As a paid license is required to download and install old PGI compilers, we are unable to downgrade to the PGI version used in [1]. The other results can be seen on Figure 19.1.

As expected the *CUDA C++* implementation is the most efficient implementa-

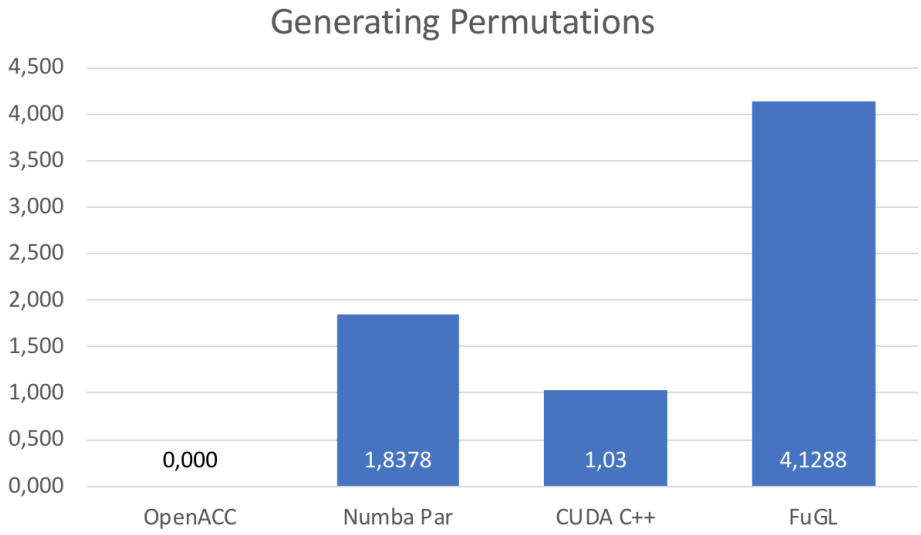

Figure 19.1: The performance results for the parallel generating germutations implementations.

tion with an average execution time of 1.03 seconds. Next is *Numba* with an average execution time of 1.8378 seconds. Last is *FuGL* with an average execution time of 4.1288 seconds. From the *FuGL* implementation to the *CUDA* implementation the difference is a 4.03X slowdown. The difference from the *FuGL* implementation to the *Numba* implementation is a 2.18X slowdown.

The second algorithm is the K-Means algorithm. On this algorithm all four languages are tested, which can be seen on Figure 19.2. The best language for this implementation is the *OpenACC* with an average execution time of 2.455 seconds, then *CUDA C++* with an average execution time of 4.92 seconds, then *Numba* with an average execution time of 7.745 seconds, and then the *FuGL* implementation with an average execution time of 10.70 seconds.

Comparing *FuGL* to the other languages in the K-Means test the slowdown for *OpenACC* is 4.36X, for *CUDA C++* 2.18X, and for *Numba* 1.38X.

The third algorithm is the Radix sort algorithm. As with K-Keans all four languages are tested, and can be seen on Figure 19.3. Again the best language is *OpenACC* with an average execution time of 1.276 seconds. Next is *CUDA C++* with an average execution time of 2.02 seconds, then *FuGL* with an average execution time of 3.26 seconds, and then *Numba* with an average execution time of 3.9318 seconds.

Comparing *FuGL* to the other languages in the Radix sort test the slowdown for *OpenACC* is 2.55X, for *CUDA C++* 1.61X. Comparing *FuGL* to the *Numba* implementation of Radix sort, a speedup of 1.21X is achieved.

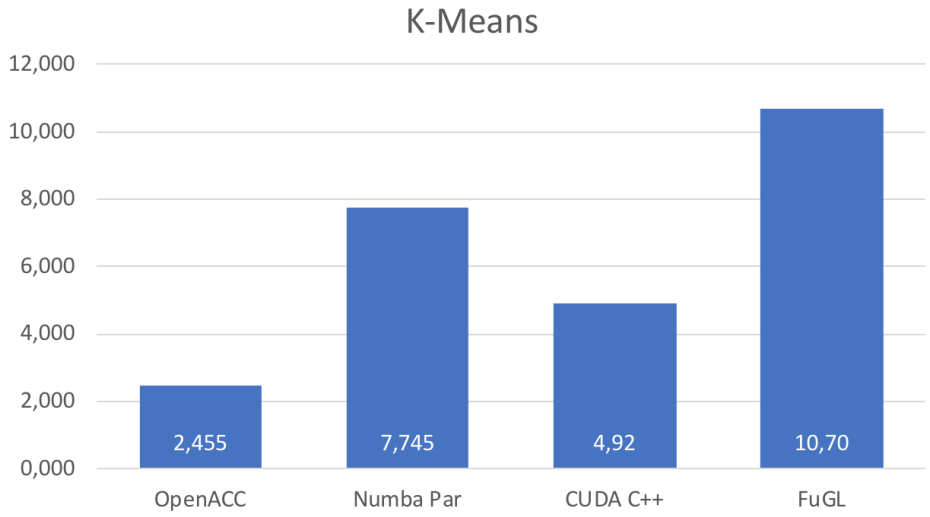

Figure 19.2: The performance results for the parallel K-Means implementations.

19.2.2 Sequential performance results

In this section, the results of the sequential performance tests are presented. The languages tested in sequential implementations are *C*, *Python*, *F#*, *FuGL* and *Numba*.

The performance results for the sequential generating permutations implementations can be seen on Figure 19.4. First is the *C* implementation with an average execution time of 46.18 seconds, then *F#* with an average execution time of 99,06 seconds, then *FuGL* with an average execution time of 206.83 seconds, then *Numba* with an average execution time of 567.79 seconds, and then *Python* with an average execution time of 1368.223 seconds.

Comparing the *FuGL* implementation with the other generating permutations implementations, going from *C* to *FuGL* would give a 4.48X slowdown, and from *F#* a 2.09X slowdown. Going from *Numba* to *FuGL* would give a 2.75X speedup, and from *Python* a 6.62X speedup.

The performance results for the sequential K-Means implementations can be seen on Figure 19.5. First is the *C* implementation with an average execution time of 7.70 seconds, then *Numba* with an average execution time of 9.49 seconds, then *FuGL* with an average execution time of 54.21 seconds, then *F#* with an average execution time of 113.08 seconds, and then *Python* with an average execution time of 1869.31 seconds.

Comparing the *FuGL* implementation with the other K-Means implementations, going from *C* to *FuGL* would give a 7.04X slowdown, from *Numba* a 5.71X slowdown. Going from *F#* to *FuGL* would give a 2.09X speedup, and from *Python* a

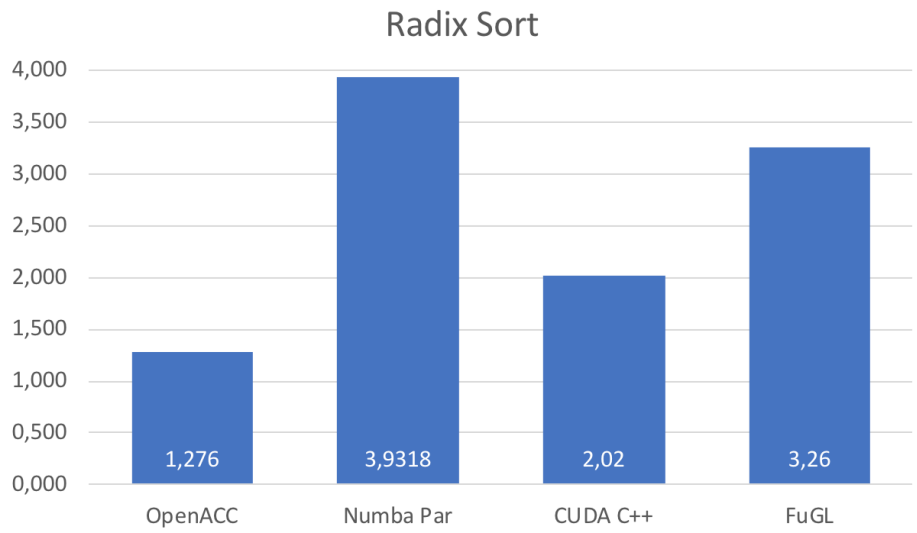

Figure 19.3: The performance results for the parallel Radix Sort implementations.

174.64X speedup.

The performance results for the sequential Radix sort implementations can be seen on Figure 19.6. First is the *C* implementation with an average execution time of 4.47 seconds, then *FuGL* with an average execution time of 5.57 seconds, then *Numba* with an average execution time of 14.31 seconds, then *F#* with an average execution time of 50.66 seconds, and then *Python* with an average execution time of 385.24 seconds.

Comparing the *FuGL* implementation with the other K-Means implementations, going from *C* to *FuGL* would give a 1.25X slowdown. Going from *Numba* to *FuGL* would give a 2.57X speedup, from *F#* a 9.10X speedup, and from *Python* a 118.35X speedup.

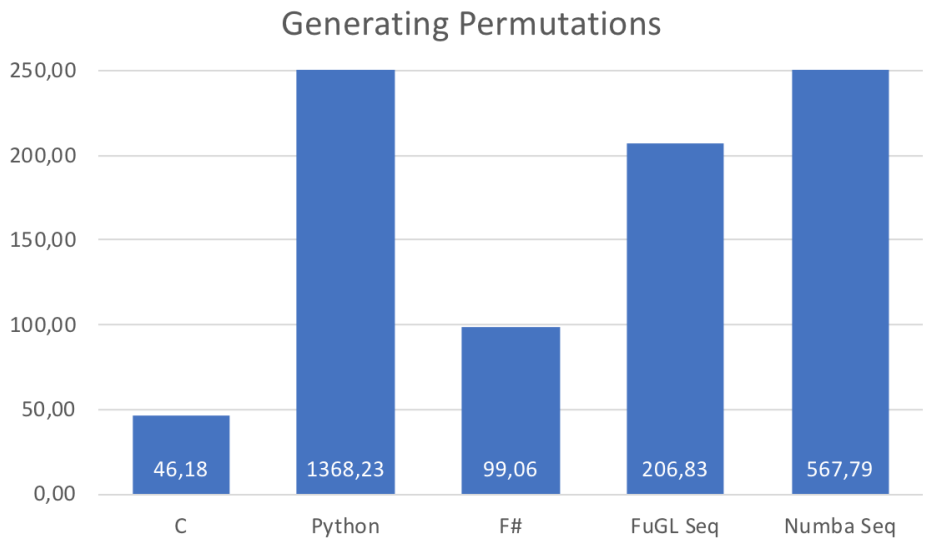

Figure 19.4: The performance results for the sequential generating permutations implementations.

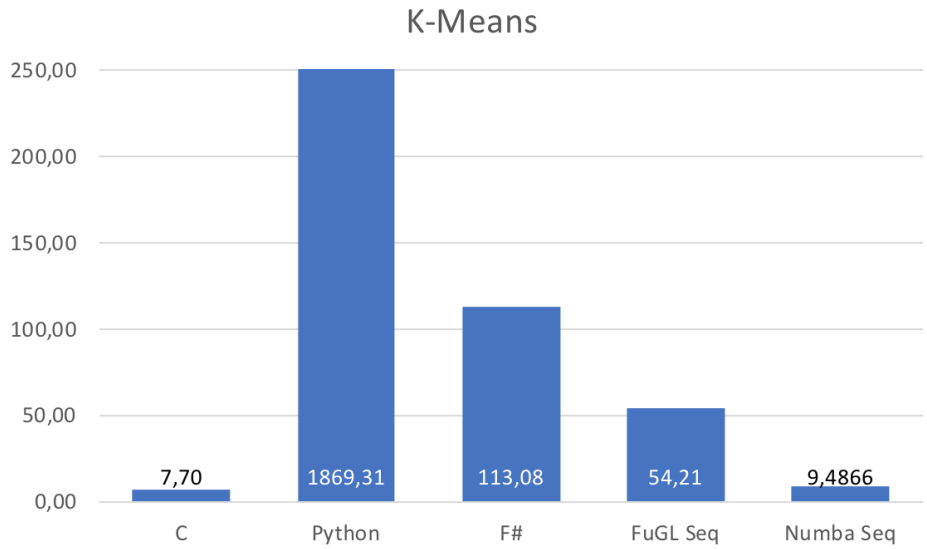

Figure 19.5: The performance results for the sequential K-Means implementations.

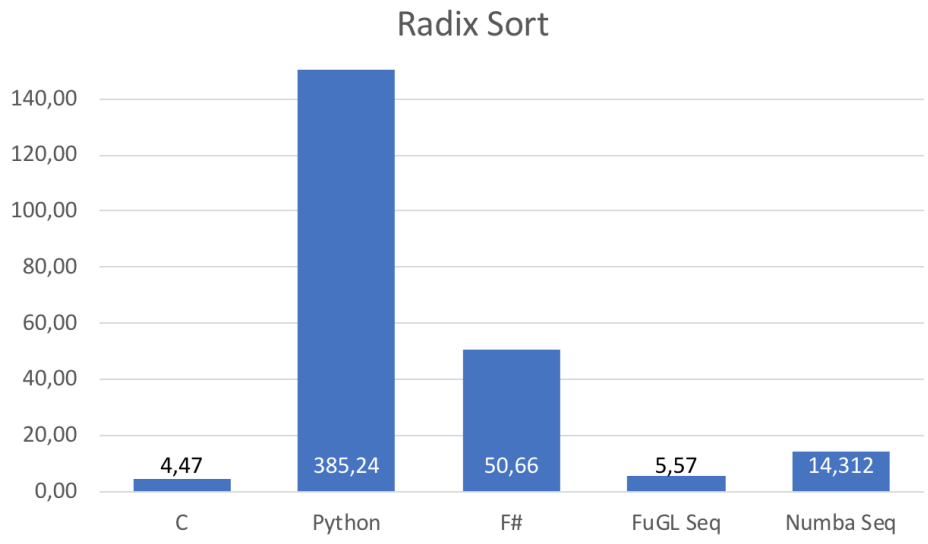

Figure 19.6: The performance results for the sequential Radix Sort implementations.

20. Testing Summary

This concludes the testing part of the report. We tested the performance of *FuGL* on both the CPU and GPU by implementing three algorithms, generating permutations, K-Means, and Radix sort, and compared the performance to implementations from [1]. Furthermore we tested the programmability of *FuGL* by performing three interviews, in an attempt to determine whether developers understand, and are able to use *FuGL* for CPU and GPU development.

20.1 Performance

FuGL was tested by implementing the three algorithms from [1], and comparing the execution times with the implementations from [1]. The comparison on the parallel implementations can be seen in Table 20.1, and the sequential ones in Table 20.2.

In the parallel implementations the majority of the other implementations are faster than *FuGL*, ranking it badly in terms of parallel performance. The reason for this is that two of the languages, *CUDA C++* and *OpenACC*, are *C* type languages with low level abstractions, but high performance. Therefore these results are expected. The best comparable language in terms of performance is *Numba*, with the lowest slowdown factors on the algorithms. On the Radix sort algorithm, *FuGL* actually performs better than *Numba* with a factor of 1.21.

	Generating Permutations	K-means	Radix Sort
CUDA C++	4.03 ⇓	2.18 ⇓	1.61 ⇓
OpenACC		4.36 ⇓	2.55 ⇓
Numba	2.25 ⇓	1.38 ⇓	1.21 ⇑

Table 20.1: Speedup (⇑) or Slowdown (⇓) factors in the parallel implementations between the presented languages and *FuGL*.

Comparing sequential execution times, *FuGL* performs better than many of the other sequential implementations. As *Python* is an interpreted language, the very high speedup factors are expected, and not a fair comparison, as *FuGL* is a compiled language and *Python* is not. Also, as expected, *C* performs better than *FuGL* in any algorithm. Comparing *FuGL* with *F#* and *Numba*, both speedups and slowdowns are seen, which is a satisfying result, as these languages are compiled languages.

Overall, the performance of *FuGL* is acceptable considering the focus of *FuGL* and project. The main focus has been programmability, and therefore no optimization steps are implemented to increase performance. Though, *FuGL* still outperforms the *Numba* parallel Radix sort implementation, and several sequential implementations. Of this reason we find the performance of *FuGL* acceptable.

	Generating Permutations	K-means	Radix Sort
C	4.48 ⇓	7.04 ⇓	1.25 ⇓
F#	2.09 ⇓	2.09 ⇑	9.10 ⇑
Numba	2.75 ⇑	5.71 ⇓	2.57 ⇑
Python	6.62 ⇑	174.64 ⇑	118.35 ⇑

Table 20.2: Speedup (⇑) or Slowdown (⇓) factors in the sequential implementations between the presented languages and *FuGL*.

20.2 Programmability

The programmability testing was performed by interviewing three developers and showing the features of *FuGL*. All testers had some issues with some syntactical elements, which we attribute to their lack of knowledge about s which we have gained inspiration from, mainly *Haskell* and *Swift* for the elements that confused the testers. Syntax can be learned, and we do not see these issues as very impactful. The usage of non-functional features, such as using `Void` as a return value is a problem. Some issues arising from this could be corrected by having recursion, as was intended. The main issues with not allowing *void* as a return value is how kernels return values from the GPU. For this case, we do not have a solution yet.

Generally the testers seemed to agree on the assumption that programmability is more important than performance, though they all stated that if performance is a requirement for the solution, then performance trumps programmability. They stated that if a solution was already implemented in *FuGL* and some performance could be improved by using the `gpu` keyword then programmability is more important than performance. They furthermore agreed that GPU development seems easy in *FuGL* and that the criterion from [1], *sequential to parallel* is high in *FuGL*. This criterion refers to how much work is required in order to turn a sequential implementation of a program to a parallel implementation that can run on the GPU. This is one of the things that we have improved in *FuGL*, which the testers agrees with.

Regarding programmability, the testers seemed to find the low number of constructs easy to understand, and they all agreed that converting code from the CPU to the GPU seemed easy. We are happy with the results from the interviews.

Part V

Summary

21. Reflection

This chapter contains reflections of the work performed in this project, including the development process used in the project, decisions taken early in the project, and the consequences of these decisions.

As described in Chapter 2, the development process for this project is based on the waterfall model, where we finish a phase before starting the next. The implementation phase was performed iteratively, meaning that we implement and improve during the entire implementation, allowing us to prioritize features during development. The iterative model allows us to implement the entire CPU part of the code generator before implementing the layer between the CPU and GPU, and then finally the GPU layer. This was easy, as the base parts were working, thus we only had to modify the parts that are not the same on the CPU and GPU, as well as transferring information to and from the GPU.

The waterfall approach lead to some problems as we did not test early in the project, which caused us to not discover the issues with recursion until late in the implementation phase. As *FuGL* is a functional language it needs recursion, but as that did not work, we had to redesign a large part of the language. Had we tested recursion in *LLVM* earlier, this could have potentially been avoided, and if not solvable, we could have designed a better workaround than the currently implemented `for`-loops, which led to *FuGL* being more imperative than intended. The reason for the design issues is most likely that we went through the design phase too fast as we knew there was a lot of work to do.

To get started with *FuGL* we chose an existing syntax from the *Expressions* language[30]. The syntax from *Expressions* is not directly copied, but it is the syntactic foundation of *FuGL*. Choosing *Expressions* as foundation was done to get started quickly, instead of spending a long time designing a completely new syntax. This worked out well, and resulted in the project taking off quickly.

LLVM was chosen as backend language, as it is used by many other languages and frameworks, including *Julia*, *Rust*, *Alea*, *Accelerate* and *Numba*. As experienced in [1], almost any high level language for the GPU utilizes *LLVM* and as such we also chose it for *FuGL*. This gives the advantage of having the same backend for both CPU and GPU code. *LLVM* worked out very well in the project, although a better documentation of *LLVM* would have made development easier.

To develop the *FuGL* compiler, we choose *C#* and the *.NET Core* framework, as we wanted a high-level object-oriented language. *.NET Core* is a good choice, as it is cross platform and it features many high level abstractions and furthermore interfaces well with the *LLVM C library* using the *LLVMSharp* package.

Implementing a compiler was a larger task than anticipated, even though we did anticipate a lot of work. Many features may seem minor or trivial, but require a lot of design and implementation in the compiler. Some design choices we had taken early on, such as delimiting lambda captures, were later found to be

poor decisions, and had to be reconsidered. We mostly found these issues when we began implementing the algorithms in *FuGL*, though we should have caught them earlier. Captures, `for`-loops, and `Void` as return type, are the three biggest features that were designed and implemented late in the project. `for`-loops and `Void` as return are necessary due to the imperative constructs in *FuGL* and due to how the GPU does not return values but instead modifies them in-place.

22. Conclusion

GPU development is difficult, and developers are often required to develop in *C*-like languages which lack high-level abstractions, where they must handle memory, threads and blocks themselves. The developers need to understand how all the concepts of GPU and low-level CPU development ties together and the limits of the GPU, in order to utilize the GPU. The purpose of this project was to create a high-level, functional language that can hide many of these details from the developer, providing high-level abstractions that are usable on both the CPU and the GPU.

In this project we designed and implemented *FuGL*, a high-level, statically typed, functional programming language with primitives for GPU development. *FuGL* attempts to solve some of the programmability issues we found in [1], and allows developers to utilize the GPU without frameworks or libraries, by having primitives in the language that help developers parallelize algorithms. The *FuGL* compiler is implemented in *C#* using *LLVM* as the backend. *FuGL* was tested by interviewing three testers to determine whether the level of programmability is high, and the performance is tested using the algorithms described in [1]; generating permutations, K-Means, and Radix sort.

The project started with the initial problem statement:

How can we develop a solution that allows developers to write efficient GPU code at a higher level of abstraction than OpenCL or CUDA, while maintaining the possibilities that a low-level language provides?

The problem statement led to a number of tasks to be performed, in order to design and implement *FuGL* and its compiler. The tasks completed are as follows:

- Determine the type of solution to implement, either a language or library.
- Analyze various frameworks and libraries for high-level GPU development.
- Determine the technical aspects of the compiler, including the development language and backend.
- Design how the constructs in the language are mapped to *LLVM* constructs.
- Implement the initial phases of the compiler, including scanning and parsing, validation, and some code generation preparation phases.
- Implement CPU code generation, implement mappings for the CPU to GPU layer, and implement the CPU code generation.
- Test the performance and programmability of *FuGL*.

Due to the findings from [1], we decided that a functional programming language for GPU development, which is easy to use and with high-level abstractions, could make it easier for developers to utilize the GPU, without having to learn about architectures, thread and block handling, and low-level languages.

We analyzed a number of frameworks for high-level GPU development, including *Accelerate*, *Alea*, and *Thrust* to determine how others have designed high-level GPU utilization. These frameworks, along with multiple languages we know and have used, made the foundation for the syntax and design of *FuGL*.

We designed *FuGL* with few constructs, in order to increase the *simplicity*. The constructs are similar to the constructs known from *Lisp*, which allows variable assignments, anonymous functions, boolean conditions, and function calls. We furthermore added records to *FuGL*, to make it easier to store data in a structured way. The syntax is inspired by many languages, mainly *C*-like languages and functional languages. We designed the GPU utilization to work by using keywords to denote which parts of a program that should run on the GPU, and we allow the developer to choose how to copy memory to and from the GPU using only keywords. Testing showed that by only adding the `gpu` keyword, and not modifying anything else, in the sequential K-Means implementation, the algorithm runs at half the execution time. We designed the language to use the same functions and syntax on both the CPU and GPU, making sure that using the GPU resembles developing on the CPU, increasing the *orthogonality* and *Host/kernel similarity*[1].

The *FuGL* compiler is implemented in *C#*, using *LLVM* as the backend for both the CPU and GPU code. The compiler is cross-platform due to it being based on *.NET Core*. To utilize the *LLVM C library* we use the *LLVMSharp* package, which adds *C#* bindings to the library.

Interviews were performed to test whether *FuGL* is easy to understand for developers. The interviews indicate that the testers were able to understand how *FuGL* works, and how the GPU is utilized. The testers all noted that some of the syntactical elements were confusing, but we attribute this to them not knowing the languages we designed the syntax after. Besides these elements, the testers seemed to understand the purpose and idea of *FuGL*. The testers stated that utilizing the GPU in *FuGL* seems easy. They furthermore mostly agreed on our initial assumption that programmability is more important than performance, but in some cases they would rather use a low-level language in order to gain better performance.

Performance testing was done, in order to show how *FuGL* compares to the languages and frameworks described in [1]. We implemented the algorithms generating permutations, K-Means, and Radix sort, and compared the execution times, both sequential and parallel, to the execution times of the implementations found in [1]. As expected, *FuGL* performs much like the higher-level languages in [1] on the CPU, but is slightly slower on the GPU. This performance is satisfying, even though *FuGL* is mostly slower on the GPU compared to the other languages tested. *FuGL* requires less code rewriting in order to utilize the GPU, and we have not performed any optimizations in the compiler or spent time tweaking the *LLVM* assembly generated by the compiler. We delimited op-

timizations early, and we are confident that some performance could be gained if we optimized the generated code and corrected some of the bottlenecks, such as lambda captures, as described in Section 15.1.

Due to our lack of testing the initial design, some issues were found during the implementation phase. This led to *FuGL* not being able to use recursion very well, making us implement `for`-loops that can be used instead. This breaks the functional paradigm, but was necessary in order for the compiler and functions to work properly. However, due to how the GPU works, *FuGL* would still need to be able to modify arrays and values in-place, meaning that some imperative-like code would have been necessary. While this is unfortunate, we are still content with the programmability of *FuGL*, and that *FuGL* makes it easier for developers to utilize the GPU by using built-in primitives. In comparison to low-level languages for GPU development, *FuGL* contains high-level abstractions, built-in memory handling, easy copying to and from the GPU, and the language is the same on the GPU and CPU.

We conclude that while *FuGL* is not a pure, functional programming language, it is a high-level language which provides high-level constructs and abstractions that allow developers to easily utilize the GPU. The performance is acceptable and as we expected, meaning that the performance is worse than the low-level languages, but matches the high-level languages. *FuGL* is mostly faster on the CPU, but mostly slower on the GPU.

23. Future Work

This chapter covers some topics that should be changed or implemented in a future version of *FuGL*. Some future work was described in Chapter 15.1, but as technical suggestions about new functionality. In this chapter, more general or abstract ideas are presented.

First are the topics described in Chapter 7.2, which are not implemented in this project. These topics cover pattern matching, partial application and list comprehensions. Although list comprehensions are implemented as a function not a construct, all the topics could be implemented in a future version of *FuGL*.

In addition to the delimitations made in Chapter 7.2, some extra design choices were made to further delimit some features. These are described in Chapter 11 and cover validation and optimization, *OpenCL* support, optimized thread and block dimensions, and fat binaries.

As a complete validation phase is missing from the compiler, the compiler may fail on some input, if it is not handled by the *LLVM* validator. Validation is implemented in some areas, as *LLVM* is for example not aware of scope rules. In a future version of *FuGL* all validation should be done by the *FuGL* compiler in order to provide better feedback on errors.

As previously described, the *FuGL* compiler has no optimizing phase, and in addition the different *LLVM* constructs and libraries have not been optimized either. In order to achieve better performance results, especially on the parallel algorithms, this is an important task to complete in a future version.

As *FuGL* handles recursion unexpectedly bad, this is an area that we should investigate more in detail. Being unable to support recursion is not very functional, and conflicts with the paradigm of *FuGL*. Because recursion is badly supported in *FuGL* the for-loop was implemented. This should also be reconsidered. For example it could be changed to a for loop that returns an array, like it is done in *Sisal*. This was described in detail in Chapter 5.7.

Another improvement to the functional aspect of *FuGL* is how data is returned from kernels. In the current version of *FuGL* data must be returned through parameters to the kernel. This is very imperative, which conflicts with functional programming. Instead values could be returned from kernels like a regular function with return values. This feature would greatly improve functional GPU programming.

Bibliography

[1] Andreas Steen Andersen & Christian Lundtofte Sørensen & Henrik Djernes Thomsen & Jonathan Hastrup & Morten Mandrup Hansen & Thomas Heldbjerg Bork. "A Comparative Study of Programming Languages for the GPU". Aalborg University, 2018. URL: http://projekter.aau.dk/projekter/da/studentthesis/a-comparative-study-of-programming-languages-for-the-gpu(172234d3-74c5-4130-bdb0-57cdd0d71f79).html.

[2] Robert W. Sebesta. *Concepts of Programming Languages*. 10th. Pearson, 2012. ISBN: 0273769103, 9780273769101.

[3] TIOBE software BV. *TIOBE Index for February 2018*. Seen 26/02/2018. URL: https://www.tiobe.com/tiobe-index/.

[4] Wikipedia. *First-class function - Language support*. Seen 26/02/2018. URL: https://en.wikipedia.org/wiki/First-class_function#Language_support.

[5] Wikipedia. *Higher-order function - Direct Support*. Seen 26/02/2018. URL: https://en.wikipedia.org/wiki/Higher-order_function#Direct_support.

[6] Wikipedia. *Anonymous function - Examples*. Seen 26/02/2018. URL: https://en.wikipedia.org/wiki/Anonymous_function#Examples.

[7] Wikipedia. *Immutable object - Language-specific details*. Seen 26/02/2018. URL: https://en.wikipedia.org/wiki/Immutable_object#Language-specific_details.

[8] J. Nickolls and W. J. Dally. "The GPU Computing Era". In: *IEEE Micro* 30.2 (2010), pp. 56–69. ISSN: 0272-1732. DOI: 10.1109/MM.2010.41.

[9] QuantAlea. *Alea GPU*. Seen 14/02/2018. URL: http://www.aleagpu.com/.

[10] Nathaniel Nystrom, Derek White and Kishen Das. "Firepile: run-time compilation for GPUs in scala". In: *ACM SIGPLAN Notices*. Vol. 47. 3. ACM. 2011, pp. 107–116.

[11] Eric Holk et al. "GPU programming in rust: Implementing high-level abstractions in a systems-level language". In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, pp. 315–324.

[12] NVIDIA. *cuBLAS*. Seen 15/02/2018. URL: http://docs.nvidia.com/cuda/cublas/index.html.

[13] NVIDIA Jared Hoberock Nathan Bell. *Thrust - Parallel Algorithms Library*. Seen 15/02/2018. URL: http://thrust.github.io.

[14] Tim Besard, Christophe Foket and Bjorn De Sutter. "Effective Extensible Programming: Unleashing Julia on GPUs". In: *arXiv preprint arXiv:1712.03112* (2017).

[15] julialang.org. *Julia*. Seen 01/03/2018. URL: https://julialang.org/.

[16] Mozilla. *The Rust Programming Language*. Seen 09/03/2018. URL: https://www.rust-lang.org.

[17] Manuel M. T. Chakravarty. *Accelerate*. Seen 05/03/2018. URL: http://www.acceleratehs.org/documentation.html.

[18] haskell.org. *Haskell*. Seen 10/03/2018. URL: https://www.haskell.org.
[19] Manuel M. T. Chakravarty. *Accelerate: Examples*. Seen 05/03/2018. URL: http://www.acceleratehs.org/examples.html.
[20] Manuel M. T. Chakravarty. *Data.Array.Accelerate*. Seen 05/03/2018. URL: https://hackage.haskell.org/package/accelerate-1.1.1.0/docs/Data-Array-Accelerate.html.
[21] Department of Computer Science at the University of Copenhagen. *What is Futhark? | Futhark - A High Performance Functional Array Language*. Seen 06/03/2018. URL: https://futhark-lang.org/.
[22] Department of Computer Science at the University of Copenhagen. *Gotta Go Fast!* Seen 13/03/2018. URL: https://futhark-lang.org/performance.html.
[23] Microsoft. *C# Guide*. Seen 09/03/2018. URL: https://docs.microsoft.com/en-us/dotnet/csharp/index.
[24] F# Software Foundation. *F# Software Foundation*. Seen 09/03/2018. URL: http://fsharp.org/.
[25] NVIDIA Jared Hoberock Nathan Bell. *Thrust - Parallel Algorithms Library*. Seen 09/03/2018. URL: http://thrust.github.io.
[26] Jean-Luc Gaudiot & Tom DeBoni & John Feo & Wim Bohm & Walid Najjar & Patrick Miller. *The Sisal Project: Real World Functional Programming*. Seen 05/03/2018. URL: https://pdfs.semanticscholar.org/8f92/f18911bd64f4ef52d7e5de97eabbc98e8ca7.pdf.
[27] John Feo. *8. Loops and Parallelism*. Seen 20/03/2018. URL: http://www2.cmp.uea.ac.uk/~jrwg/Sisal/08.Loops.par.html.
[28] Apple Inc. *Swift.org - Welcome to Swift.org*. Seen 10/03/2018. URL: https://swift.org.
[29] The Common Lisp Foundation. *Welcome to Common-Lisp.net*. Seen 10/03/2018. URL: https://common-lisp.net.
[30] Christian Lundtofte Sørensen. *Expressions*. Seen 16/02/2018. URL: https://github.com/krillere/expressions.
[31] multiple sources. Wikipedia. *Partial Application - Wikipedia*. Seen 15/05/2018. URL: https://en.wikipedia.org/wiki/Partial_application.
[32] Marc Moreno Maza. *LL(1) grammars*. Seen 16/02/2018. URL: http://www.csd.uwo.ca/~moreno/CS447/Lectures/Syntax.html/node14.html.
[33] Cppreference. *C++ Operator Precedence - cppreference.com*. Seen 01/03/2018. URL: http://en.cppreference.com/w/cpp/language/operator_precedence.
[34] NVIDIA. *CUDA LLVM Compiler*. Seen 16/02/2018. URL: https://developer.nvidia.com/cuda-llvm-compiler.
[35] ?? *User Guide for AMDGPU Backend*. Seen 16/02/2018. URL: https://llvm.org/docs/AMDGPUUsage.html.
[36] LLVM. *LLVM FAQ - I'd like to write a self-hosting LLVM compiler. How should I interface with the LLVM middle-end optimizers and back-end code gen-*

erators? Seen 22/02/2018. URL: https://llvm.org/docs/FAQ.html#id-like-to-write-a-self-hosting-llvm-compiler-how-should-i-interface-with-the-llvm-middle-end-optimizers-and-back-end-code-generators.

[37] Microsoft. *LLVM bindings for .NET and Mono, written in C# using ClangSharp*. Seen 22/02/2018. URL: https://github.com/Microsoft/LLVMSharp.

[38] LLVM. *LLVM-C: C interface to LLVM*. Seen 16/02/2018. URL: http://llvm.org/doxygen/group__LLVMC.html.

[39] Yi Yang et al. "Fixing performance bugs: An empirical study of open-source GPGPU programs". In: *Parallel Processing (ICPP), 2012 41st International Conference on*. IEEE. 2012, pp. 329–339.

[40] NVIDIA Mark Harris. *How to Optimize Data Transfers in CUDA C/C++*. Seen 18/12/2017. URL: https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/.

[41] NVIDIA Mark Harris. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels*. Seen 18/12/2017. URL: https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/.

[42] Microway. *GPU Memory Types – Performance Comparison*. Seen 22/02/2018. URL: https://www.microway.com/hpc-tech-tips/gpu-memory-types-performance-comparison/.

[43] NVIDIA. *CUDA Programming :: CUDA Toolkit Documentation*. Seen 01/03/2018. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/#features-and-technical-specifications.

[44] Victor W. Lee et al. "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU". In: *SIGARCH Comput. Archit. News* 38.3 (2010), pp. 451–460. ISSN: 0163-5964. DOI: 10.1145/1816038.1816021. URL: http://doi.acm.org/10.1145/1816038.1816021.

[45] NVIDIA Corporation. *CUDA C Best Practices Guide*. Seen 01/03/2018. URL: http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#differences-between-host-and-device.

[46] NVIDIA Corporation. *NVIDIA CUDA Compiler Driver NVCC*. Seen 16/03/2018. URL: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[47] Khronos Group. *SPIR Overview*. Seen 05/04/2018. URL: https://www.khronos.org/spir/.

[48] Maratyszcza. *OpenCL offline compiler - Github*. Seen 20/03/2018. URL: https://github.com/Maratyszcza/clcc.

[49] KhronosGroup. *LLVM/SPIR-V Bi-Directional Translator - github*. Seen 20/03/2018. URL: https://github.com/KhronosGroup/SPIRV-LLVM.

[50] Microsoft. *LLVMSharp*. Seen 09/03/2018. URL: https://github.com/Microsoft/LLVMSharp/issues.

[51] LLVM. *User Guide for NVPTX Back-end*. Seen 08/03/2018. URL: https://llvm.org/docs/NVPTXUsage.html.

[52] Paul Smith. *How to get started with the LLVM C API*. Seen 18/03/2018. URL: https://pauladamsmith.com/blog/2015/01/how-to-get-started-with-llvm-c-api.html.

[53] IBM Arpan Sen. *Create a working compiler with the LLVM framework, Part 1*. Seen 18/03/2018. URL: https://www.ibm.com/developerworks/library/os-createcompilerllvm1/.

[54] LLVM. *LLVM Language Reference Manual*. Seen 17/03/2018. URL: https://llvm.org/docs/LangRef.html.

[55] Rib Rix-2. *Returning a structure*. Seen 17/03/2018. URL: http://llvm.1065342.n5.nabble.com/Returning-a-structure-td40519.html.

[56] Michael Nicolella. *[llvm-dev] C returning struct by value*. Seen 17/03/2018. URL: https://groups.google.com/forum/#!topic/llvm-dev/RSnV-Vr17nI.

[57] Rodney M. Bates. *[LLVMdev] Another struct-return question*. Seen 17/03/2018. URL: http://lists.llvm.org/pipermail/llvm-dev/2015-January/080886.html.

[58] NVIDIA. *NVIDIA CUDA Driver API*. Seen 09/04/2018. URL: http://docs.nvidia.com/cuda/cuda-driver-api/index.html.

[59] Siu Kwan Lam, Antoine Pitrou and Stanley Seibert. "Numba: A llvm-based python jit compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM. 2015, p. 7.

[60] Andrea Trevino. *Introduction to K-means Clustering*. Seen 17/05/2018. URL: https://www.datascience.com/blog/k-means-clustering.

[61] Nadathur Satish, Mark Harris and Michael Garland. "Designing Efficient Sorting Algorithms for Manycore GPUs". In: *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161005. URL: https://doi.org/10.1109/IPDPS.2009.5161005.

[62] Michele Trenti and Piet Hut. *N-body simulations (gravitational)*. Seen 17/05/2018. URL: http://www.scholarpedia.org/article/N-body_simulations_(gravitational).

Part VI

Appendix

```
/*
Based on: http://matt.might.net/articles/grammars-bnf-ebnf/
{ } = Repeat 0 or more times
[ ] = Optional
*/

// Program. Consists of multiple namespaces
program := namespace { namespace }

// Namespace. Consists of multiple types and functions
namespace       := "namespace" name "{" namespaceDecl "}"
namespaceDecl := funcDecl | funcDecl program | typeDecl | typeDecl ↩
    program

// Function declarations
funcDecl        := [ "gpu" ] "func" name funcParsRet block
funcParsRet     := "(" typeNamePairs "->" name ")"

// Type declaration
typeDecl        := "type" name typeBlock
typeBlock       := "{" typeIdentifierPairs "}"

// Type and Name pairs
typeIdentifierPairs     := typeIdentifierPair { "," typeNamePairs }
typeIdentifierPair      := name name

// Function calls
funcCall        := name "(" [ funcCallPars ] ")"
funcCallPars    := expr { "," funcCallPars }

// Lambda (Same syntax as function declaration, though different ↩
    semantics in the compiler)
lambdaExpr      := "func" funcParsRet block

// Misc
block := '{' exprs '}'

// Expressions
exprs := expr { exprs }
expr := "(" expr ")"
        | expr op expr
        | op expr
        | name
        | stringLit
        | charLit
        | ifExpr
        | boolLit
        | numberLit
        | funcCall
        | letExpr
        | lambda
        | arrayLit

// If expression
ifExpr          := "if" expr block { elseIfCase } "else" block
elseIfCase      := "elseif" expr block

// Let (Variable assignments)
letExpr         := "let" varDecls block
varDecls        := varDecl { "," varDecls }
varDecl         := [ 'mutable' ] name name "=" expr | name "=" expr
```

```
arrayLit          := "[" exprs "]"
arrayLitExpr      := { "[" } expr { "]" }
typeDecl          := "(" name { "," name } "->" name ")"
numberLit         := (0-9)+ | (0-9)+.(0-9)+
boolLit           := "true" | "false"
stringLit         := """ (.*) """
charLit           := "'" (ASCII symbols | \ ASCII symbols) "'"
name              := (a-Z)(a-Z0-9_)*
op                := "and" | "or" | "!" | "<" | ">" | "<=" | ">=" | "↩
    ==" | "!=" | "*" | "/" | "+" | "-" | "." | "++" | "="

// Comments are ignored by the scanner
// Singleline comments stops at newline, multiline stops when the ↩
    stop characters are met
singlelineComment       := "//" (.*) "\n"
multilineComment        := "/*" (.*) "*/"
```

Listing A.1: EBNF for the *FuGL* language.

```
// Numbers
func min(Int a, Int b -> Int)
func max(Int a, Int b -> Int)
func even(Int a -> Bool)
func odd(Int a -> Bool)

func fCeil(Float f -> Float)
func dCeil(Double d -> Double)
func fFloor(Float f -> Float)
func dFloor(Double d -> Double)
func fPow(Float x, Float y -> Float)
func dPow(Double x, Double y -> Double)


// Arrays
func get([A] lst, Int64 index -> A)
func set(mutable [A] lst, Int64 index, A newVal -> Void)
func length([A] lst -> Int64)
func append([A] lst, A elem -> [A])
func push(mutable [A] lst, A elem -> Void) // In-place append (Internal usage!)
func list(Int64 start, Int64 end, (Int64 -> A) fnc -> [A])

func empty([A] lst -> Bool)

func first([A] lst -> A)
func last([A] lst -> A)

func tail([A] lst -> [A]) // All but first element
func init([A] lst -> [A]) // All but last element

func map([A] lst, (A -> B) fnc -> [B])
func filter([A] lst, (A -> Bool) fnc -> [A])
func zipWith([A] lstA, [B] lstB, (A, B -> C) fnc -> [C])

// Imperative..
func for(Int64 start, Int64 end, (Int64 -> Void) fnc -> Void)
func forDown(Int64 start, Int64 end, (Int64 -> Void) fnc -> Void)

// Casting
func toBool(A a -> Bool)
func toChar(A a -> Char)
func toInt(A a -> Int)
func toInt64(A a -> Int64)
func toFloat(A a -> Float)
func toDouble(A a -> Double)
```

Listing B.1: CPU Standard library functions in *FuGL*.

```
// Arrays
gpu func get([A] lst, Int64 index -> A)
gpu func set(mutable [A] lst, Int64 index, A newVal -> Void)
gpu func length([A] lst -> Int64)

// Imperative..
gpu func for(Int64 start, Int64 end, (Int64 -> Void) fnc -> Void)
gpu func forDown(Int64 start, Int64 end, (Int64 -> Void) fnc -> Void)

// Casting
gpu func toBool(A a -> Bool)
gpu func toChar(A a -> Char)
```

```
13 gpu func toInt(A a -> Int)
14 gpu func toInt64(A a -> Int64)
15 gpu func toFloat(A a -> Float)
16 gpu func toDouble(A a -> Double)
17 gpu func toString(A a -> String)
```

Listing B.2: GPU Standard library functions in *FuGL*.

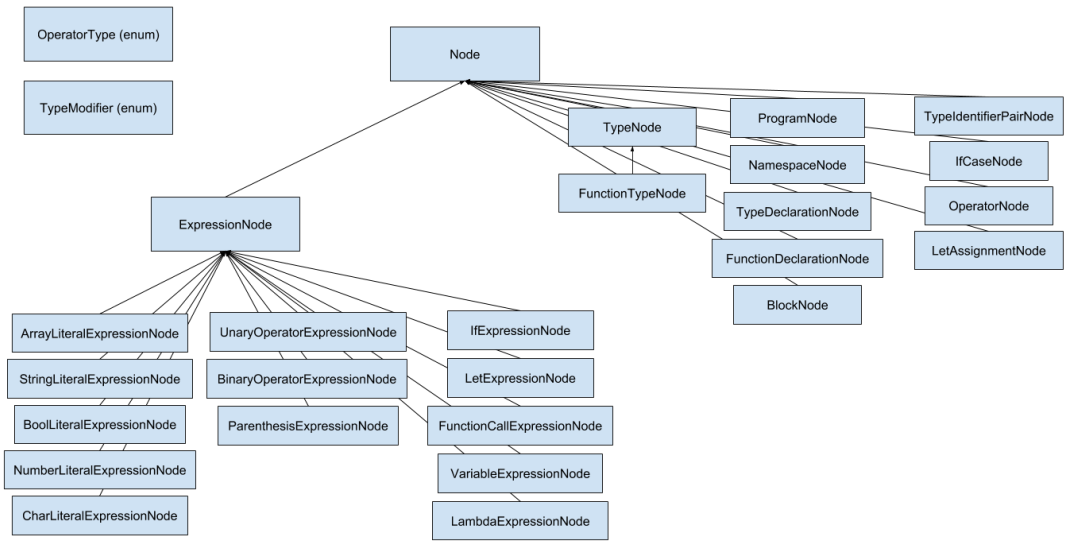

Figure C.1: Classes used in building the AST.

D. Test Machine Specifications

The hard- and software used for the performance tests.

Hardware:

Operating System: Ubuntu 17.10

CPU: Intel Core i7-920

GPU: NVIDIA Geforce GTX 1070 (Zotac GTX 1070 Mini)

Tools and software:

OpenACC compiled with: PGI Compiler, pgcc 18.4-0 64-bit (Supporting *OpenACC* version 2.5)

CUDA compiled with: NVIDIA CUDA Compiler, nvcc 8.0.61 (supporting *CUDA* version 8)

F# compiled with: Dotnet core 'dotnet build', version 2.1.103

Python runtime: 3.6.5

NumPy library version: 1.14.2

Numba library version: 0.37.0