YaleNUSCollege

Generating Cyclo-Static Dataflow Graphs from Lift for FPGA Programming

Hebe Hilhorst

Capstone Final Report for BSc (Honours) in

Mathematical, Computational and Statistical Sciences

Supervised by: Bruno Bodin

AY 2019/2020

YaleNUSCollege

YALE-NUS COLLEGE

Abstract

Department of Mathematical, Computational and Statistical Sciences

B.Sc (Hons)

Generating Synchronous Dataflow Graphs from Lift for FPGA programming

by Hebe HILHORST

Parallel computing is increasingly necessary for high-performance programming, but can be difficult to successfully implement. Lift is an exciting new high-level language designed to make it easier to code performant and error-free parallel programs. However, performance portability requires separate compilers to make Lift accessible to different hardware platforms. This paper proposes a novel approach for generating parallel FPGA code from Lift programs using Cyclostatic Dataflow Graphs (CSDFs) as an intermediate representation. It tests this design in practice with the implementation of an automatic generator for creating CSDFs from Lift programs, allowing for preemptive analysis and scheduling optimizations. CSDFs for common operations are generated, and different optimizations tested and analyzed. Future work may demonstrate the creation of efficient FPGA programs from this basis.

Contents

Abstract **i**

1 Introduction **1**

1.1 Motivation 1

1.1.1 Parallelism: An attractive difficulty 1

1.1.2 In this work 3

1.2 Context 3

1.2.1 World as it stands: Parallel Programming Models . 3

1.2.2 Parallel Hardware 4

1.2.3 Optimization 6

2 Background Information **7**

2.1 Lift: Making parallel programs performance portable . . . 7

2.1.1 Structure 7

2.1.2 Rewrite Rules 10

2.2 Cyclo-Static Dataflow Graphs 11

2.3 Description 11

2.4 Fit for Model 13

3 Methods **14**

3.1 Design Principles 15

3.1.1 Modular Design 15

3.1.2 Dataflow 17
3.1.3 Optimization and Analysis 18
3.2 CSDF Construction 18
3.2.1 Parser 18
3.2.2 Type Inference 20
3.2.3 CSDF Generation 21
3.3 Optimizations 24

4 Results **25**
4.1 Sample 25
4.2 Coverage 26
4.2.1 Impossibilities 26
4.2.2 Future Improvements 27
4.3 Analysis 27
4.3.1 Inaccuracies 28
4.3.2 Default CSDF 28
4.4 Rewrite Rule Optimizations 28
4.4.1 Join/Split Cancellation Rules 29
4.4.2 Reorder Rules 30
4.4.3 Split-Join Rule: Parallel Map 30
4.5 CSDF-specific improvements 34
4.5.1 To Array Or Not To Array 34
4.5.2 Reduce 36

5 Conclusion **39**
5.1 Artifacts 40
5.1.1 Lift Project 40

5.1.2 CSDF Generator 40

Bibliography **41**

List of Figures

2.1 An overview of the Lift framework, from (Steuwer et al., 2015) 8
2.2 Lift's algorithmic rewrite rules, from Steuwer et al., 2015 . 11

3.1 The Join function CSDF representation. 16
3.2 Different representations of the dot program. 19
3.3 How a CSDF representation of the `Map` function is generated from the AST. 23

4.1 Percentage of programs from which CSDF representations were successfully generated. 26
4.2 Example functions using cancellation rules 29
4.3 The Lift HLL asum program. 30
4.4 Illustration of different CSDF representations of the scalar-vector addition program. 31
4.5 Time taken to generate CSDF for scalar-vector addition, parallel implementation of `Map` compared to the default. . . 31
4.6 MMNN CSDF execution time with `Map` implemented in parallel, relative to the naive execution. 32
4.7 Effect of core restriction on parallel `Map` 33
4.8 The Map Fusion Rule CSDF equivalence. 34

4.9 Speed improvement caused by de-arrayification applied to the parallel-map and naive versions of the dot function. . . 35
4.10 Different ways Reduce can be represented as a CSDF subgraph. 37
4.11 Speed improvement to dot function with different levels of parallelization. 37

Chapter 1

Introduction

This work outlines the first step towards producing a compiler for FPGA code from the Lift high-level language (HLL). It explains how parallelizable Cyclo-Static Dataflow Graphs (CSDFs) are generated, and details the optimizations and analysis performed on CSDFs to date.

1.1 Motivation

1.1.1 Parallelism: An attractive difficulty

When it comes to computing, faster is better. Luckily, computing power has continuously increased over the past decades in accordance with Moore's Law for single core performance. Unfortunately, Wirth's less well known law - that software gets slower more quickly than hardware gets faster - has also held true. In addition, there is significant speculation that the physical limits of Moore's law have been reached, at least until quantum computing becomes commercially viable (Khan, Hounshell, and Fuchs, 2018).

Parallelism has emerged as a major driver of high-performance computing. Multi-core devices have become ubiquitous, in everything from

servers to mobile phones. At the immediate level, this allows for multiple applications to be run at once. More interestingly, it means that one program can run multiple tasks in parallel. Parallel computing architectures are gaining significant traction. As well as being faster, parallelism generally allows for simpler design and better hardware and energy efficiency (Asanović et al., 2006, Chandrakasan, Sheng, and Brodersen, 1992). Admittedly, some use-cases are better suited that others. As Amdahl's law dictates, the efficiency of parallelizing a program is limited by the percentage that has to be executed sequentially (Amdahl, 1967). Data heavy applications, particularly mathematical and scientific analysis involving matrix manipulations, are particularly appropriate (Asanović et al., 2006). However, parallel computing is a promising avenue for improving performance across a range of applications.

This introduces new issues. Efficient, bug-free parallel programs are challenging to code (Asanovic et al., 2009). Many programmers struggle to adapt. Concurrent code is "notoriously difficult" to competently produce due to a combination of complexity, tricky resource management and a lack of overarching methodology (Darlington et al., 1993). High-level functional programming attempts to address this by introducing strong abstractions and implicit parallelism.

A further complication is introduced by hardware specialization. Different devices implement parallelism in vastly different ways, from multi-core CPUs and parallel GPUs to the focus of this paper, Field Programmable Gate Arrays (FPGAs). The mechanics of parallelism are fundamentally platform specific. The efficiency of different hardware also varies. High-level languages need platform-specific compilers to enable portability,

specifically tailored to maintain performance.

In summary, performant parallel code is an attractive way to increase computing performance. Unfortunately, it's difficult to write and not very performance portable across hardware platforms. Targeted functional languages help with the first issue, but require optimized compilers to resolve the second.

1.1.2 In this work

This work describes the first stage in writing an optimized compiler for the Lift parallel computing framework, targeting FPGAs. It explains how CSDFs are generated from Lift HLL code. It then details different optimizations that are experimented with, and the effect they have on projected performance.

1.2 Context

There are three key elements here: a user-facing high-level language (provided by Lift), a parallel hardware platform (FPGA) and an intermediate representation suited to optimization (CSDF).

1.2.1 World as it stands: Parallel Programming Models

Currently, there are three dominant parallel programming models: shared memory (OpenMP), message passing (MPI) and functional programming (MapReduce). Of these, the first two focus on managing information shared between systems running in parallel. Functional programming, in

contrast, prevents parallel threads from overwriting each other's memory by making all data immutable. The lack of side effects makes decomposition easy. Functional programs makes it easier to delegate parallelization to the compiler and runtime environment instead of the programmer, since the delineation between parallelizable parts and critical sections can be inferred from dataflow (Hammond, 2011). The functional paradigm carries great promise for parallel programming.

Of the various functional programming frameworks, this paper is focused on Lift. Lift is a new research framework intended to tackle the problem of performance portability across parallel architectures (Steuwer et al., 2015). Unlike many classic functional languages, it has been designed specifically to take advantage of parallelism. While there are several other new languages that try to do the same, Lift is distinct for a few reasons, discussed in depth in section 2.1, which make it ideal for highly data-parallel array operations.

1.2.2 Parallel Hardware

Hardware is often a trade-off between flexibility and efficiency, particularly when it comes to parallel programming. On one end of the spectrum is the CPU, that jack of all trades and master of none. At the other is Application Specific Integrated Circuits (ASICs), tailored to be very good at one application and no more. In between these two extremes you have GPUs and FPGAs. GPUs are closer to CPUs, with greatly improved parallelization abilities, but with priority also given to ease of programming. FPGAs are reconfigurable integrated circuits. They have similar efficiency improvements to ASICs, but are programmable. They

are specifically good for parallel programming, due to their origin as digital signal processors (Research, 2019). A performant parallel programming language like Lift will benefit from being extended to FPGAs.

Unfortunately, FPGAs are notoriously difficult to work with (Darlington et al., 1993). Programming is challenging, more akin to circuit design than modern code (Fine Licht, Blott, and Hoefler, 2018). Code compilation (more exactly, feature synthesis and implementation) takes a significant amount of time - often longer than it took to write the code. Speedup is generally only achieved at the cost of quality (Mulpuri and Hauck, 2001). FPGAs offer many desirable features, but are not approachable for the novice programmer. This makes an optimized compiler all the more desirable.

Several projects already tackle the painful coding process through High Level Synthesis (HLS). This is the process of taking a higher abstraction of the program and converting it to Verilog, a low level hardware description language (HDL) for programming FPGAs. Of the existing 80+ HLS systems, most use C or C++ as the source language, with others using MATLAB, Java and Python (Nane et al., 2015). However, these can still be challenging to engage with. Few properly abstract away all hardware concerns. Chisel, the only Scala-based project, feels like a re-implementation of the HDL in Scala (Bachrach et al., 2012). While OpenCL can be compatible with FPGAs, it generally has very poor performance. A Lift to Verilog compiler would allow programmers to code without any consideration at all for the hardware, meaning that a program initially written for a GPU could be easily run on a FPGA with no adjustments. This is an extreme level of portability. The difficulty is in

ensuring that performance is equally portable.

This is the problem that this work addresses, by examining how the CSDF intermediate representation can be used for optimization.

1.2.3 Optimization

This brings us to the issue of optimization. Lift's Rewrite Rules (subsection 2.1.2) provide one avenue to exploit. There is also a lot of opportunity for FPGA-specific improvement. The existing HLS tools mentioned above give a lot of good examples. Most include improvements like operations chaining and augmented scheduling, as well as bitwidth, memory, loop and spatial parallelism optimisation. To both implement and analyze this, most use some form of dataflow-based intermediate representation (Stewart et al., 2017). In particular, Cyclo-Static DataFlow Graphs (CSDFs) are very popular (Vlugt et al., 2019, "Advanced Model-Based FPGA Accelerator Design" 2017, Aubry et al., 2013).

CSDFs are good intermediate representations for FPGAs, since both exemplify the pipe-based dataflow model. Further, CSDFs have several algorithms available for scheduling improvement, which is a key optimisation for FPGA code (Bodin, Munier-Kordon, and Dinechin, 2013). They also allow you to model and adjust programs before loading them into the FPGA, which helps with iterative developments.CSDFs are a good fit for Lift, preserving its functional nature. Overall, CSDFs have proven to be an excellent vehicle for optimizing FPGA code.

Chapter 2

Background Information

2.1 Lift: Making parallel programs performance portable

Lift is still in development, changing over time. The key features have already been discussed: it's functional, array-specific and parallel. To build a compiler, deeper knowledge is necessary.

2.1.1 Structure

There are three major elements to Lift, as seen in Figure 2.1. The first is a functional, high-level language. This is pretty straightforward to use for anyone familiar with other functional languages like Haskell or Lisp. It is restricted to algebraic datatypes and provides six core algorithmic functions, although it also allows for user-defined methods. The second is a collection of rewrite rules, which gives a selection of ways to manipulate the algorithmic primitives while maintaining semantic equality. This gives a range of possibilities for optimization. The third is a generator for OpenCL code, targeted at a specific GPU or CPU hardware.

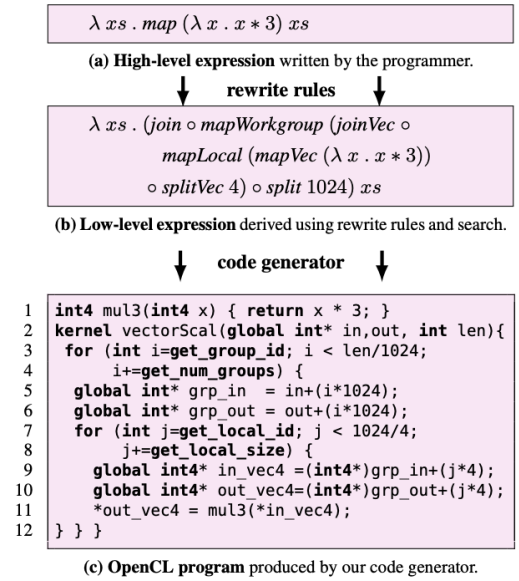


FIGURE 2.1: An overview of the Lift framework, from (Steuwer et al., 2015)

Algorithmic Primitives

Lift's high-level language is based on an extension of the `Map-Reduce` functional pattern to include further array manipulations, such as `Zip`, `Split` and `Join`. Different articles from the Lift project introduce other primitives as well, with varying behaviour (Steuwer et al., 2015, Steuwer, Remmelg, and Dubach, 2017). While we initially developed the theory for our optimizations based on this research, this ran into some conflict with the reality of the project codebase. As is understandable for an early-stage project, the Lift codebase,documentation, and academic representation (Steuwer et al., 2015, Steuwer, Remmelg, and Dubach, 2017) diverge in significant points. For the sake of consistency, the work in this paper was built with the Lift codebase at commit hash 5e8a18df as the source of truth, represented in Equation 2.1.

$$
\begin{aligned}
&zip_{A,B,I} \ : \ [A]_I \to [B]_I \to [A \times B]_I \\
&get_{A,I} \ : \ [A \times B] \to I \to A \\
&split_{A,I} \ : \ (n : size) \to [A]_{n \times I} \to [[A]_n]_I \\
&join_{A,I,J} \ : \ [[A]_I]_J \to [A]_{I \times J} \\
&map_{A,B,I} \ : \ (A \to B) \to [A]_I \to [B]_I \\
&reduce_{A,I} \ : \ ((A \times A) \to A) \to A \to [A]_I \to [A]_1
\end{aligned}
\tag{2.1}
$$

Lift also allows for extra user-defined functions to be written in OpenCL. This works for the initial structure, since the user functions can be easily incorporated into the OpenCL code generated by Lift. It works less well for this project, since OpenCL is not a Domain-Specific Language (DSL) for FPGAs. We settled for incorporating the `multiply` and `add` operations into our generator at the same level as native Lift primitives.

Types

Only algebraic types are available in Lift, although the base `int` and `float` can be extended into tuples and arrays (of various depths). The two base types are `float` and `int`. Then there are two collection types: you can have an `array` of I elements of the same type, of a `tuple` of two elements of any type - the same, or different. These can be built on each other, such that you can have an `array` of `tuples`, where the first element of each `tuple` is a multi-dimensional `array` of `int` and the second is a `float`. While this is comparatively limited, it suffices for most programs suited to parallelism such as graphics rendering, scientific computing and mathematical analysis. The simple structure also makes building

a CSDF generator easier, and optimizations are easier to prove without possible complications from edge-cases.

2.1.2 Rewrite Rules

Since Lift is formed from a few clearly defined mathematical formulae, there are several options for transforming one expression into a different but semantically-equivalent one, which may be more efficient for parallelism. For instance, sequential `Map` calls can be composed into one. The Lift development community has developed several rewrite rules already, reproduced in Figure 2.2. These are of two classes: algorithmic and OpenCL-specific. Algorithmic rules transform an expression formed from high-level primitives into a semantically equivalent expression, also of high-level primitives. The OpenCL specific rules transform high-level expressions into ones that contain functions like `splitVec`, which are explicitly targeted at OpenCL. FPGAs don't use OpenCL, so this second class is not relevant. However, the algorithmic rewrite rules have potential.

These rules are useful because different semantically equivalent expressions may not be equally parallelizable. In the work so far, they have come in most useful for optimizing storage and memory access (Steuwer et al., 2015). Not all continue to be relevant; rewrite rules in the literature sometimes make use of primitives that didn't stick around. However, since the hardware organisation of FPGAs and GPUs is significantly different, the optimal rewrite rules are also likely to be different. Part of this Capstone is focused on empirically showing which are best suited for the FPGA platform.

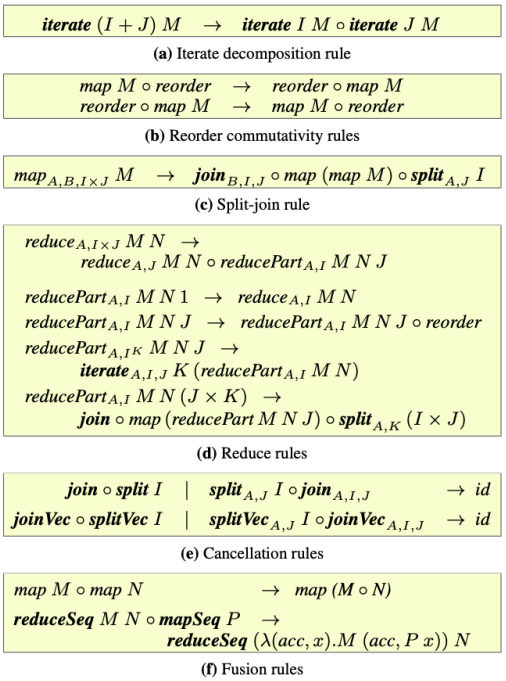


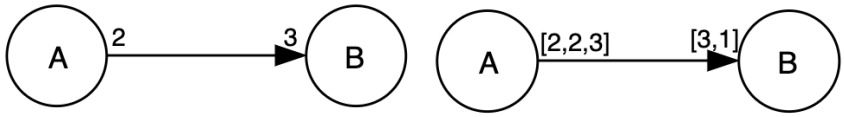
FIGURE 2.2: Lift's algorithmic rewrite rules, from Steuwer et al., 2015

2.2 Cyclo-Static Dataflow Graphs

Interestingly, CSDF graphs were also originally designed for signal processing, which goes some way to explaining why they're such a good fit for FPGA programming. The way they handle dataflow and scheduling is very similar.

2.3 Description

CSDFs follow the flow of data through a series of operations. Each node represents a side-effect free function. Each edge represents a data buffer. This is a directed graph; the source node of an edge feeds data into the buffer, while the destination removes and processes it. At each end of the buffer is a port; the input port connecting it to the producer node



(A) A very simple representation of an CSDF graph

(B) A very simple representation of an CSDF graph, with phases

and the output port connecting it to the consumer node. Each node produces/consumes a certain amount of data each time it's executed, with this number annotated at the appropriate end of the connecting edge, representing the port. For instance, in Figure 2.3a node A produces 2 datum with every execution, while B consumes 3. While data follows the flow of the graph, time does not. Any node can execute at any time, so long as there is enough data in its input buffers. Scheduling the execution order of nodes is very important. For instance, in Figure 2.3a A must be executed at least twice before B can be, since after the first execution there will only be 2 datum in the buffer, and B requires 3. In larger systems, this is more complicated (Bodin, Munier-Kordon, and Dinechin, 2013).

While this covers the general idea of a dataflow graph, there are three other important additions. First is a delay. A delay refers to data preloaded into the buffer when the system is started. In Figure 2.3a a delay of 1 on the edge would mean that B could run straight after A's first executions, since there would be 1 + 2 = 3 datum in the buffer. A delay makes possible another feature: self-reflexive edges. These allow nodes to feed data back to themselves. Finally, a key element to CSDFs is that nodes can cycle through phases. In other words, they can produce/consume a different number of data depending on the phase. In the modified CSDF graph in Figure 2.3b, node A has three phases while node B has two. The first time A fires, it produces 2 datum, and again on the second. The third time, it

produces 3 datum, before cycling back to the start of the phase vector to produce 2 datum in the next execution. Meanwhile, B alternates between consuming 3 and 1 datum. An excellent explanation of this can be found in Bodin, Munier-Kordon, and Dinechin, 2013.

2.4 Fit for Model

CSDF graphs are an excellent fit for our project. Since both CSDF and FPGAs were designed for signal processing, they are a strong match for modelling FPGA code, because the assumptions and general architectural model are the same. For instance, key elements of both are variable schedule and clock time. They are functional like Lift, and different rewrite rules can be modelled as differently arranged CSDF graphs. This allows for testing using a CSDF-specific analysis tool like Kiter (Bodin, Munier-Kordon, and Dinechin, 2016). These tools also include algorithms for finding the best schedule, and as previously mentioned, scheduling is an immediate avenue for optimization of FPGA code. CSDF graphs are an excellent intermediate representation, retaining a strong structure for easy compilation while allowing for deeper analysis and scheduling.

Chapter 3

Methods

CSDF graph generation is a multi-step process. A high-level Lift program is first parsed to create an abstract syntax tree (AST) representation. Data-type and length information is then cascaded down from the initial input values to populate every `Node` in the AST. A CSDF representation can then be created. This work provides several different methods of producing a CSDF representation, and an analysis of which methods of optimization succeeded best.

CSDFs are generated directly from high-level Lift code. This work includes a lexer/parser suite that produces a tailored AST, which is used to generate and optimize a CSDF representation. Finally, datatype and buffer information are derived, for later use in the FPGA code. Initial plans were to generate the CSDF directly from the internal Lift AST. Unfortunately, this did not retain all the data necessary for FPGA programming, meaning that a tailored parser was necessary.

3.1 Design Principles

3.1.1 Modular Design

When converting Lift HLL code into a CSDF graph, each Lift primitive is treated as a discrete unit. A CSDF graph comprises of a list of `Nodes` and `Channels`. A `Node` contains input and outpur `Ports`, connected other `Nodes` by a `Channel`. Each primitive is compiled into a CSDF `Node` or sub-graph, with incoming and outgoing `Ports`. Figure 3.1 gives an insight into how this works for the `Join` `Node`. This strategy prioritizes modularity. To build the graph, only the input and output `Ports` associated with each `Node` must be known. These components are then linked together by appropriate `Channels` to form a complete CSDF graph that accurately represents the original program. The design of each primitive can be easily updated, so long as the input and output `Ports` remain the same. This makes CSDF-based optimizations easier, since different representations of each primitive can be tested without disturbing the graph as a whole.

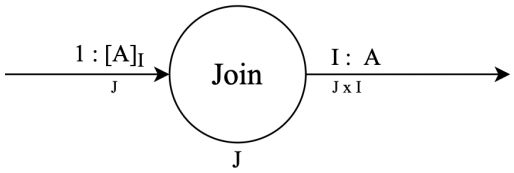
This modularity also encompasses higher-order functions, like `Map` and `Reduce`. Operations like `Zip` perform the same operation every time, parameterized only by integer length variables. In contrast, higher order functions depend on an input function. That might be a user-defined function or it might be an expression containing several Lift primitives. In the creation of a CSDF from Lift, that means these functions actually translate to CSDF sub-graphs. However, the interface exposed to the rest of the CSDF (the input and output `Ports` of the `Map` or `Reduce` subgraph) is kept constant. This means the design of these higher-order functions can be experimented with in isolation. The focus on modular design allows

```
class Channel:
    def __init__ (self, src_act, dst_act,  src, dst, init_token = 0):
        self.name = src+"_to_"+dst
        self.src_port = src
        self.src_act = src_act
        self.dst_port = dst
        self.dst_act = dst_act
        self.init_token = init_token
        self.datatype = False # set at the type inference stage

class Port:
    def __init__ (self, direction, name, rate):
        self.name = name
        self.direction = direction
        self.rate = rate

class Join(CSDFNode):
    def __init__(self, name, count):
        self.name = name
        self.input = Port('in', name + '_in', [count])
        self.output = Port('out', name + '_out', [1])
```

A. Class definition of `Channels`, `Ports`, and the Join `Node`.



B. A graphical representation of the Join function as a CSDF `Node`.

FIGURE 3.1: The Join function CSDF representation.

for different optimization to be easily implemented at the CSDF design level.

3.1.2 Dataflow

Dataflow is vital to dataflow diagrams, so inferring and tracking data type and size throughout a Lift program is vital to building the subsequent CSDF. This is prioritized at all stages of the CSDF generation process. Lift is a strongly typed language and all primitives behave in a predictable manner, which makes things easier. Type inference is performed at an early stage and information is retained throughout the process.

Data information is important for several reasons. Some of the information is necessary to construct the CSDF topology. For instance, array length dictates how many instances of a `Reduce` subgraph are necessary. Type information is also vital for informing both buffer size and phase count. This is important for CSDF design and accurate analysis. Accurate CSDF analysis is vital for optimization decisions. Buffer size information in particular will be necessary for the end goal of producing efficient FPGA programs. Type information is also helpful for compile time checking.

This requires an addition to the notation. Traditional CSDF notation only denotes the length of the input data at each buffer, not the type of the data. In this work, instead of just being labelled with the phase vector `P`, the input and output `Ports` are labelled `P : T`, where `T` is the type information.

3.1.3 Optimization and Analysis

The CSDF intermediate representation presented in this work has two primary purposes; program optimization and analysis. The compilation procedure is tailored to optimize for this, by allowing for different optimizations to occur at multiple levels. They may occur at the initial stage to the parsed Lift code, during CSDF generation, or after. Parallelism is a particular focus. This means that Lift primitives are given CSDF representations that should allow for as much parallelism as possible.

The Kiter analysis tool requires the CSDF to be rendered into a gicen XML format, with buffer and timing information. The code representation and class definitions (a sample is given in Figure 3.1) were tailored to make this simple.

3.2 CSDF Construction

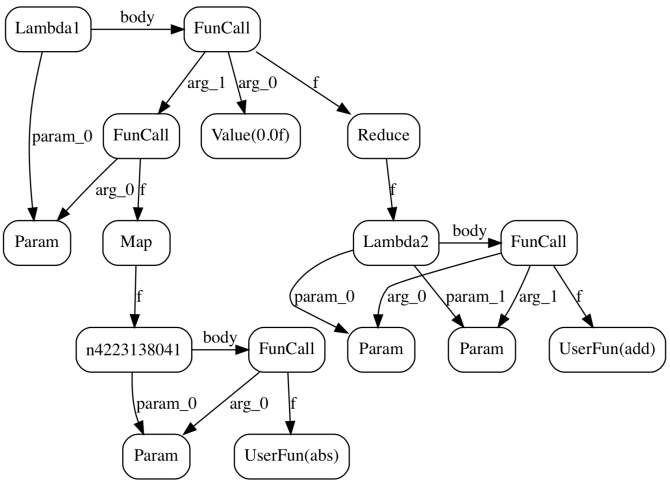
3.2.1 Parser

Any compiler requires a parser, to turn a text file into a workable Abstract Syntax Tree (AST). The Lift project already includes a parser that produces a parse tree, but we found it an imperfect match for this project. The parse tree introduced a lot more complexity than a normal AST, including a lot of additional syntactic structure, while lacking a lot of information that was necessary for generating accurate CSDFs. Thus the first component of this work is a purpose-built lexer and parser[1] to produce a

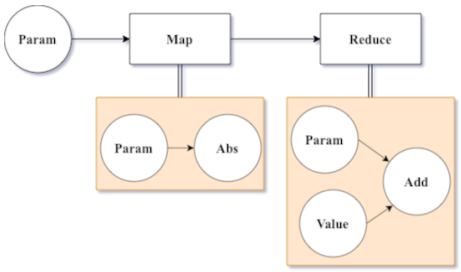
[1] In retrospect, it would have been a wiser decision to make use of an automatic Lexer and Parser Generator, like the ones found in the catalog of compiler construction tools. Making one from scratch was a novice mistake, which resulted in a comparatively brittle lexer.

```
A =>
    reduce(add, 0) ∘ map(abs) $ A
```

A. Dot program included in the Lift project.



B. Lift parse tree representation.



C. Our AST representation.

FIGURE 3.2: Different representations of the dot program.

bespoke AST. Figure 3.2 shows the difference between our AST and the Lift parse tree.

This AST is an intermediate representation to transform the Lift program from a string into a form that can easily be manipulated into a CSDF graph. While the parser was additional work, the resultant AST is a much cleaner stepping stone, intended to be easily interoperable with a CSDF representation. Where possible, the AST `Nodes` use the same classes as the CSDF graph `Nodes`, and the graph uses a similar structure, with a layer

of abstraction where there is expected to be many CSDF representations. Higher order functions, like `Map` and `Reduce`, are represented simply in the AST as a single `Node`. These `Nodes` then hold the AST of their input functions as a field, as illustrated in Figure 3.2 C. This maintains the principle of modular design discussed earlier.

Other than that, the AST structure naturally echoes a simplified CSDF structure. This is largely due to the reasons discussed in chapter 2 about how CSDFs are a good model for the Lift high level language. However, the parser was also tailored to amplify these similarities. The end result is that the AST can usually be taken to be a very, very simplified CSDF.

3.2.2 Type Inference

Type inference is performed immediately on the AST. This had initially been left until later in the process, but type information proved important enough to be made available as early as possible. The AST stage is also where a Lift program is most predictable, making it easier to perform type inference.

Lift's deterministic nature makes this step significantly earlier, since each `Node` transforms the input data in a predictable fashion. Lift programs also provide the input data type in the definition of the main function. These can then be cascaded down the linked `Nodes`, with the cascade only continuing once all input data types haves been inferred.

The one complication is user-defined OpenCL functions. The output type of these is not always known, which is problematic. Future

work could look into performing type inference on them, where possible. However, even without it, some type information can be inferred directly from the available Lift information. For instance, if a user-defined function feeds into a `Join`, then the user-defined functions input type can be inferred to be an Array of Arrays of unknown elements (but the same length). If the `Join` funnels into a `Reduce` with 'add' as the reduction function, the unknown element must be either an Int or a Float. Float is generally assumed since it's a working supertype. This allows for some type information to be inferred in otherwise challenging places. Unfortunately, this is still incomplete. If the user-defined function output is not operated on again, no further information can be gained. Further, while data *type* information may be inferred, data *length* cannot be.

At this stage, data type information is stored in the `Nodes`. This means that their arrangement can be more easily manipulated. However, this information is actually particularly relevant for the `Channels` between the `Nodes`. This is because the data type (and length) information is necessary for calculating the required buffer size for each `Channel`. Buffer information is important for analyzing CSDFs and absolutely vital for compiling FPGA code.

3.2.3 CSDF Generation

Once the AST has been formed, CSDF graphs can be generated from it. This mainly involves replacing each `Node` in the AST with an appropriate CSDF representation. This is particularly complicated for higher order functions. Since the initial AST was largely structured as a CSDF, replacing each `Node` in a modular fashion is most of the work, but not all.

Higher order functions

As mentioned earlier, each higher order function is represented as a single `Node` in the AST, with the input function as an attribute of that `Node`. In order to generate the CSDF, the first step is to generate a CSDF subgraph from that input function. This is then subbed into a more detailed CSDF representation of the higher order function to create a concrete subgraph. Because the components are modular, the initial `Node` can be directly swapped out for this subgraph. For greater clarity, an example is given in Figure 3.3.

Parameters

One of the complications with transforming an AST into a CSDF graph is ensuring the `Param Nodes` representing the inputs are linked correctly to every use. The AST is multi-level, with `Params` from higher levels being used at ones below. For instance, one parameter may be used as input for a `Zip` function on the top level, but also within the input function of a `Map`. These are separate things in the AST, especially as the input function AST is distinct from the top-level one. However, in the CSDF, both of these occurrences must be fed from the same `Param Node`. This is accomplished with good labelling in our AST.

Final touches

Some extra information needs to be derived for the CSDF to be complete. Notably, the `Channels` between `Nodes` require work. Data type and length information is gathered from the input `Node`. Information about data rates and phases is usually derived when the `Nodes` are being swapped

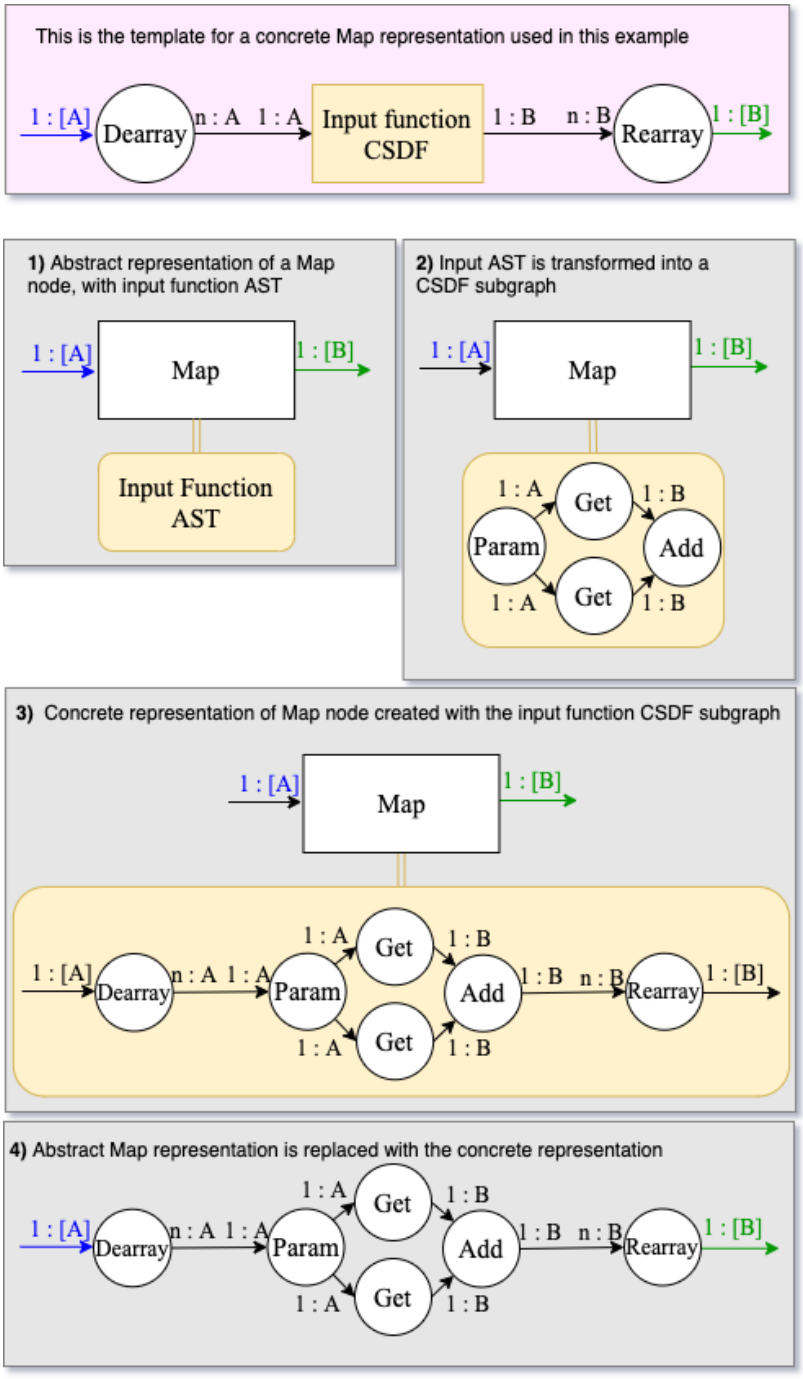


FIGURE 3.3: How a CSDF representation of the `Map` function is generated from the AST.

out. Each `Node` object comes with certain vital knowledge, like execution time.

Finally, this can be used to generate an XML description of the CSDF graph, which can be easily analyzed.

3.3 Optimizations

As previously discussed, we are generating CSDFs because they are a good vehicle for optimization of Lift code. By this we mean that it is possible to create several different CSDFs all equivalent to the same Lift program. Some of the CSDF representations will perform better than others.

There are two primary forms of optimizations. The first is rewrite rules from the Lift project. The second is CSDF-specific. These will be further elaborated on in chapter 4.

To test optimizations, multiple CSDFs may be generated using different methods. This is also helpful in practice, to test whether different CSDFs will work best given certain hardware constraints. This is another place where the focus on modularity comes in useful. Some optimizations can be expressed as different designs of the CSDF representation of a Lift primitive. As described earlier, these can be easily swapped in and out of a CSDF graph by maintaining the same input/output `Port` interface.

Chapter 4

Results

The first goal of this work was to successfully automate the generation of CSDF graphs from Lift high level code. The secondary goal was to consider and implement potential optimizations, and analyze which were most effective. The results are below.

4.1 Sample

The Lift framework provides a selection of 25 prototypical high level programs for testing. These are primarily fairly simple programs, covering archetypal algorithms such as matrix multiplication and k-means clustering.

This paper contributes four more programs to this suite. These include one for calculating a dot product and another for adding scalars to an array.

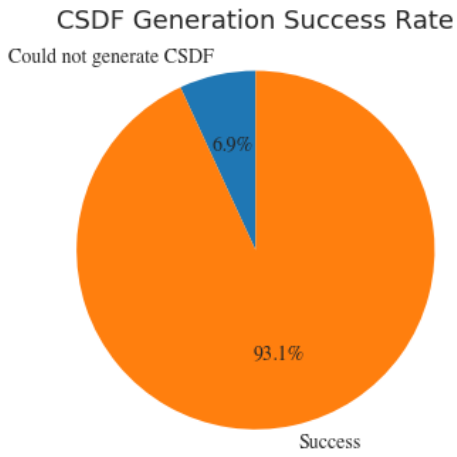


FIGURE 4.1: Percentage of programs from which CSDF representations were successfully generated.

4.2 Coverage

The CSDF generator has a 93% success rate on the sample programs. This unreliability is caused by the user-defined OpenCL functions. Of the programs which did not include user-defined OpenCL functions, we had a 100% success rate.

4.2.1 Impossibilities

As mentioned in subsection 3.2.2, the output of user-defined functions is not as well-defined as for Lift primitives. In particular, even if the type of the output can be inferred to be an array, the array length remains unknown. This is problematic because knowledge of array length is often necessary for generating CSDF representations other functions, in particular `Reduce` and `Join`.

This means that a CSDF cannot be generated for a program where the input data for a `Reduce` operation has passed through a user-defined

OpenCL function. An accurate `Reduce` CSDF subgraph cannot be produced without the missing data length information. Two of the sample programs featured this structure, the k-means clustering and molecular dynamics algorithms.

A CSDF generator for OpenCL would be necessary to overcome this issue, but is outside the scope of this work.

4.2.2 Future Improvements

This makes an OpenCL CSDF generator a clear next step. Unfortunately, it would be far more difficult to produce a similar CSDF generator for OpenCL due to the vast difference in complexity. A possible workaround would be to request that the programmer provide the necessary information when they define the function. However, this would break the interoperability between FPGAs and other hardware platforms for the Lift HLL. It is worth considering.

4.3 Analysis

As previously mentioned, we use Kiter to analyze the CSDF graphs (Bodin, Munier-Kordon, and Dinechin, 2016). This tool formulates the optimal schedule for executing each CSDF graph. It then returns the minimum period, or the lower bound for program execution using a given CSDF formulation.

4.3.1 Inaccuracies

In order to calculate this, Kiter requires the execution time for each node. For each Lift primitive this can be derived once and expected to remain constant, although theoretical values are used in this study. However, user-defined functions once more raise an issue, since they are typically unique with unknown execution times.

The current way to handle this is to request the execution time of all user-defined functions when generating the CSDF. For the most accurate analytics and optimizations, the user-defined function should be tested on an FPGA. However, testing revealed that the execution time of user-defined functions had very little impact on the optimization provided by different transformations.

4.3.2 Default CSDF

The first method for generating CSDF representations of Lift graphs is the simplest. It comprises the default (non-parallel) `Map` shown in Figure 3.3 and Figure 4.4, as well as the recursive `Reduce` shown in Figure 4.10. We refer to this as the default or naive method, and it forms the basis for comparison to evaluate speed-ups of other optimizations.

4.4 Rewrite Rule Optimizations

Not all of the classical Lift rewrite rules were applicable in this case. Around 75% were either specific to OpenCL threading, or made use of low-level OpenCL primitives (Steuwer et al., 2015). However, there were a handful of useful algorithmic rewrite rules.

```
A, I =>
    join() ∘ split(I)
    ∘ map(x => mult(x,
        10))
    $ A
```

A) Single occurrence

```
MN, I =>
    map(x =>
        join() ∘ split(I)
            $ A)
    $ MN
```

B) Multiple occurrence

FIGURE 4.2: Example functions using cancellation rules

4.4.1 Join/Split Cancellation Rules

$$join \circ split\ I \quad \rightarrow \quad id \tag{4.1}$$

$$split_{A,J}\ I \circ join_{A,I,J} \quad \rightarrow \quad id \tag{4.2}$$

These cancellation rules state that the operation of splitting an array into I pieces and then joining them together again is equivalent to doing nothing at all, as is the converse. This makes intuitive sense.

These rewrite rules were a successful, if minor, optimization. This is not a common structure, so made no impact in most cases. Where it did, there was a small but reliable improvement. The cancellation rule removes two CSDF nodes, `Join` and `Split`, which in this model are constant-time functions, resulting in a constant-time improvement for every occurence of this pattern in the CSDF. This is a minor improvement if the pattern only occurs on the top level as in program A from Figure 4.2, but can have a much greater impact if it occurs within a `Map` or `Reduce` subfunction, as in program B.

Overall, this was a positive optimization with no drawbacks, and is used in the default (or naive) method of generation.

```
(vector, alpha) =>
    Map(fun(x => mult(x, alpha)))
    $ vector
```

FIGURE 4.3: The Lift HLL asum program.

4.4.2 Reorder Rules

$$reorder \circ \texttt{Map}M \rightarrow \texttt{Map}M \circ reorder \tag{4.3}$$

$$\texttt{Map}M \circ reorder \rightarrow reorder \circ \texttt{Map}M \tag{4.4}$$

The reorder rules clarify that, if order of an array doesn't matter at one side of a `Map`, it doesn't matter at the other side either. These rules were somewhat unhelpful in this context, as it became clear that reordering data, randomly or otherwise, would require interrupting or changing dataflow channels in the CSDF in a way that would cost more than it saved. Instead, the `Reorder` primitive was simply dropped in the default mode.

4.4.3 Split-Join Rule: Parallel Map

$$map_{A,B,I\times J}M \rightarrow join_{B,I,J} \circ map_{A,B,I\times J}M \circ split_{A,I\times J}I \tag{4.5}$$

The split-join rule essentially states that a `Map` can be done in parallel. It can be broken into multiple segments using `Split`, on which the mapping function is performed in parallel, which are then combined back into the original array with `Join`. Figure 4.4 shows how this CSDF representation differs from the default.

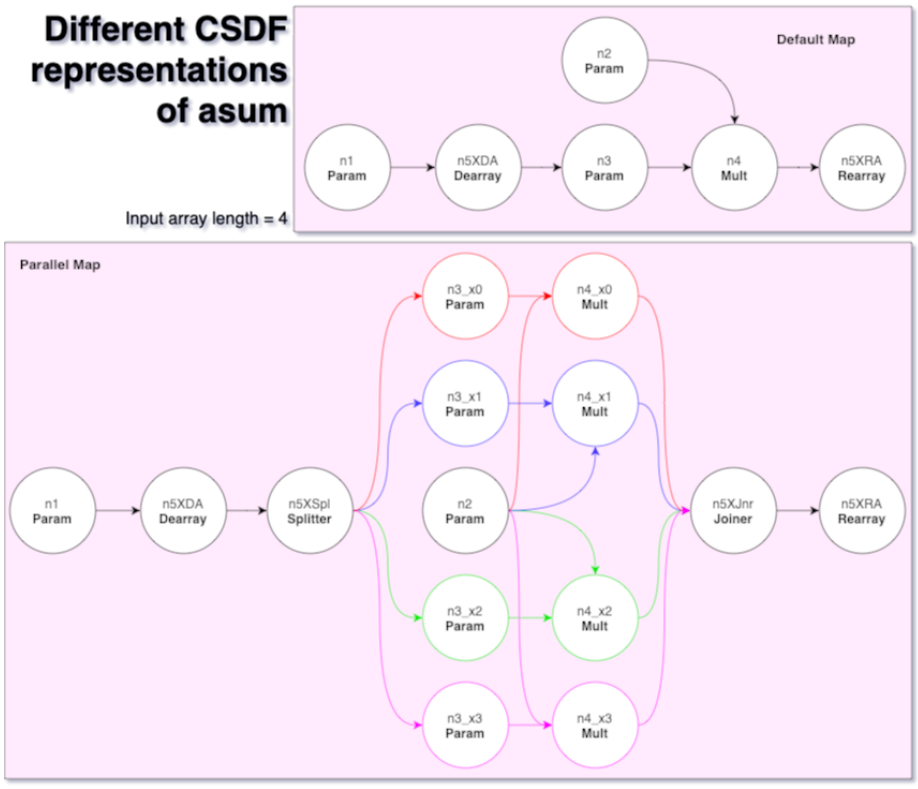


FIGURE 4.4: Illustration of different CSDF representations of the scalar-vector addition program.

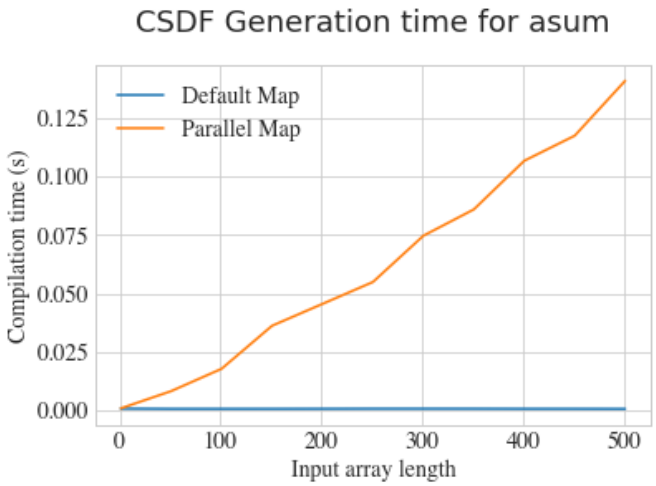


FIGURE 4.5: Time taken to generate CSDF for scalar-vector addition, parallel implementation of `Map` compared to the default.

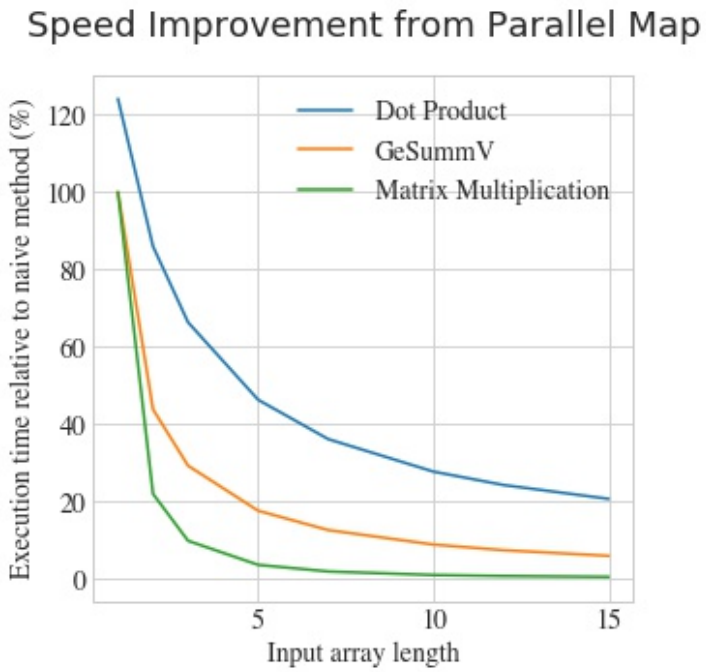
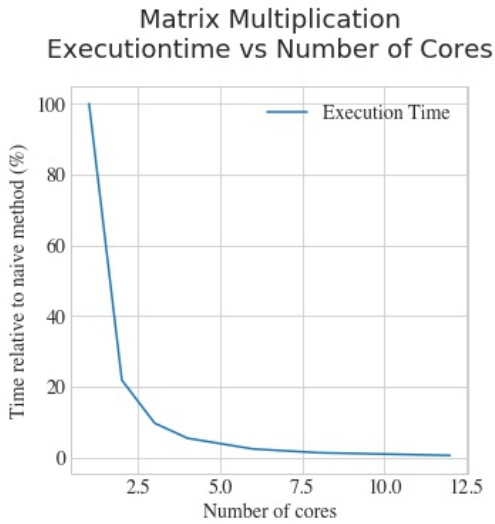


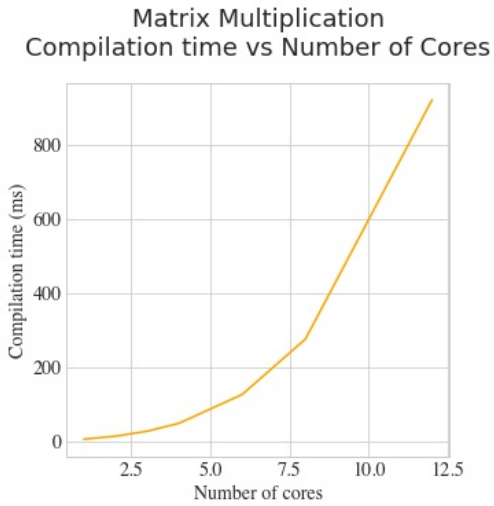
FIGURE 4.6: MMNN CSDF execution time with `Map` implemented in parallel, relative to the naive execution.

Compilation times are a lot longer for this parallel `Map`. This is because the mapping subgraph needs to be duplicated for each segment running in parallel, as shown in Figure 4.4. This needs to be a deep copy, which is an extremely expensive operation given that a subgraph is an extremely complicated, multilayered and heavy object. On top of that, almost everything needs to be renamed to prevent id collisions. Python is a particularly poor choice for speeding this up.

Using the default implementation of `Map`, the CSDF generation time remains constant. With the parallel method, it increases linearly with the length of the input array. Figure 4.5 illustrates how this effect quickly becomes quite noticeable for `asum`, the scalar-vector addition function provided in the Lift codebase. This takes even longer for more complicated programs, like matrix multiplication.



A) Speed improvement for Matrix Multiplication (15x15 matrix) dependent on the number of cores available



B) CSDF generation time for Matrix Multiplication (15x15 matrix) dependent on the number of cores available

FIGURE 4.7: Effect of core restriction on parallel `Map`

However, the long compilation times pays off with major speed improvements. Figure 4.6 shows the execution time of a CSDF with `Map` implemented in parallel as a percentage of the execution time for the naive CSDF, plotted against the array length of the input. The given applications are all heavily data-parallel, so the input array data is a serviceable proxy for the parallelizable portion of the program according to Amdahl's Law (Amdahl, 1967). Implementing a parallel `Map` can very easily result in speed-ups of more than 20x.

These stats are, admittedly, assuming an infinite number of nodes can be run at the same time. The actual number of nodes that can execute simultaneously will be dependent both on the size of the FPGA and the type of nodes, since each node type takes up a different number of logic blocks. Using the simplifying assumption that one process can run in

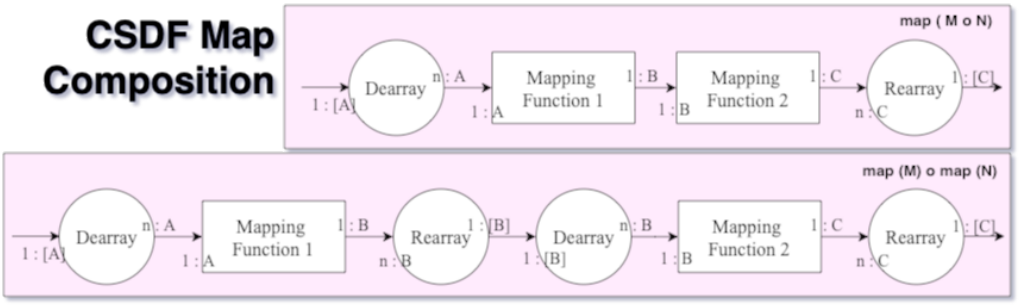


FIGURE 4.8: The Map Fusion Rule CSDF equivalence.

one core at a time, Figure 4.7 A shows the impact of restricting the number of cores on the speed improvement from parallel mapping for matrix multiplication of a 15 by 15 matrix. 10x speed improvement is seen with just four cores. This does improve as more cores are added, but at a markedly lesser rate. Figure 4.7 B shows that fewer cores also result in a quicker CSDF generation time. Generating a CSDF using parallel mapping across four cores takes 48 milliseconds. For eight cores, it takes 276 milliseconds.

4.5 CSDF-specific improvements

4.5.1 To Array Or Not To Array

Arrays and their elements are an essential aspect of Lift, as is the transition between them. Functional operations like `Map` take in an array as input, but then operate on each array element individually. In the CSDF representation, this is handled by `Dearray` and `Rearray` nodes. `Dearray` takes in an array as input, and outputs each of its elements individually. `Rearray` does the opposite. Thus the `Map` representation pipes the input

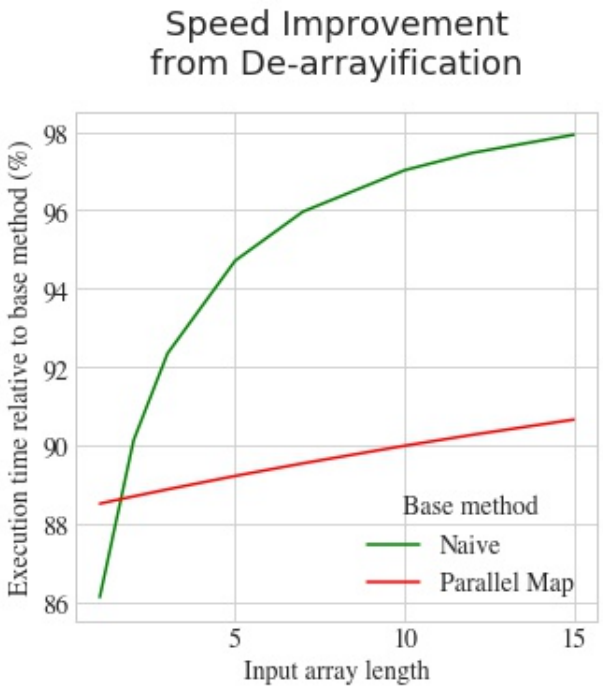


FIGURE 4.9: Speed improvement caused by de-arrayification applied to the parallel-map and naive versions of the dot function.

array through a `Dearray` node, pipes each element through the subfunction CSDF, then finally `Rearrays` them. `Reduce` also has this structure, with the reduction function being bookended by `Dearray` and `Rearray` nodes.

It is also possible to make the design choice of having `Zip` follow the same pattern. In the default implementation, `Zip` simply takes in two arrays, and outputs and array. Alternatively, the inputs can be dearray-ified so that each element pair is `Zipped` together independently before being rearray-ified at the end. In isolation the default `Zip` is the better choice, since `Dearray` and `Rearray` would just be unnecessary overhead.

$$\texttt{Map}\ M \circ \texttt{Map}\ N \quad \rightarrow map(N \circ M) \tag{4.6}$$

However, sometimes `Dearray` and `Rearray` nodes may cancel each other out. Lift already has a version of this called the Map Fusion Rule (Equation 4.6), as seen in the CSDF equivalence from Figure 4.8. This concept is extensible. Rearraying an array only to immediately dearray it is unnecessary.

$$rearray \circ dearray\ A \quad \rightarrow \quad id \tag{4.7}$$

$$dearray \circ rearray\ A \quad \rightarrow \quad id \tag{4.8}$$

As Figure 4.9 shows, this optimization can give up to a 10% improvement. However, the `Dearray` and `Rearray` nodes are typically outside of `Map` or `Reduce` subfunctions and thus not data-parallel, meaning that the proportional benefit quickly decreases with data size. The absolute decrease in execution time, however, remains constant.

4.5.2 Reduce

So far, the CSDFs have been formed by replacing the abstract representation in the AST with a more detailed recursive design. This work also implements a parallel version. Figure 4.10 shows these different designs.

This parallel `Reduce` is a very successful optimization, with a 5.2x speed-up on four cores for the dot product. This version took 3x longer to compile than the fully recursive version.

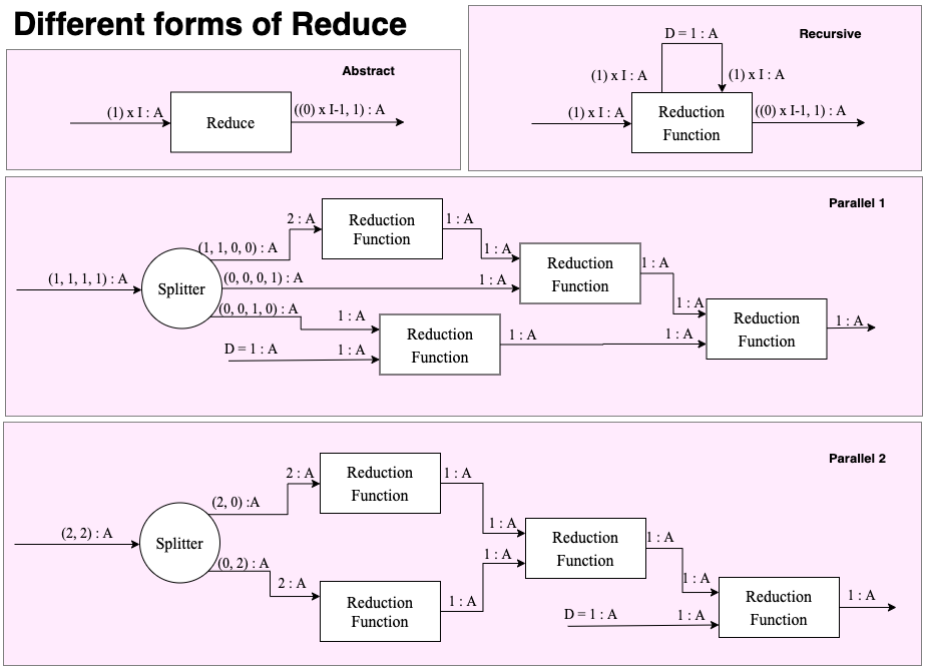


FIGURE 4.10: Different ways Reduce can be represented as a CSDF subgraph.

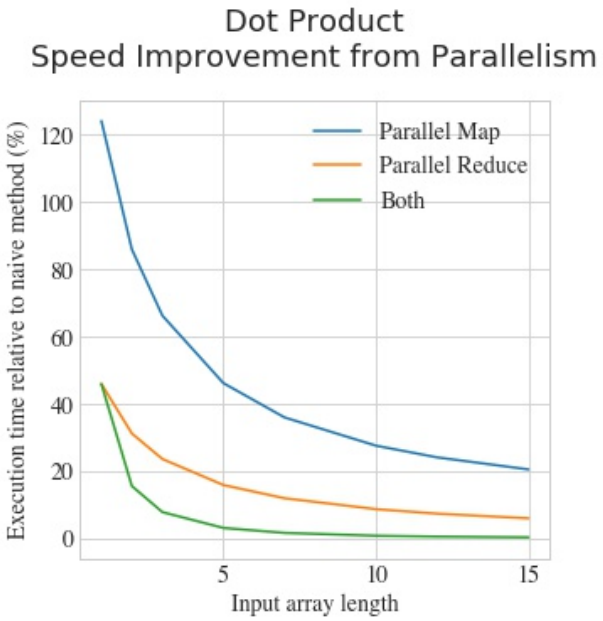


FIGURE 4.11: Speed improvement to dot function with different levels of parallelization.

For the dot function, using a parallel implementation of `Reduce` provides twice the speed-up as a parallel `Map`, although this does depend on the exact formulation of the program. Implementing both primitives in a parallel manner can theoretically give over 100x speed-up with infinite computing resources. Limiting it to four cores, we found the dot product was limited to a 5.5x speed-up.

Chapter 5

Conclusion

This work set out to build the first part of a compiler for FPGAs from Lift, and in doing so demonstrate that CSDFs are a good intermediate representation for optimization. We have successfully produced a working CSDF generator, and engineered several conclusive improvements to the naive design.

It is unsurprising that parallelization produces the strongest optimizations, although the magnitude of the improvement can be greater than we expected. This is partly due to the way the Lift HLL is structured to encourage highly data-parallel code. Speed improvement of 10x was possible even with only 4 cores. However, these are not the only optimizations we have shown. The Lift cancellation rule also causes minor improvements, as does this paper's proposal of a more de-arrayified `Zip` combined with more streamlined array mechanics.

Overall, our initial assumption that CSDFs are a good intermediate representation for Lift code compilation has held up, and we have successfully used them for early analysis and optimization.

5.1 Artifacts

This work has contributed an additional compiler for the Lift high level language, which is readily available on Github. Producing this also involved working with the Lift project.

5.1.1 Lift Project

The Lift artifact is available on GitHub at github.com/lift-project/lift. This represents the work done by the Lift Project, excluding this compiler. We have helped make this more available by contributing a public docker container at hub.docker.com/repository/docker/hebehh/lift. Additionally, we have augmented the Lift artifact with a fork that will run on Mac (tested on Mojave and Catalina), and with a higher Java version (tested up to 11). This is available at github.com/HebeHH/lift.

5.1.2 CSDF Generator

The artifact described in this work, the CSDF compiler for the Lift high level language, is available on my GitHub at github.com/HebeHH/lift-to-csdf.

Bibliography

"Advanced Model-Based FPGA Accelerator Design" (2017). In: *FPGA-based Implementation of Signal Processing Systems*. John Wiley Sons, Ltd. Chap. 10, pp. 200–224. ISBN: 9781119079231. DOI: 10.1002/9781119079231.ch10. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119079231.ch10. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119079231.ch10.

Amdahl, Gene M (1967). "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, pp. 483–485.

Asanovic, Krste et al. (2009). "A view of the parallel computing landscape". In: *Communications of the ACM* 52.10, pp. 56–67.

Asanović, Krste et al. (2006). *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

Aubry, Pascal et al. (2013). "Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor". In: *Procedia Computer Science* 18, pp. 1624–1633.

Bachrach, Jonathan et al. (2012). "Chisel: constructing hardware in a scala embedded language". In: *DAC Design Automation Conference 2012*. IEEE, pp. 1212–1221.

Bodin, Bruno, Alix Munier-Kordon, and Benoît Dupont de Dinechin (2013). "Periodic schedules for cyclo-static dataflow". In: *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*. IEEE, pp. 105–114.

— (2016). "Optimal and fast throughput evaluation of CSDF". In: *Proceedings of the 53rd Annual Design Automation Conference*. ACM, p. 160.

Chandrakasan, Anantha P, Samuel Sheng, and Robert W Brodersen (1992). "Low-power CMOS digital design". In: *IEICE Transactions on Electronics* 75.4, pp. 371–382.

Darlington, John et al. (1993). "Parallel programming using skeleton functions". In: *International Conference on Parallel Architectures and Languages Europe*. Springer, pp. 146–160.

Fine Licht, Johannes de, Michaela Blott, and Torsten Hoefler (2018). "Designing scalable FPGA architectures using high-level synthesis". In: *ACM SIGPLAN Notices* 53.1, pp. 403–404.

Hammond, Kevin (2011). "Why Parallel Functional Programming Matters: Panel Statement". In: *Reliable Software Technologies - Ada-Europe 2011*. Ed. by Alexander Romanovsky and Tullio Vardanega. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 201–205. ISBN: 978-3-642-21338-0.

Khan, Hassan N, David A Hounshell, and Erica RH Fuchs (2018). "Science and research policy at the end of Moore's law". In: *Nature Electronics* 1.1, p. 14.

Mulpuri, Chandra and Scott Hauck (2001). "Runtime and quality tradeoffs in FPGA placement and routing". In: *Proceedings of the 2001 ACM/SIGDA*

ninth international symposium on Field programmable gate arrays. ACM, pp. 29–36.

Nane, Razvan et al. (2015). "A survey and evaluation of FPGA high-level synthesis tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.10, pp. 1591–1604.

Research, Industry (2019). *GLOBAL FIELD PROGRAMMABLE GATE ARRAY (FPGA) MARKET RESEARCH REPORT FORECAST TO 2025*. Industry Research.

Steuwer, Michel, Toomas Remmelg, and Christophe Dubach (2017). "Lift: a functional data-parallel IR for high-performance GPU code generation". In: *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, pp. 74–85.

Steuwer, Michel et al. (2015). "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code". In: *ACM SIGPLAN Notices* 50.9, pp. 205–217.

Stewart, Robert et al. (2017). "Profile guided dataflow transformation for FPGAs and CPUs". In: *Journal of Signal Processing Systems* 87.1, pp. 3–20.

Vlugt, Steven van der et al. (2019). "Modeling and analysis of FPGA accelerators for real-time streaming video processing in the healthcare domain". In: *Journal of Signal Processing Systems* 91.1, pp. 75–91.