High-Level Synthesis of Functional Patterns for Reconfigurable Logic

Martin Kristien

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2017

Abstract

Together with the rise of heterogeneous programming, FPGAs are becoming more and more popular. These devices can often outperform CPUs and GPUs while consuming less power. This is due to FPGAs bypassing software abstractions and executing applications directly in hardware. Programming FPGAs, however, requires extensive knowledge of computer design, which renders them inaccessible to most programmers.

To improve programmability of FPGAs, I have extended an existing compiler with a new back-end targeting FPGAs. The compiler, Lift, provides to user a high-level functional platform-agnostic interface, where user can describe applications at algorithmic level ignoring implementation details. The compiler was extended with new low-level patterns and hardware generator for these patterns. Note, transforming high-level platform-agnostic patterns to low-level platform-specific patterns is the question of future work.

Evaluation of generated hardware against numpy-based implementations of the same applications shows promising results. The best speedups are achieved for larger input sizes due to a large constant overhead involved in data movement to FPGA. Speedups of more than 15X were measured for multiplication of matrices of size 128KB * 4096KB using 64-bit integer data type.

Acknowledgements

I would like to thank my supervisor, Christophe Dubach, for helpful guidance and discussions during the course of the project and during the writing of this report.

I express gratitude to Nigel Topham for helping with setting up Linux on ZYBO board.

I also thank to Jan Horvath for discussing possible solutions to hardware design issues encountered during the project.

Furthermore, I am grateful to Zuz Olsinova for final proofreading of this report.

Lastly, thanks to all the people around me that offered small pieces of help throughout the course of this project.

Table of Contents

Chapter 1

Introduction

This report concerns work done as part of an Honours Project at The University of Edinburgh. The work was done in 4th year of undergraduate study towards BSc in Computer Science. The supervisor of the project was Christophe Dubach.

1.1 Motivation

In recent years, hardware and software developers observe a paradigm shift in design of modern computer systems. Clock frequency does no longer increase due to insufficient heat dissipation (i.e. power wall). Superscalar computers gain only diminishing returns from exploiting instruction-level parallelism. To gain further benefits from Moor's law, the computing industry has shifted to other forms of parallelism, namely data-level and task-level parallelisms. This shift results in emergence of multi-core processors and, by extension, heterogeneous systems. Such a system can be characterised as a combination of computational devices of different characteristics collaborating to boost the performance of the whole system.

In particular, increased programmability of graphics processing units (GPUs) allowed developers to exploit data-level parallelism to an unprecedented extend. However, mapping application logic to heterogeneous systems is a challenging task for developers. They need to explicitly expose application's fine and coarse grained parallelism, decide which parts of the application to be performed by which device, and synchronise the communication among all collaborating devices. Furthermore, developers must understand low level specifications of the system, which are often platform-specific, to yield the optimal performance. Due to these constraints, several researchers argue for better abstractions and platform compatibility [27, 6].

Field Programmable Gate Array (FPGA) is another type of device used in heterogeneous computing. It can be viewed as a re-programmable silicon. Despite a magnitude lower clock frequencies, FPGAs can match or outperform CPUs and GPUs computation while consuming less power. This is a result of FPGA's hardware design performing application logic directly, as opposed to indirect execution through interpretation of software instructions present in CPUs and GPUs. Note, Application Specific Integrated Circuits (ASICs) are even more performant than FPGAs, which is the price FPGAs must pay for being re-programmable. Programming FPGAs is, however, even more demanding than programming GPUs. The lack of abstraction and higher-level concepts in Hardware Description Languages (HDLs) requires FPGA developers to posses extensive knowledge of computer architecture and hardware design [6, 12].

1.2 Goals

We aim to create a framework for high-performance computing in heterogeneous environment while providing programmability through high-level platform-agnostic programming language. In particular, it should be possible to express an application at algorithmic level in a declarative way without any need to understand and/or choose the target platform. It should also be possible to decide (at compile time) which platform the application should be run at. Besides high-level application description, the framework should provide a good performance compared to specialised target-specific frameworks.

This project makes the first step towards this goal. It does so by extending an existing compiler framework, Lift [1], with back-end targeting FPGA-based platforms. The existing framework exposes to the programmer a functional language based on parallel patterns, such as map and reduce. Functional programming paradigm fits well with the aims of this project, as it is a high-level declarative approach. It is also easier to map into hardware design, as functional programs are based on composable substructures (i.e. functions) that can be directly translated to composition of functional hardware modules. Furthermore, functional programming avoids the problems of tracking global state and mutable data, which allows for representing applications as data flows. Capabilities of Lift framework are also aligned with the aims of this project. Lift front-end is platform-agnostic while providing a promising performance when targeting GPUs [24] by exploring the design space.

This project focuses on generating code from an intermediate representation of Lift program to be used in a heterogeneous system targeting FPGA. The generated code consists of hardware description files to be synthesised and used in the FPGA part of the system and software code to be used by the CPU as an interface for a user and driver of the FPGA logic.

1.3 Contribution

These are the main contributions I have made as part of this project:

- Extending Lift compiler with new FPGA-specific patterns
- Implementing compiler back-end to generate code for the target FPGA-based system from the new patterns

- Creating runtime for CPU and FPGA heterogeneous system. The runtime allows user applications inside an OS to use Lift generated code for FPGA computing.
  - Designing CPU-FPGA integration
  - Implementing driver of FPGA-based computing device
- Implementing framework to compile (synthesise) generated code and to apply the result in the target system
- Evaluating the FPGA-specific patterns against a CPU-based solution for several applications

1.4 Report Organisation

The rest of the report is organised as follows. Chapter 2 provides background to the concepts presented in this report. Chapter 3 presents the context of this project in terms of related work. It compares and contrasts different approaches to hardware generation with the approach taken in this project. Chapter 4 introduces the new content this project brings to the field. In particular, it presents the new functional patterns in Lift language and overviews how these patterns can be mapped to hardware design. Chapter 5 gives more detailed description of implementing the Lift back-end. Chapter 6 presents the actual target framework within which this project was developed and evaluated. It also reports the results of evaluation for several applications. Finally, chapter 7 summarises the work done in this project and proposes new directions the project might be taken into.

Chapter 2

Background

This chapter presents concepts the reader is required to be familiar with in order to understand the work done in this project. It starts by introducing compilation of functional programs and presenting overall structure of Lift compilation framework. The rest of the chapter is concerned with hardware design concepts, from hardware design languages to means of representing data in hardware.

2.1 Lift

Lift is a novel approach to achieving performance portability in heterogeneous environment. It brings a high-level functional language that allows programmers to describe application in a platform-agnostic manner. Lift compiler generates code for selected architecture through design space exploration, resulting in competitive performance.

2.1.1 Intermediate Representation

Most compilers are organised in a pipelined structure. When an intermediate representation (IR) of a program is generated, a compiler can perform multiple transformations (possibly into different IR) before generating target code. Note, target code can be an executable file or a programming language that can run on target platform.

Lift represents functional programs as graphs [1]. Nodes of the graphs can be of two main classes, `Expr` (expression) and `FunDecl` (function declaration). Class diagram of the Lift IR is depicted in Figure 2.1.

`Expr` class corresponds to values and have associated type. The main subclasses of `Expr` are `Param` (parameter) and `FunCall` (function call). `Param` is a value provided by the context of an expression (e.g. user) while `FunCall` corresponds to application of a function to a number of arguments. Note, since the type of arguments in `FunCall` is `Expr`, arguments can be another `FunCall` node, allowing chaining of function calls. Lift uses two classes of `Type` that are relevant to this project. `Scalar` type corresponds

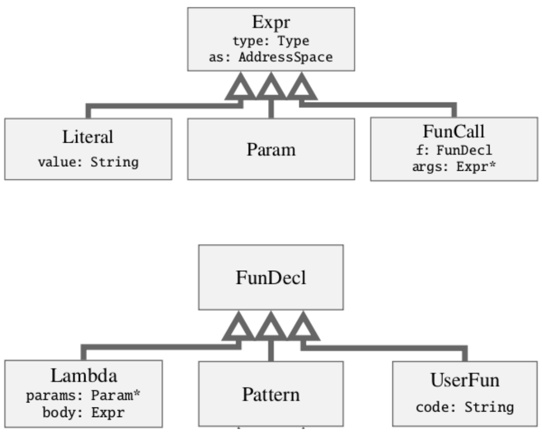


Figure 2.1: Class diagram of Lift IR. Source [24].

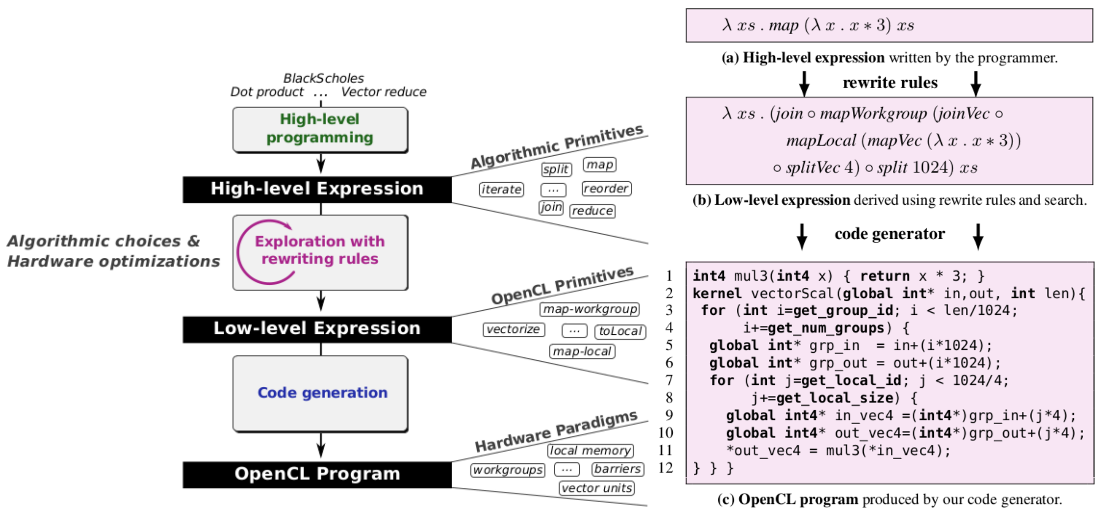
to scalar values. Each `Scalar` type has associated size corresponding to the number of bytes required to represent the type. `Array` type represents a sequence of elements. Each `Array` type has associated element type (type of each element in the sequence) and size (length of sequence). Element type can be either `Scalar` or `Array`, allowing for construction of multi-dimensional arrays.

`FunDecl` class corresponds to function declaration. This can be either a generic `Lambda` expression, functional `Pattern`, or user-defined function `UserFun`. It is the `Pattern` class that determines the level of abstraction of IR. High level patterns in Lift are platform-agnostic and are used by programmer to describe the program. These are well known patterns often used in other languages, such as *Map*, *Reduce*, *Split*, or *Join*. Each high-level pattern can be expressed by one or more low-level patterns. While low-level patterns preserve the semantics of the corresponding high-level patterns, they affect the generated code and thus are specific to the target architecture. It is the purpose of this project to introduce new low-level patterns to the Lift compiler and implement back-end for these patterns.

2.1.2 Compilation Flow

Lift programs are described as `Lambda` function declarations. Since Lift is a functional language, Lift compiler can decide on how to map semantics of programs to the target architecture. It does so by applying semantically consistent rewrite rules to high-level IR, in order to explore the design space. The rewrite rules can lower the abstraction of IR, effectively deciding on what architectural primitives will participate in performing the computation described by the program.

The example in Figure 2.2 demonstrates different levels of abstraction of Lift programs. In 2.2a, application is expressed as a map of multiply by 3 function applied to the input data. Note, the expression in 2.2a does not specify how the function should be computed. As a result of applying rewrite rules, Lift compiler generates semantically identical representation of the input program 2.2b. This representation uses low-level patterns that specify how the function should be executed using primitives of the target



$\lambda\, xs\, .\, map\ (\lambda\, x\, .\, x * 3)\ xs$

(a) High-level expression written by the programmer.

$\lambda\, xs\, .\, (join \circ mapWorkgroup\ (joinVec \circ mapLocal\ (mapVec\ (\lambda\, x\, .\, x * 3)) \circ splitVec\ 4) \circ split\ 1024)\ xs$

(b) Low-level expression derived using rewrite rules and search.

```
int4 mul3(int4 x) { return x * 3; }
kernel vectorScal(global int* in,out, int len){
 for (int i=get_group_id; i < len/1024;
      i+=get_num_groups) {
  global int* grp_in  = in+(i*1024);
  global int* grp_out = out+(i*1024);
  for (int j=get_local_id; j < 1024/4;
       j+=get_local_size) {
    global int4* in_vec4 =(int4*)grp_in+(j*4);
    global int4* out_vec4=(int4*)grp_out+(j*4);
    *out_vec4 = mul3(*in_vec4);
} } }
```

(c) OpenCL program produced by our code generator.

Figure 2.2: Lift compiler pipeline with example of IR transformation and code generation targeting OpenCL. Source [24].

system. The last stage of the Lift compiler pipeline is a straight-forward OpenCL code generation.

This project adds new low-level patterns to Lift framework and implements the last stage of the compiler pipeline, namely code generation, for these patterns targeting FPGA.

2.2 Target Languages

Since this project targets heterogeneous system, namely combination of CPU and FPGA, generated code must span both target devices. This section introduces languages used to describe CPU (host) side and FPGA side of the system.

2.2.1 Host

Language `python` was chosen to create an entry point for user to start a Lift application. As `python` is a high-level language, it fits well with the goals of this project. `Python` extension `numpy` is used to represent and manipulate data. In particular, user is expected to provide input data as `numpy` array and application's output is also a `numpy` array. Arrays in `numpy` are implemented using fixed length continuous memory and have associated type information. This is useful for type-checking and for transferring data to and from FPGA using direct memory access.

`Numpy` is also a standard framework for scientific computing, thus many users are already familiar with it. Its back-end uses pre-compiled optimised C implementations of several data operations that this project explores (i.e. matrix slicing, transpose, matrix multiplication), thus it can be used by CPU side of Lift generated applications. Furthermore, it serves as adequate benchmark comparison for the whole applications.

2.2.2 FPGA

The most common hardware description languages are VHDL and Verilog. In this project, VHDL language was chosen to design FPGA part of the system. It is a low-level hardware description language, which allows for direct control of generated hardware by compiler. It is also strongly typed, which allows for detections of more errors in designed data flows at compile time. Furthermore, designer can create custom types and structures to alias composed types (e.g. arrays) and to represent tuples.

Functional units in VHDL are described by *entities*. Each entity can be viewed as a black box with well defined static boundary interface. Entities can instantiate other entities inside their body and connect own context with the instantiated entity's context using the boarder interface. This leads to hierarchical structure of the hardware design.

In VHDL, data is represented using *signal*s, which can be of arbitrary type. The two basic types in VHDL are *std_logic* and *std_logic_vector*, representing single bit and multi-bit data, respectively. Note, other basic types carrying similar semantics can be used in VHDL, such as *boolean*, *integer*. The data carries either combinatorial or sequential information, depending on the context it is used in.

2.3 Hardware Design

This section concerns designing systems on the FPGA. First, architecture of FPGA is introduced. Then, different ways of representing data in hardware are presented.

2.3.1 FPGA

A Field-Programmable Gate Array is an integrated circuit that can be configured after it has been manufactured. It is usually constructed using an array of programmable logic blocks and hierarchy of interconnects to wire together the logic blocks. Each logic block can perform complex combinatorial or sequential operation. Furthermore, FPGAs often include distributed memory for data storage in form of simple flip-flops or larger memory blocks (BRAM).

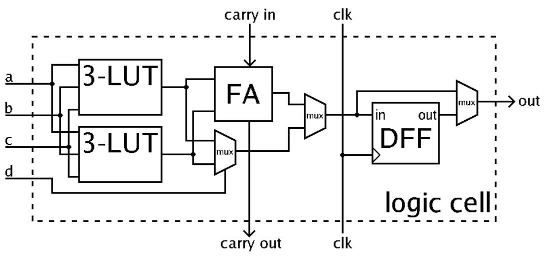


Figure 2.3: FPGA cell composed of look-up table, full adder, and flip-flop [11]

Logic blocks are constructed using several logic cells (for scheme of a logic cell see Figure 2.3). Typically, a logic cell consist of 4-input look-up table (LUT), full adder

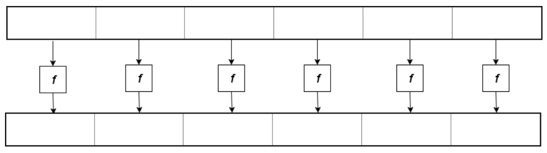
(FA), and D-type flip-flop (FF). LUTs perform combinatorial logic, FA can perform addition if cell is used in math mode, and FF performs sequential logic. In Figure 2.3, LUT is split into two 3-input LUTs and the fourth input is controlling the left-most multiplexer. Switching between normal and math modes is done by the middle multiplexer. Switching between combinatorial and sequential output mode is done by the right-most multiplexer.

FPGA logic can be designed using a hardware description language, similarly to designing application-specific integrated circuits (ASIC). Hardware design is synthesised into circuitry (netlists) which can be converted to bitstream by place-and-route process which is specific to the FPGA being targeted. The bitstream configures all reconfigurable elements of FPGA. Bitstreams are usually generated by software provided by FPGA's manufacturer.

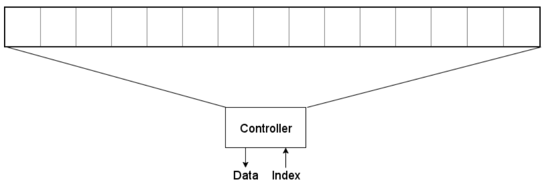
2.3.2 Array representation

Sequences of data can be represented in multiple ways in VHDL. The representations differ in the way elements of sequence can be accessed.

As discussed in Section 2.2.2, VHDL allows designer to create an array type as a collection of element types. This representation permits accessing arbitrary and/or all elements of sequence in a singe clock cycle (or even combinatorially). Accesses, however, incur significant cost for larger arrays in terms of logic delay. For example, reading an arbitrary index (not statically known) of an array requires a multiplexer, which can result in substantial propagation delays in case of large arrays. This representation, however, allows a single cycle access to all elements, thus is suitable for application of naturally parallel operations, such as *map* (see Figure 2.4a).



(a) Array-based representation



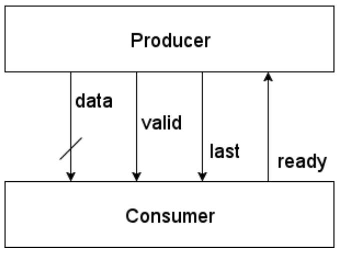
(b) Block RAM based representation

Figure 2.4: Visualisation of sequence access in array and BRAM representations

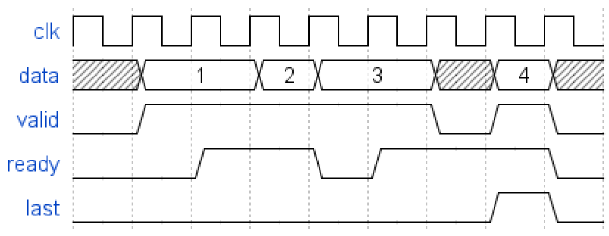
FPGAs also often provide distributed blocks of RAM, which can be instantiated to store arbitrary data. Sequences of data can be stored in this block RAM. BRAM-based arrays can only be accessed one element at a time (per clock cycle) (see Figure 2.4b). In this access pattern, index of the array can be interpreted as a memory address. The advantage of using BRAM is the utilisation of FPGA elements optimised for storage, thus BRAM is more suitable for large arrays than VHDL array.

Sequences can also be represented as stream of elements. Stream arrays can be accessed only one element per clock cycle and the elements can be only accessed in the strict order of occurrence in the array. This streaming framework operates with two actors exchanging the data, namely *producer* and *consumer*. Synchronisation of the two is provided using control signals *valid*, *last*, *ready*, together implementing a streaming protocol. Note, once a data element is consumed it cannot be reproduced again, thus all designs reusing data must cache it.

According to the streaming protocol, producer must assert valid signal when valid data is available without waiting for consumer to assert ready signal. Producer cannot desert valid signal until the consumer has consumed the data, as indicated by asserting ready signal. Note, data is being transferred in a clock cycle when both valid and ready signals are asserted. Last signal is asserted by producer when the current data element is the last element in sequence being transferred. Figure 2.5b depicts transfer of sequence of length 4 using the streaming protocol.



(a) Streaming communication



(b) Stream sequence transfer example

Figure 2.5: Visualisation of sequence access in array and BRAM representations

Chapter 3

Related Work

Many other projects try to improve programmability of hardware. The main approaches seen in industry are provision of hardware libraries, development of high-level hardware languages, translation of low-level software languages into hardware description, and translation of high-level software languages into hardware.

3.1 Hardware Libraries

Many standard operations and data types can be imported to hardware design as a library. Often, EDA (Electronic Design Automation) tools directly provide optimised version of primitive operations, such as addition or multiplication. Alternatively, programmer can import a library. For example, library `ieee.fixed_float_types` [8] provides programmer with fixed point type and arithmetic. These libraries greatly improve programmability as programmer can use a single + operator instead of manually designing the complex fixed point addition logic. However, the programmer must still express the application and connection of library functions manually at low level.

3.2 Hardware Language Abstractions

Many projects increase the level of abstraction of hardware design languages and introduce high-level programming concepts. For example, SystemVerilog [3] is a major improvement compared to Verilog or VHDL. It introduces object-oriented programming, creates strong type system, and allows parametrisation of hardware modules. However, it has limited support among EDA tools. Bluespec [20] builds on SystemVerilog by providing dedicated compiler. It also represents functional modules at behavioural level using concurrent finite state machines. Esterel [15] creates abstraction of Verilog control flow elements, reducing code size and producing optimised circuits. Chisel [5], a hardware description language embedded in Scala, abstracts hardware design even more. It provides high-level concepts such as object-oriented and functional programming, type inference and type parametrisation. Furthermore, it can generate fast C++ simulator for a given application.

All these approaches, however, require the programmer to understand hardware design concepts. Moreover, designed application is directly mapped to hardware without much compiler optimisation. Therefore, achieving good performance is hard for any application of moderate size.

3.3 Low-level Software Languages

Another approach seen in industry is to express application logic as software program and then map the logic to hardware.

3.3.1 C-based

Most EDA tools developed or acquired frameworks to compile C-based program into hardware. These include Xilinx Vivado (originally AutoPilot) [30], Cadence C-to-Silicon [7], Synopsys Synphony C compiler [25], and Mentor Graphics Catapult C [9]. The greatest advantage of hardware generation from C is that software programmers can design hardware, which is a great step in programmability of FPGAs. C-based approaches, however, have limited power to express coarse and fine grained parallelism in application, synchronisation and communication, and bit-level data precision, all of which are well known concepts in hardware design. Furthermore, C and C++ have software specific constructs, such as dynamic memory management, pointers, and recursion, which complicate designing hardware.

3.3.2 CUDA/OpenCL-based

CUDA and OpenCL are parallel computing platforms mainly targeting GPUs. Several projects generate hardware from CUDA [23] or OpenCL [21, 13, 17, 22] programs. Since these platforms already target heterogeneous devices, it is possible to express coarse and fine grained parallelism of applications. The platforms are, however, still low-level and generally harder to program than C or C++. The programmer is also required to express the parallelism of application, which requires good understanding of target architecture to result in optimal performance. Furthermore, no performance portability across devices is guaranteed.

3.4 High-level Approach

Vuletic and Dubach [26] managed to utilise FPGA accelerator with oblivious user by embedding FPGA driver in java virtual machine. Through the power of virtualisation they managed to seamlessly integrate software and hardware. Furthermore, they managed to implement software concepts, such as memory allocation and recursion, in hardware context through sharing virtual memory of an application with the hardware accelerator.

Project Kiwi [16] raises hardware design abstraction to the level of high-level parallel imperative language. It allows user do describe application using standard library in C# for multi-threading programming. It, however, is an imperative language leaving little space for optimisation and design space exploration inside compiler.

LiquidMetal [4] introduces a functional language for programmer to describe application in a declarative way. This is a co-execution system that automatically splits application into tasks that can run in software or hardware parts. It also seamlessly integrates all computing devices to participate in execution of application. Although LiquidMetal uses a declarative front-end, its compiler only explores runtime combination of devices in heterogeneous system to perform application's tasks. It is unclear how to express design space exploration for individual devices.

TyTra project [19] uses Idris [10] functional program as front-end to generate Verilog. TyTra is the most similar project to this project as it uses type transformation to explore design space and cost model to evaluate program variants. However, it implements the whole application as FPGA computation instead of splitting it into software and hardware tasks.

3.5 Summary

Programmability gradually improves with the level of abstraction. Starting with hardware libraries, programmer's effort is reduced which leads to faster development, fewer programming mistakes, better code quality, and more optimal hardware. Generating hardware from software description of an algorithm further widens the range of programmers that can design hardware. Low-level software, however, is often sequential, which makes it hard to express parallelism that could be utilised by hardware design. Alternatively, programmer can use parallel languages to describe the application's parallelism. Low-level software, however, is still hard to program and functional and performance portabilities are not guaranteed. The next step is to enable hardware generation from high-level platform-agnostic description. Such level of abstraction requires compilers to bridge the big gap between description of an algorithm and hardware design.

This project is developed in exactly such environment, enabling user to describe application in a high-level platform-agnostic manner, exploring hardware design space to find optimal way to map application's logic to hardware, and generating optimal integration and implementation of software and FPGA-base computation.

Chapter 4

Methodology

This chapter presents original work done in this project. Unless stated otherwise, all work presented is my own. The work is presented here at conceptual level and particular details of implementation are postponed to the following chapter. Note, the system design is tightly connected to implementation, thus some implementation details relevant to the conceptual design are introduced in this chapter.

The chapter starts by introducing a working example used to demonstrate the design of this project. First, the example is presented at high level of abstraction avoiding any implementation-specific concepts. The chapter continues by including some implementation-specific concepts necessary for reasoning about semantics of new functional patterns. Then, new functional patterns added to Lift are presented in terms of functional type signature and semantics. Finally, the working example is revisited at the end of the chapter. Here, the application is represented using the new patterns developed in this project.

4.1 Working Example: High-Level

The working example corresponds to dot product application. Here, the application is expressed in Lift code that directly corresponds to the application's IR. The code uses only high-level patterns that would be used by programmer. At the end of this chapter, the same application is expressed using new low-level patterns targeting FPGA.

Listing 4.1: Dot product in Lift high-level patterns

```
program = (
  (u, v) => Reduce(+,0) o Map(*) o Zip(u, v)
)
```

In the example application in Listing 4.1, `program` is a function declaration that takes two arguments, `u` and `v`. Type declaration of the arguments is omitted for simplicity. It is assumed both arguments are one-dimensional arrays of 1024 64-bit integers. Note, functional programs must be read from right to left, i.e. the last function in program is the first function to be executed. In the working example, two input arguments are zipped into an array of pairs. Then, multiplication operation is applied to each pair in the array, resulting in an array of integers. Finally, all elements (integers, not tuples) are summed together to give the final result.

4.2 Implementation Overview

There is a big gap to be filled by compiler between functional patterns and hardware design. This section overviews how to map concepts in functional abstraction of an algorithm to hardware generated in this project.

4.2.1 Patterns to Hardware Design

Generated hardware code is represented as a top level entity in VHDL. First-order functional patterns are enclosed in separate entity definitions and the entities are instantiated in the top level entity. The top level entity has a fixed interface for consuming and producing data in stream representation (Figure 4.1). At the inputs side (top), the top entity acts as consumer of stream and at the output side (bottom) as producer of stream (for stream protocol description, see section 2.3.2).

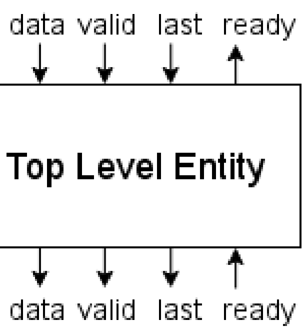
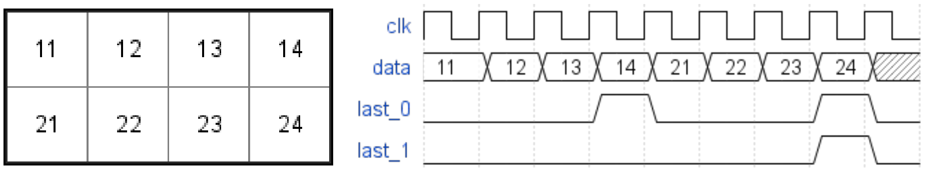


Figure 4.1: Module lift_device with its interfaces

Higher-order functional patterns (i.e. patterns that take arbitrary functions as arguments) are implemented in global scope (i.e. in the top level entity). These include Map, Reduce, and Let patterns. Since these patterns must instantiate arbitrary functions that can access expressions in global scope, enclosing the patterns in separate entities would require to provide the required context to the entities, which would violate uniform entity interface and complicate the overall design.

4.2.2 Multi-dimensional Stream Arrays

As described in section 2.3.2, the end of sequences in stream representation can be signalled by asserting `last` signal. This, however, permits only one dimensional sequences. In order to represent multi-dimensional sequences in stream representation, the last signal was extended to be a vector of last signals. Each dimension of this vector can denote end of sequence along corresponding dimension. For example (Figure 4.2), 2-dimensional matrix sequence can use one dimension of last vector to denote the end of rows and another dimension to denote the end of the whole matrix.



(a) Logical view

(b) Last signal vector for stream sequence

Figure 4.2: Multi-dimensional array

4.3 Functional Patterns

This section describes functional patterns that were introduced to Lift. Two patterns operate at the hardware-software interface, namely *ToFPGA* and *ToHost*. These can be used to move data between the main memory and the FPGA. Other patterns operate at either software or hardware side. Each pattern is described in terms of type signature and semantics. Type signature uses the following notation:

- $[T]_s$ - array of arbitrary element type T and size s
- $[T]_s^{stream}$ - array of arbitrary element type T and size s in *stream* representation
- S_s - scalar type of size s bytes

For hardware side patterns, block diagrams are often present as well to depict the logical view of the hardware design underlying the patterns' semantics. For software side patterns, python code implementing the patterns is listed directly.

4.3.1 Host-FPGA Interface

Patterns in this section represent operations used for movement of data from the main memory to FPGA and vice versa.

4.3.1.1 ToFPGA

$ToFPGA() : [S_8]_n \rightarrow [S_8]_n^{stream}$

This pattern does not require any parameters and accepts a single argument of type *Array* of which base type is *Scalar* of size 8 bytes. This pattern, as the name suggests, initialises movement of data into FPGA without any modification to the data. Note, the output type is identical to the input type with addition of *stream* attribute. The basetype size is required to be 8 bytes to match the data bus width of 64 bits used for data transfer between main memory and FPGA (see Chapter 6 for details).

As a result of using this pattern Lift compiler generates both software side code and hardware description code. Software side code initialises DMA transfer from the main memory to FPGA. Hardware side code creates an entry point for reading the data in

FPGA. In Lift IR, argument to the corresponding function call is generated in software context.

VHDL code top level entity has only one input and only one output port for communicating data from and to the main memory. If application requires two pieces of data to be present in FPGA at the same time it is not possible to access the two pieces of data by connecting to a single input port directly.

To resolve this issue, input demultiplexing is introduced at the input port of the top level entity. This way, demultiplexer can send different data to different parts of the FPGA logic. The demultiplexing selection switches upon observing last signal on the input port. In Lift compiler, every occurrence of ToFPGA pattern in application's IR registers an input consumer. Demultiplexer then sends input data to consumer in order they registered for the input, i.e. in order of occurrence in IR.

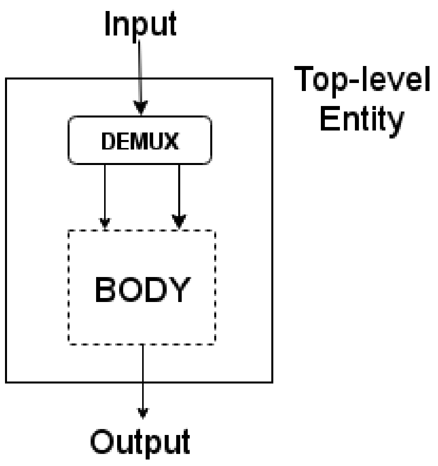


Figure 4.3: Scheme of input demultiplexing

4.3.1.2 ToHost

$ToHosT() : [S_8]_n^{stream} \to [S_8]_n$

This pattern does not require any parameters and accepts a single argument of type *Array_Stream* of which base type is *Scalar* of size 8 bytes. *ToHost* pattern is symmetrical to *ToFPGA*, only it moves the data from FPGA back into the main memory.

As a result of using this pattern Lift compiler generates both software side and hardware side codes. Software side code allocates space in the main memory to store the result and initialises DMA transfer from FPGA to the main memory. Hardware side code creates a channel for writing data from FPGA. In Lift IR, argument to the corresponding function call is generated in hardware context.

Note, no multiplexing is done at the output port of the top-level entity. In this version it is assumed output of the FPGA part of the application is produced by only one producer entity.

4.3.2 Host Patterns

This section concerns patterns to be used in the host side of application. Using following patterns in FPGA part of application would result in compilation error.

4.3.2.1 Split

$Split(n) : [T]_s \rightarrow [[T]_n]_{s/n}$

Split pattern requires one parameter `n` of integer type. This pattern can be used to add dimensions to arrays. Parameter `n` specifies the size of the inner dimension. For example, `Split(2)` acting on the array of size 16 results in the array of size 8, of which each element is an array of 2 elements of the original array. Note, `size` must be divisible by `n`. In python, this pattern is implemented by changing the view into data by reshaping without any data movement.

Listing 4.2: Application of Split pattern

```
arg.shape = (arg.shape[0]/n,n) + arg.shape[1:]
```

4.3.2.2 Join

$Join() : [[T]_{n1}]_{n2} \rightarrow [T]_{n1*n2}$

This is a counterpart to Split pattern, as it removes one dimension form a multi-dimensional array by joining the two outer-most dimensions. Join pattern does not require any parameter, as it can infer size of the resultant dimension from size of the two dimensions being joined. For example, `Join()` acting on two-dimensional array of size (8, 2) results in one-dimensional array of the same base element type and of size 16. In python, this pattern is implemented by changing the view into data by reshaping without any data movement.

Listing 4.3: Application of Join pattern

```
arg.shape = (arg.shape[0]*arg.shape[1],) + arg.shape[2:]
```

4.3.2.3 Flatten

$Flatten() : [...[[T]_{s_n}]_{s_{n-1}}...]_{s1} \rightarrow [T]_{\prod^n s_i}$

Flatten pattern has similar semantics to Join pattern. It, however, acts across all dimensions of the input array, resulting in one dimensional array. The size of the resultant array is inferred as product of sizes of all dimensions of the input array. For example, `Flatten()` acting on three-dimensional array of size (4,3,2) results in one-dimensional array of the same base element type and of size 24. It is useful for recasting the data type before sending it to FPGA (see ToFPGA signature). In python, this pattern is implemented by changing the view into data by reshaping without any data movement.

Listing 4.4: Application of Flatten pattern

```
arg.shape = (-1,)
```

4.3.2.4 Transpose

$Transpose() : [[T]_{s1}]_{s2} \rightarrow [[T]_{s2}]_{s1}$

Transpose pattern swaps the two outer-most dimensions of the input array. As opposed to previous patterns that only change the view of the data, Transpose transformation requires changing the layout of data in memory. This is necessary as data might be sent to FPGA later in program, thus it must be stored in memory in the order in which it is to be read. The transformation is done in place and thus is a destructive operation.

Listing 4.5: Application of Transpose pattern

```
temp = arg.view()
arg.shape = (arg.shape[1], arg.shape[0]) + arg.shape[2:]
arg[:] = np.transpose(temp, [1,0] + range(2, arg.ndim))
```

4.3.3 FPGA Patterns

This section presents patterns used at FPGA side of application. As described in section 4.2.1, patterns can be implemented in separate modules/entities or in global scope. When depicting hardware design schemes, entity boundaries are depicted using boxes with full boundary. Logic inside full boundary box is encapsulated in terms of scope an can only access signals inside the box or at the box's interface. Global scope patterns are depicted using only logical boundaries, represented by dashed boxes. As opposed to full box, logic inside dashed box can access signals outside of the box.

4.3.3.1 ToArray

$ToArray() : [T]_s^{stream} \rightarrow [T]_s$

ToArray pattern is used to convert sequence from stream representation to VHDL array representation. It does so by consuming the input stream one element at a time and filling VHDL array. Once the array is full it can be passed to the output by asserting corresponding valid signal. Note, even though output sequence is not represented as stream, signals of stream protocol (valid, ready) are used for controlling data flow.

4.3.3.2 FromArray

$FromArray() : [T]_s \rightarrow [T]_s^{stream}$

FromArray is a symmetric pattern to ToArray. It converts VHDL array representation of sequence to stream. It does so by outputting individual elements in order of occurrence to produce an output stream.

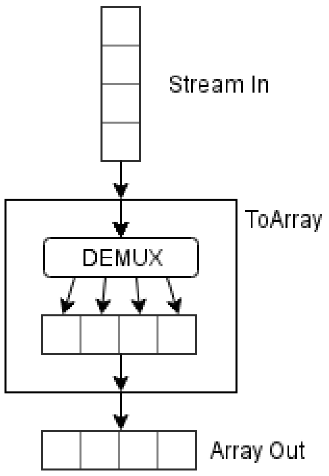


Figure 4.4: Scheme of ToArray pattern

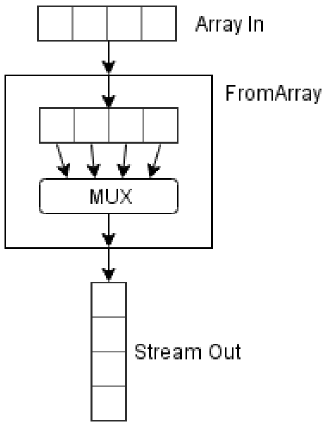


Figure 4.5: Scheme of FromArray pattern

4.3.3.3 Pipe

$Pipe() : T -> T$

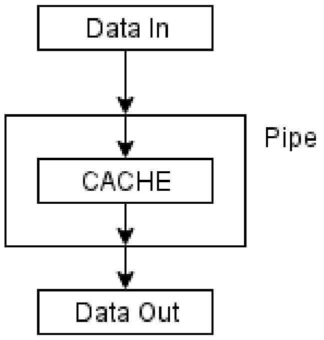


Figure 4.6: Scheme of Pipe pattern

Pipe pattern creates a pipeline stage in data flow without any modification to data. It does so by caching the input data together with associated data flow signals in registers, effectively postponing propagation of data by 1 clock cycle. As some patterns may result in combinatorial operation on data, Pipe may be required to split demanding combinatorial operations into multiple stages.

4.3.3.4 Zip

$ZipStream() : [T]_s^{stream} \rightarrow [T]_s^{stream} \rightarrow [(T,T)]_s^{stream}$

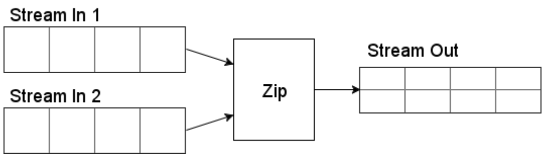


Figure 4.7: Scheme of ZipStream pattern

ZipStream pattern can be applied to two arrays in stream representation to create an array of pairs. It is required that the base types of the two input arrays are identical and that the sizes of the two arrays are equal. ZipStream pattern not only reshapes the input data into tuples, it also synchronises consuming of the two input sequences, as the output sequence has only one set of stream protocol control signals.

4.3.3.5 Let

$Let(f) : T \rightarrow f(T)$

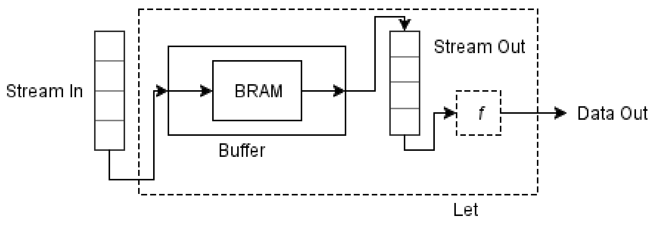


Figure 4.8: Scheme of Let pattern

Let pattern takes a function declaration as parameter to be body of Let. It then applies the body to an input argument. In the body of Let, an input argument x can be read multiple times. Note, otherwise, reading any data would consume the data, thus it could be read only once. Let caches the data in Buffer module, which instantiates block RAM on FPGA to store the input data in. The same data can then be repeatedly read from BRAM until the Buffer is reset. Buffer reset occurs when the Let's body has produced the final output, as indicated by the corresponding last signal in the streaming protocol.

Note, Let pattern is implemented in global scope, as indicated by dashed boundary. This is necessary, since body f can be an arbitrary function potentially accessing data in global scope.

4.3.3.6 SplitStream

$SplitStream(n) : [T]_s^{stream} \to [[T]_n^{stream}]_{s/n}^{stream}$

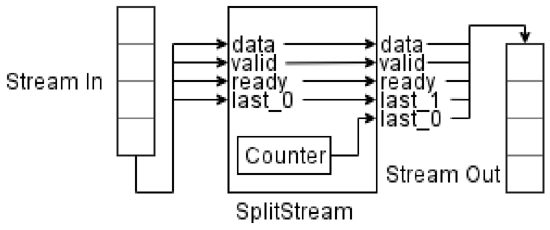


Figure 4.9: Scheme of SplitStream pattern

SplitStream is hardware equivalent of software Split, only it acts on sequences in stream representation. As discussed in section 4.2.2, multi-dimensional arrays in stream representation are represented using multi-dimensional last signal. Therefore, splitting stream array requires generating the inner dimensional last signals. For example, SplitStream(4) pattern applied to one-dimensional array of size 16 produces last signal corresponding to the new inner dimension using up to 4 counter.

4.3.3.7 JoinStream

$Join() : [[T]_{s1}^{stream}]_{s2}^{stream} \to [T]_{s1*s2}^{stream}$

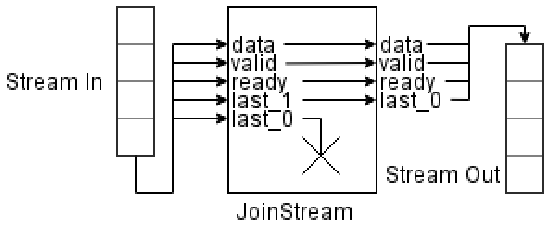


Figure 4.10: Scheme of JoinStream pattern

JoinStream is hardware equivalent of software Join, only it acts on sequences in stream representation. As opposed to SplitStream pattern, applying JoinStream pattern results in discarding the last signal corresponding to the inner dimension of the input array, resulting in one-less-dimensional output.

4.3.3.8 MapStream

$MapStream(f) : [T]_s^{stream} \to [f(T)]_s^{stream}$

MapStream accepts a function declaration as parameter. Application of MapStream pattern results in application of function f to all elements of the input array. Since the function f is to be applied to elements of the input array, last signal corresponding

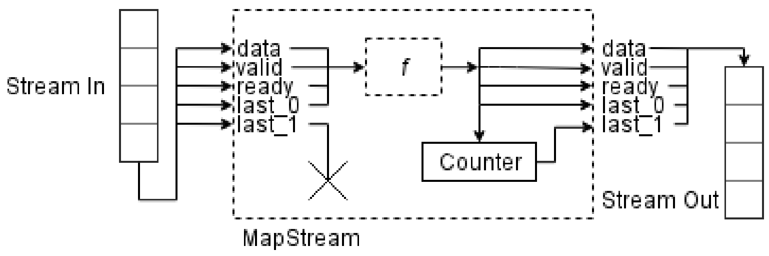


Figure 4.11: Scheme of MapStream pattern

to the outermost dimension is discarded and only inner dimensional data is passed as input to *f*. In the output, outer-dimension last signal must be restored. This is done using a counter that counst the elements of inner dimension. For example, applying MapStream to two-dimensional array of shape (4,2) discards last_1 signal (outer dimension) and effectively passes four two-dimensional arrays to function *f*. When function *f* produces 4 elements, as indicated by 4 occurrences of last signal coming from output of *f*, MapStream asserts last_1 signal to the result. Note, similarly to Let pattern, MapStream has only logical boundaries (i.e. is not enclosed in separate entity) to allow function *f* to be arbitrary, giving it access to signals in global scope.

4.3.3.9 ReduceStream

$ReduceStream(f, init) : [T]_s^{stream} \rightarrow [f(T, init)]_1$

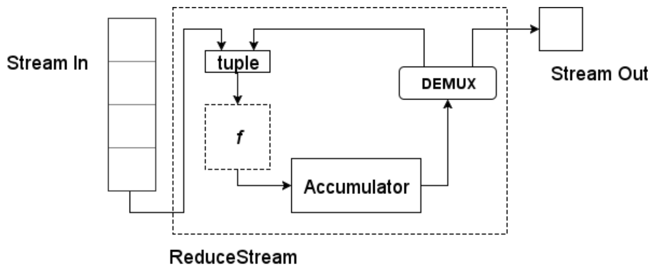


Figure 4.12: Scheme of ReduceStream pattern

ReduceStream pattern reduces a sequence of multiple elements into a sequence of one element. It does so by repeated application of two-input function *f* to all elements in the input sequence, storing the output in accumulator, which is fed back into the function *f* as one of its two inputs. When the whole input sequence is consumed, as indicated by corresponding last signal, ReduceStream outputs the value of accumulator. ReduceStream also accepts an initial value *init* as parameter, which is used to initialise accumulator. Similarly to Let and MapStream pattern, ReduceStream has only logical boundaries to allow arbitrary function *f* to be reduced over.

Note, no explicit synchronisation of inputs stream and accumulator is used. This design, thus, assumes function *f* does not introduce any clock cycle delay. Otherwise, elements from input stream could be consumed with old values of accumulator, resulting in incorrect semantics. Future designs might introduce explicit synchronisation in a form similar to application of Zip pattern.

4.3.3.10 UserModule

$UserModule() : T\_in- > T\_out$

UserModule has user-defined signature and functionality. Is can only be implemented as a separate entity, encapsulating the logic of UserModule. This pattern is to be used for custom simple modification of data that do not require any context (i.e. global scope).

4.4 Working Example: Low-Level

This section revisits the working example introduced in Section 4.1. This time, the same application is expressed using the new low-level patterns introduced in this chapter in Section 4.3. Dot product expressed in Lift using low-level FPGA-specific patterns is depicted in Listing 4.6.

Listing 4.6: Dot product in Lift

```
add = (x => UserModule.Addition() $ x)        # x is of type Tuple
mul = (x => UserModule.Multiplication() $ x) # x is of type Tuple
program = (
  (u, v) => ToHost() o Let(
    x =>
    ReduceStream(add, 0) o MapStream(mul) o
    ZipStream(x, ToFPGA() $ v)
  ) o ToFPGA() $ u
)
```

In the example above, addition and multiplication operations are expressed using `UserModule` pattern implementing custom transformation to the input data. The program starts by sending argument `u` to the FPGA, where it is stored using `Let` pattern. The argument can be referred to in the body of `Let` as `x`. Then, second argument is sent to the FPGA, where it is zipped with the first argument. Zipped arguments are pair-wise multiplied and the resultant sequence is reduced using addition operation. At the end, the program sends the result back to host using `ToHost` pattern. Block diagram corresponding to the program in Listing 4.6 is depicted in Figure 4.13.

Figure, 4.13 demonstrates composability of patterns in Lift. Aligned with the paradigm of functional programming, Lift compiler has strong type inference. Type-checking ensures pattern composition follows the type transformations inside the compiler. Similarly to compiler inference of type, hardware design closely follows type transformations in the data flow, resulting in a valid composition at hardware design level as well.

In the example in Figure 4.13, there is no explicit control unit to orchestrate the execution of individual modules. All control is performed by individual entities through

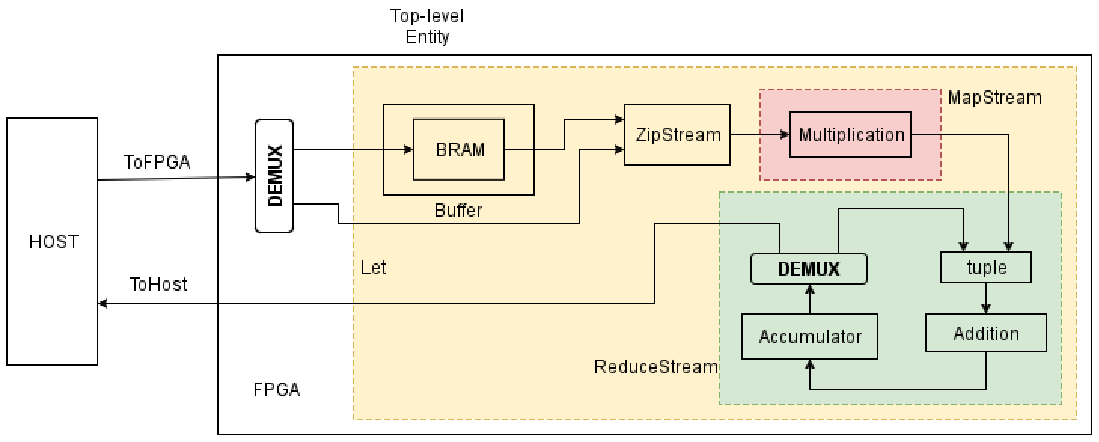


Figure 4.13: Dot product application in hardware design

controlling the streaming protocol signals. This design results in fully distributed control based on interactions of producers and consumers in the data flow.

4.5 Summary

This chapter introduced the conceptual design of this project. First, design of the resultant system was overviewed to provide the reader with information necessary to understand the new functional patterns in Lift. Then the new functional patterns were described in terms of type signature, semantics, and mapping of the semantics to the target system. The next chapter describes the details of implementation of the most important aspects of the system.

Chapter 5

Implementation Details

This chapter concerns implementation details of the most important aspects of the system. In particular, it starts by explaining how type information is expressed in the target system. Then, it presents implementation of FPGA code generator. The chapter also presents solutions to implementation challenges encountered in this project that arose from composition of patterns into more complex applications.

5.1 Distributed Control

As mentioned in Section 4.4, control logic in generated hardware design is distributed into entities controlling corresponding streaming protocol signals. In particular, besides data bus each entity can control three streaming signals (assuming one-dimensional streaming data), namely `ready_in`, `valid_out`, `last_out`.

This control is often implemented using state machines. States in these state machines correspond to particular state of operation of a given entity. For example, Buffer entity instantiated as part of Let pattern has two states. In the initial state s0, Buffer operates in consumer mode. It accepts data and writes it into a Block RAM. State transition to s1 is triggered when the allocated BRAM is full, as indicated by write-specific index reaching the size of the allocated BRAM. In the second state, Buffer operates in producer. It reads data from BRAM and sends it to following entity. No transition is performed unless a reset signal is observed, when the state machine resumes its initial state. Assignment of the control signals for each state in Buffer entity is presented in Table 5.1.

state	ready_in	valid_out	last_out	transition to s0	transition to s1
s0	1	0	0	default	write_index==size
s1	0	1	read_index==size	reset	default

Table 5.1: State machine for control in Buffer entity

5.2 Type-Checking

Generated application exposes python interface to user as an entry point. This entry point encapsulates the whole Lambda function declaration corresponding to generated application. When user starts the application, the parameters of the Lambda function declaration (i.e. arguments of python entry point method) are type-checked. Throughout the rest of the system, type information is not being explicitly expressed in the generated code. Type information is only present in Lift compiler to enable correct transformations to be performed on expressions. It is otherwise assumed all Lift compiler infers correct data for all operations and thus, provided the input arguments are properly type-checked, correct type propagates throughout the system.

5.3 VHDL Templates

Since VHDL language is more verbose than other hardware description languages and Lift compiler can generate VHDL code in a single pass through application's IR (i.e. no further passes, analyses, or modification to VHDL code are required), creating full IR for VHDL is not necessary. Therefore, VHDL generation is performed using templates of VHDL code for individual patterns/entities. Since functionality of an entity is fixed (e.g. counter) and only particular instantiations differ according to the context (e.g. size of counter, type of data the entity operates on), VHDL entities can be described in terms of templates with placeholders for context-dependant variations. At code generation time, compiler can then load entity's template, replace placeholders with information provided by the context and emit the final code.

The simplest example of templating VHDL entities is identity entity Id (Listing 5.1). In this example, placeholders are denoted by being enclosed in <>. Id entity performs the same operation in all contexts. However, different VHDL code must be generated based on the type of input data. This is controlled by <type>placeholder that is replaced by compiler with the type of data it is applied to. Note, name of the entity must also be annotated with information about specific instantiation of the template, as otherwise Id entities operating on different data types would result in name conflict.

Listing 5.1: VHDL template for Id entity

```
entity Id_<name> is port(
  clk:   in   std_logic;
  reset:  in   std_logic;

  data_in:   in   <type>;
  valid_in:  in   std_logic;
  ready_in:  out  std_logic;
  last_in:   in  <last_type>;

  data_out:  out <type>;
  valid_out:   out  std_logic;
  ready_out:   in   std_logic;
  last_out:    out  <last_type>
  ); end entity;
architecture behav of Id_<name> is begin
  data_out   <= data_in;
  valid_out <= valid_in;
  ready_in  <= ready_out;
  last_out  <= last_in;
end architecture;
```

5.4 Implementation Issues

5.4.1 Greedy Consuming

Entity can be described as greedy consumer. Greedy consumer consumes input from its producer even through its consumer does not request any data. For example, `Pipe` entity caches incoming data before its consumer asserts `ready` signal.

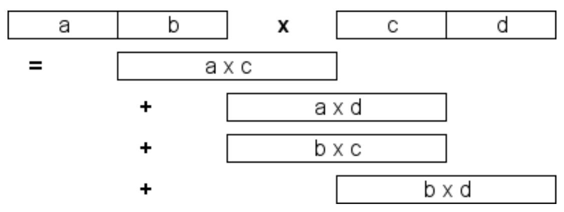
As discussed in Section 4.3.3.5, Let pattern instantiates Buffer to store input data in Block RAM for reuse. An issue arises when Let body contains a greedy consumer. Such a consumer may make Buffer to overflow and start reproducing stored data. If, in the meantime, the body of Let finishes, Buffer is reset, invalidating stored data, while the greedy consumer might hold the old ”pre-consumed” data. Then, if an entity following the greedy consumer requests data, it may receive old invalid data, resulting in incorrect execution and possible deadlock due to de-synchronisation of the stream control signals.

This issue was resolved in the project by restricting the greedy consuming behaviour. In particular, Pipe entity was modified to stop consuming upon observing last signal in the input. The last signal, in this case, indicates Buffer entity has reached the end of stored data. The greedy consumer entity resumes consuming only when its consumer asserts `ready` signal. This solution might result in unnecessary halt when stored data from `Buffer` is split or otherwise segmented before reaching the greedy consumer. It, however, halts in all cases of the `Buffer` reaching the end of stored data, as split segmentation does not erase `Buffer`'s `last` signal but only adds additional `last` signals for new dimensions. Note, the solution assumes the body of `Let` consumes the stored data in full before the `Buffer` is reset.

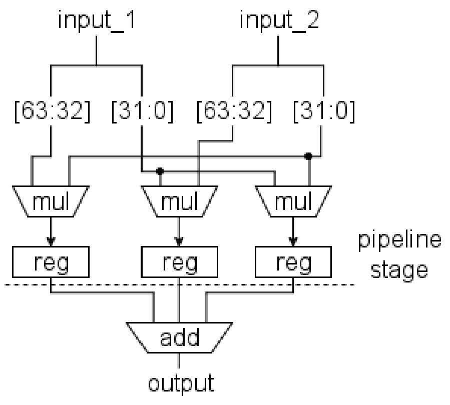
5.4.2 Pipelined Multiplication

Multiplication of 64-bits integers incurs a large delay. For explored designs, direct multiplication of integers could not be achieved for clock frequencies greater than 95MHz. This limit resulted in sub-optimal performance. To resolve this issue, new `UserModule` was developed to perform integer multiplication in two pipeline stages. The logical view of the integer multiplication is depicted in Figure 5.1.

The multiplication can be split into 4 partial multiplications. Then, the partial results are added together to give the final result. Since `UserModule` in question accepted two 64-bit integers and produced 64-bit integers, the upper bit of the multiplication could be discarded, thus `a*c` partial multiplication was not computed at all. The other three



(a) Logical division of integer multiplication



(b) Hardware implementation of pipelined multiplication

Figure 5.1: Pipelining integer multiplication

partial multiplications (i.e. `a*d`, `b*c`, and `b*d`) were computed in the first stage of the pipeline. In one clock cycle time, these partial results can be summed together to produce the final output.

Using pipelined version of integer multiplication produced correct results even for clock frequencies of 210MHz. Since the solution caches partial result the same way `Pipe` pattern caches data, it also exhibits greedy consuming behaviour. The problem of using greedy consuming entity in the body of `Let` pattern was resolved as described in previous section.

Chapter 6

Evaluation

This chapter evaluates the project against reference CPU-base implementation in Numpy. It starts by presenting the target system in which the project was evaluated. Then, the evaluation metrics and methodology is introduced. Finally, the system is evaluated for four applications, each posing different requirements on the system.

6.1 System Setup

The target system used for evaluation of the work of this project was Xilinx's ZYBO board [29]. ZYBO is the smallest board of the Xilinx Zynq-7000 family, thus is not commonly used in the industry for computationally heavy applications but is ideal for teaching and experimentation. The target system can be described in terms of hardware and software. Hardware concerns the specifications of the ZYBO board and design of the FPGA fabric. The terminology refers to the hard IP (i.e. in actual silicon) of the board as Processing System (PS) and to the soft IP (i.e. FPGA fabric) as Programmable Logic (PL). Software concerns OS setup, interfaces to PL (IO devices) and drivers for the IO devices.

6.1.1 Hardware

The board features dual-core Cortex-A9 processor in PS running at clock frequency of 650 MHz. The core has access to 512MB of DDR3 memory with bandwidth of 1050Mbps and 8 DMA (Direct Memory Access) ports. PL comprises of 240 KB Block RAM (i.e. PL memory) and 28,000 logic cells. The architecture of the board is depicted in Figure A.1 in Appendix A.

To be able to pass data between PL and main memory, logic modules based on AXI (Advanced eXtensible Interface) specification were used. AXI is a third generation of AMBA (Advanced Microcontroller Bus Architecture), which is an open-standard for on-chip communication developed by ARM. The target system is configured using a block design that instantiates AXI DMA module and connects it to a high performance

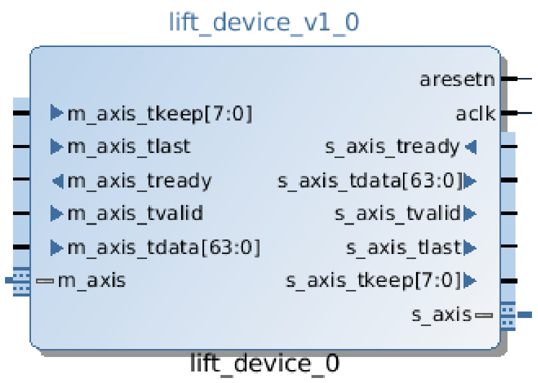


Figure 6.1: Module lift_device with its interfaces. Screenshot of Vivado Design Suite [28].

slave AXI interface on the PS side. Through this interface AXI DMA module can initialise DMA transfer. This module is also connected to master PS interface which allows PS to configure AXI DMA module. A custom module *lift_device* is integrated in the block design. This module implements the design generated from a Lift program. The interface of this module is depicted in Figure 6.1. The whole block design of PL is depicted in Figure A.2.

The described system is fixed for all Lift programs, thus cannot be modified as a result of compilation of a Lift program. Only the design of the lift_device module can be modified by the logic generated by Lift compiler.

6.1.2 Software

In terms of software environment, Linux OS was installed on the board. The particular Linux distribution was acquired from Digilent's GitHub repository [14]. The Linux installation was guided by an online tutorial [2]. Modifications were required to create a custom system for this project and to resolve compatibility issues.

To be able to control the communication of data between main memory and lift_device, AXI DMA module was registered as MMIO (Memory Mapped IO) device in Linux's device tree. Since this device must be given physical address to initialise DMA transfer, memory device was created by splitting the main memory in half (256 MB) and registering the upper address space as another MMIO device. This part of the memory was used as a scratch space where input data, intermediate results, and output data can live. Note, only data placed in the scratch space can be used for DMA transfer. Furthermore, since the scratch memory is interfaced as IO device, the memory accesses ignore the memory hierarchy (i.e. caches).

To be able to communicate with these two devices, python drivers using *mmap* library were created. Using these drivers the address spaces of the scratch memory and configuration registers of AXI DMA device could be accessed. The registration of the two IO devices in Linux and the code for the devices' drivers were guided by an online tutorial [18].

Using the two subdrivers a lift_device driver was developed to manage the scratch space and coordinate data DMA transactions. This driver provides an API to user that wishes to start a Lift program computation. This API is directly generated by Lift compiler as software side code. For the user, other relevant methods of the driver are *malloc*, and *free* methods. The user can use these methods to allocate and deallocate space in the scratch memory. The user can then load data into the scratch memory before starting the Lift program.

6.2 Experimental Methodology

Evaluation of application is performed using two independent variables, namely clock frequency and input data size. The maximal and minimal clock frequencies the PL system could be configured to were 210 MHz and 50MHz, respectively. The input data size was limited by allowed size of DMA transfer. The maximal DMA transfer could span 4MB of consecutive memory. The minimal transfer was limited by the interface to the lift_device, i.e. 8 bytes. For all applications the basic data type used was 64-bit integer.

For any combination of the independent variables the application being evaluated was first tested for correctness of the result. Even though produced hardware design could be semantically correct by construction, the produced result could be invalid due to clock frequency being too high. Therefore, it must be assured hardware can produce valid result for all operations on data within given clock period.

Applications were evaluated by measuring execution time. The time was measured across the call to the driver's API specific to the application (i.e. `submit`). Before measuring time, input data was initialised to random values. Performing one evaluation for the working example application introduced in Chapter 4 is depicted in Listing 6.1. Five measurements of execution time were taken for each combination of independent variables. Only average execution time is reported. Standard deviation of the measurements was less than 5% of the measured value for all experiments, which is negligible compared to differences in measured values for FPGA-based and numpy-based versions of applications. Such a low deviation was expected, as FPGA is a stand-alone device not sharing resources other than the main memory bus with other actors. The evaluation system itself runs light-weight Linux with no other applications running. Therefore, noise in the system is minimal.

Listing 6.1: Measuring execution time for working example application

```
import timeit
device = LiftDevice()
a = device.malloc(1024, np.int64)
b = device.malloc(1024, np.int64)
a[:] = np.random.randint(255, size=a.shape)
b[:] = np.random.randint(255, size=b.shape)
execution_time = timeit.timeit('submit(device,a,b)', number=1)
```

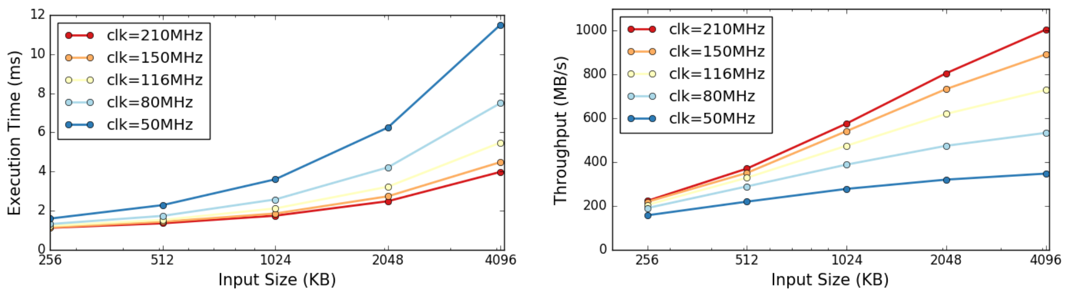
6.3 Applications

This section presents evaluation of the system for several applications. Most applications are compared to native `numpy` implementation of the applications. The baseline application explores theoretical boundaries on computational power and overheads involved in using FPGA-based heterogeneous system. The rest of the applications evaluate the system by imposing different requirements.

6.3.1 Data Movement

This application consists of a simple identity operation resulting in copying the data from one memory location to another with passing the data through FPGA in-between. This corresponds to the communication bottleneck that represents a theoretical upper bound on computational performance of the FPGA device. Note, two-way communication is measured. One-way communication might achieve slightly higher throughput than the values reported in this section due to exclusive access of the communication channel to the interconnect bus.

The execution time of identity application is measured for input data sizes of 4096, 2048, 1024, 512, and 256 KB. Clock frequencies were varied from 210 MHz to 50 MHz. Execution times and throughputs for different clock frequencies per input sizes are depicted in Figure 6.2.



(a) Execution times per data size

(b) Throughput per data size

Figure 6.2: Evaluation of identity operation

From the data collected, execution time of the data transfer linearly scales with the size of the data being transferred. Looking at the smallest input data size, the transfer time includes some constant overhead. Using data size of 8 bytes, the overhead was measured to be 0.877ms. This overhead was within 0.5% of the reported value for all clock frequencies.

Clock frequency is negatively correlated with transfer time, i.e. for larger clock frequencies smaller execution times were measured. This observation was expected, as the streaming protocol used in the data-flow is limited in speed to one data element per clock cycle.

Due to the constant overhead, greater throughputs are observed for larger data sizes, as the constant overhead makes up smaller proportion of the transfer time for large data sizes. In agreement with execution time, throughput is positively correlated with clock frequency.

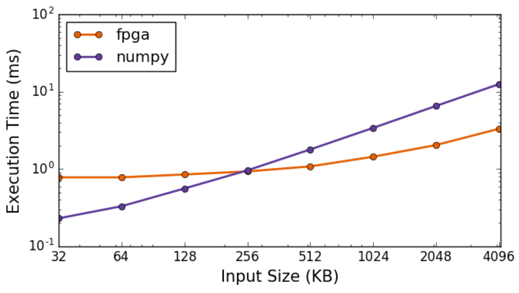
6.3.2 Sum

This application corresponds to summing all elements of an input array. It can be expressed using Lift low-level patterns as depicted in Listing 6.2. The listing also depicts numpy alternative against which the fpga implementation is evaluated. Experiments were performed only for the highest allowed clock frequency of 210 MHz, as this frequency results in the highest throughput of data communication (see Section 6.3.1).

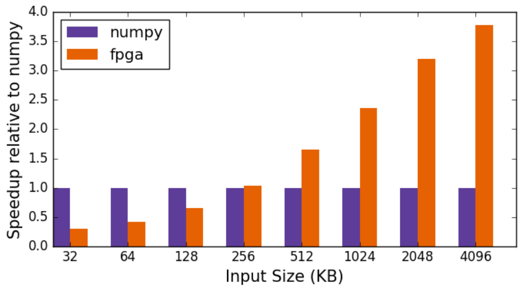
Listing 6.2: Sum of array elements in Lift and numpy

```
# Lift
add = (x => UserModule.Addition() $ x)
program = (x => ToHost() o ReduceStream(add, 0) o ToFPGA() $ x)
# NumPy
np.sum(x)
```

Data collected with several input sizes are depicted in Figure 6.3. FPGA implementation of this application is compared to numpy implementation.



(a) Execution times per data size



(b) Speedup compared to numpy per data size

Figure 6.3: Evaluation of sum operation

From the data collected, FPGA implementation of the application results in better execution time than numpy implementation for inputs of size at least 256KB. This can be explained by the constant overhead involved in communication of data between the main memory and FPGA. Note, this overhead is not present (or is negligible) in numpy's on-CPU computation. Since the overhead is constant, it is relatively less significant for larger data sizes. This explanation is further supported by FPGA's speedups compared to numpy being larger for larger input sizes. The equivalence point where FPGA and numpy runtimes are comparable can be found at input size of 256KB.

6.3.3 Matrix-Vector multiplication

This application corresponds to multiplication of matrix and vector. It can be expressed in Lift low-level patterns as depicted in Listing 6.3. Numpy alternative used for execution time comparison is also depicted. FPGA-based implementation first sends the input vector to FPGA and stores it in Block RAM using Let pattern. Note, Let pattern allows consuming the input vector multiple times. Then, the input matrix is flattened to match the input type of ToFPGA pattern, which sends the matrix to FPGA. On FPGA, the matrix is split into rows and dot product with the input vector is computed for each row.

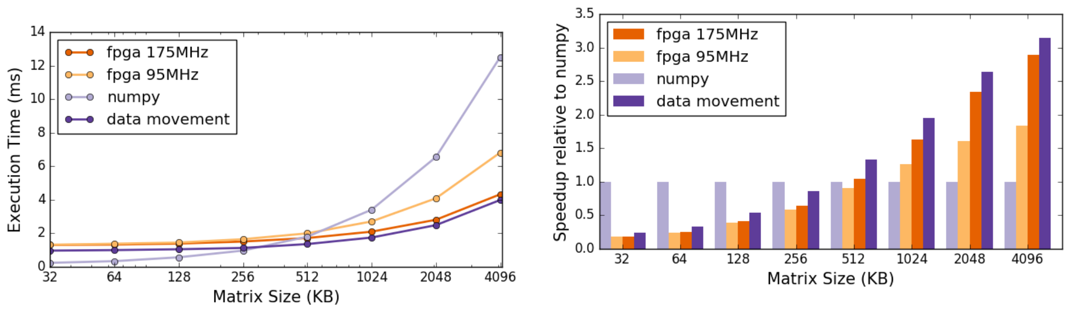
Listing 6.3: Matrix-vector multiplication in Lift and numpy

```
# Lift
add = (x => UserModule.Addition() $ x)
mul = (x => UserModule.Multiplication() $ x)
dot = ((x,y) => ReduceStream(add, 0) o MapStream(mul) o Zip(x,y))
program = ((M, v) => ToHost() o Let( v_cache =>
  MapStream(M_row => dot(M_row, v_cache)) o
  SplitStream(M_num_col) o ToFPGA() o Flatten() $ M
) o ToFPGA() $ v)
# NumPy
np.dot(M,v)
```

In terms of independent variable, size of both inputs can be varied. Since the input vector is stored on FPGA, its size is limited by the on-FPGA memory available. The upper bound is 240 KB. On the other hand, size of the input matrix is only limited by the maximal size of DMA transfer, which is 4MB. Since the vector size is negligible compared to the matrix size, matrix size is the main determinant of the execution runtime. For experiments with this application, vector size was fixed to 8KB and the matrix size was varied from 4MB to 32KB along logarithmic scale.

Since the application involves multiplication of 64-bit integers, which is a heavy operation, the hardware design did not manage to produce correct result when using clock frequency of 210 MHz. When using a single cycle multiplication, the highest clock frequency producing correct results was 95MHz. Changing the multiplication module to pipelined version introduced in Section 5.1 allowed using clock frequency of 175MHz. Experiment results for both versions of multiplication module are reported. Furthermore, runtimes for data movement application for the same input size and clock frequency of 175MHz are also reported. Data collected with several input matrix sizes are depicted in Figure 6.4.

Collected data suggests similar speedups of FPGA implementation against numpy implementation as for sum application. The best speedups are observed for large input sizes, when the constant overhead in FPGA implementation is amortised over the large input. The equivalence point were FPGA and numpy implementation runtimes are comparable is found at matrix size of 512KB. This is in contrast with Sum application, where it was 256KB. The equivalence point shift can be explained by this application requiring to send two pieces of data to FPGA. Thus, one more DMA transfer is required compared to Sum application, resulting in greater constant overhead for FPGA implementation.



(a) Execution times per matrix size

(b) Speedup compared to numpy per matrix size

Figure 6.4: Evaluation of matrix vector multiplication with vector size of 8KB

The greater constant overhead of this application is also apparent from comparison of FPGA implementation to the data movement application. Since the data movement only indicates execution time of communication of the input data as a single DMA request, it incurs smaller constant overhead than this application, which moves data as two DMA requests. Comparison to the data movement application suggests FPGA implementation could further improve the execution time by sending the vector together with the matrix in a single DMA request. This would, however, require some logic in FPGA to split the continuous stream into vector and matrix. At the moment, the Lift patterns are not expressive enough to cover this execution path.

From the data, performance difference between FPGA versions using non-pipelined and pipelined multiplication is apparent. Pipelined multiplication version performs better than non-pipelined version for all input sizes. The difference is, however, greater for larger input sizes, which mimics the throughputs observed in data movement application.

6.3.4 Matrix-Matrix multiplication

This application corresponds to matrix multiplication. It can be expressed in Lift low-level patterns as depicted in Listing 6.4. Numpy equivalent used for evaluation is also depicted. For this application, generated host code and block diagram of generated hardware design are included in Appendix A.3. FPGA implementation first transposes the second input matrix B so that it can be accessed by columns. Then, B is sent to FPGA to be stored in BRAM using ToFPGA pattern followed by Let pattern. Then, the first matrix A is sent to FPGA. On FPGA, flattened A is split into rows and function of the outer MapStream pattern is applied to each row. In the body of this function matrix B is split into columns and dot product with the current row of A is applied to all columns of B.

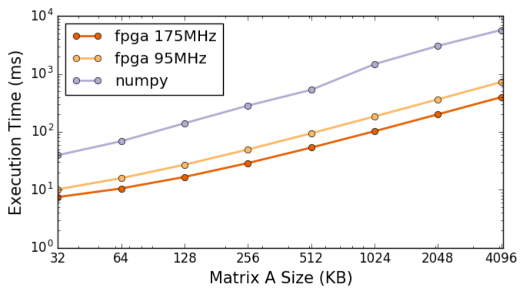
Listing 6.4: Matrix multiplication in Lift and numpy

```
# Lift
add = (x => UserModule.Addition() $ x)
mul = (x => UserModule.Multiplication() $ x)
dot = ((x,y) => ReduceStream(add, 0) o MapStream(mul) o ZipStream(x,y))
```

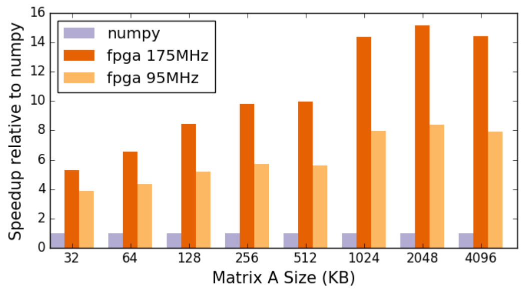
```
program = ((A, B) => Split(B_num_col) o ToHost() o
  Let(B =>
    JoinStream() o
    MapStream(A_row =>
      Let(A_row =>
        JoinStream() o
        MapStream(B_col => dot(A_row, B_col)) o
        SplitStream(B_num_row) $ B
      ) $ A_row
    )
    o SplitStream(A_num_col) o ToFPGA() o Flatten() $ A
  ) o ToFPGA() o Flatten() o Transpose() $ B
)

# NumPy
np.dot(A,B)
```

In terms of independent variables, sizes of matrices are limited from above by the amount of on-FPGA memory available. This memory for this application must provide enough capacity to hold the whole matrix B and one row of matrix A. Since the total on-FPGA memory is 240KB, matrix B was chosen to be a square matrix of size 128KB. This results in 128*128 dimensional matrix of 64-bit integers. Given the shape of matrix B, matrix A was required to have 128 columns in each row, which fits the available on-FPGA memory. The other dimension of matrix A was varied to result in several total sizes of matrix A, resulting in 8 values of total size of matrix A, from 4MB to 32KB on logarithmic scale. To ensure the application produces valid output for all sizes of input data, clock frequency of 175MHz was used for pipelined multiplication version and 95MHz for non-pipelined version. Data collected with several total sizes of matrix A are reported in Figure 6.5.



(a) Execution times per matrix A size



(b) Speedup compared to numpy per matrix A size

Figure 6.5: Evaluation of matrix multiplication with matrix B of size 128KB

FPGA implementation exhibits significant speedups over numpy implementation for all sizes of matrix A. Similarly to previous applications, this speedup is greater for greater input sizes. This again can be explained by mitigating the constant overhead of FPGA implementation over large inputs. Furthermore, CPU-based implementation might incur additional penalties for large inputs due to cache invalidation issues. These penalties are not present in FPGA implementation as no data that is used in the future is ever invalidated during execution of FPGA implementation. Since no data is reused in any of the previous applications, the cache invalidation penalties were not present in the previous applications either.

This application is not compared against the data movement application. This is due to dramatically smaller data movement runtime compared to measured FPGA runtime. If reported, comparison would overshadow the comparison with numpy implementation. For example, runtime of this application for the largest input size was 100X greater than the data movement runtime only. The big gap between theoretical boundary given by the data movement and actually measured runtime of this application suggests possible improvements of the FPGA implementation. Consumption of matrix B is comparable to the baseline, as the matrix is stored in BRAM at the rate of one element per cycle. Consumption of matrix A is the factor that slows down the application compared to the baseline. Since for each row of matrix A the whole matrix B must be traversed, improving the access throughput to matrix B in BRAM would result in faster execution of the whole application. Alternatively, if both matrices could be stored on FPGA (e.g. using larger FPGA board), the matrices could be loaded and the result could be computed in parallel in just a few clock cycles. All these improvements can be the subject of future work.

Similarly to matrix vector multiplication, using pipelined multiplication version result in significant improvement of runtime compared to non-pipelined version. However, for this application the performance difference is even more apparent. This may be explained by the constant overhead skewing the relative improvement of runtime for matrix vector multiplication. Since this application results in runtime much greater than the data movement, the constant overhead is relatively small, thus the pipelined version speedup is more pronounced.

6.4 Summary

This chapter reported results of evaluation of the system produced in this project on four applications. The first application represented theoretical upper bound on computation speed given by data communication throughput. From other applications, it was apparent the produced system is capable of outperforming reference CPU-based implementations of the applications in standard framework for scientific computing. However, achieved speedups are only observed for large enough inputs. For smaller inputs, constant overhead involved in FPGA-based implementation results in worse execution time than CPU-based implementation.

Chapter 7

Conclusion

This dissertation demonstrated how to generate hardware design from functional description of an application. By integration with a high-level functional compiler we are one step closer to providing programmability of hardware through high-level platform-agnostic front-end.

First, chapter 4 presented the integration with Lift compiler be addition of new FPGA-specific low-level patterns. The patterns are described in terms of semantics and hardware design generated from using the patterns is presented. Then, chapter 5 explains more details of how hardware generation was implemented in Lift compiler. It also presents issues encountered during development and how they were resolved. Finally, chapter 6 evaluates the created Lift back-end on several applications, comparing runtimes of Lift generated hardware with CPU based reference implementation of the same application in Numpy framework.

7.1 Critical Evaluation

This project is deemed successful. Functional patterns introduced to Lift compiler compose well at functional level as well as at hardware design level. As demonstrated by several applications, changing one pattern (e.g. `UserModule` from non-pipelined to pipelined multiplication) might impact the resultant performance while preserving the validity of modules composition. The new framework also gives good results compared to reference numpy implementation for several application. The greatest speedups are achieved for largest input data sizes, which is explained by a large constant overhead involved in data communication within heterogeneous environment. However, comparison of runtimes of applications based on data reuse (e.g. matrix multiplication) to the runtime of data movement only for the same input size shows unfulfilled potential of the FPGA-based implementation. This gives a good base for future work that could design optimisations targeting particular access patterns and/or implementation of patterns at either host or FPGA side.

The main drawback of the project is the limitations of `ReduceStream` pattern. The reduction function is limited to combinatorial transformation which prohibits usage of computationally heavy operations and by extension usage of advanced data types such as floats. Furthermore, accumulator of the reduction must be updated in one clock cycle, i.e. the new accumulator value must be available in full in one clock cycle. This limitation prohibits reduction over arrays (i.e. where the accumulator is of type array) which would be useful, for example, for tiled version of matrix multiplication. Resolving this problem is particularly difficult due to the semantics of `ReduceStream` pattern. The pattern instantiates arbitrary module to perform some data transformation. Since the module's input is dependant on its output, more synchronisation is required to lift the described limitations. It is also hard to reason about providing valid input to the arbitrary module in case of only partially updated accumulator.

During the project, last signal in streaming communication (and data flow control) protocol was extended into vector to represent multi-dimensional streams. This extension, however, is not utilised by any of the patterns used in the applications experimented with. For example, `MapStream` pattern instantiates an arbitrary module to perform transformation on elements of input stream, where each element can also be a stream. However, it cannot propagate the last signal corresponding to the higher dimension directly to the output due to the possibility of the arbitrary module introducing a clock delay between input and output. Therefore, `MapStream` pattern must count the outputs of the arbitrary module to reintroduce the last signal for higher dimension, rendering multidimensional last vector useless as only the low-dimensional last signal is used. It in unclear whether any future patterns and/or application would utilise the vector representation of the last signal. Reverting to a scalar last signal would result in a unified control signals that are datatype agnostic. Then, multi-dimensional streams would be only implicit in the generated hardware. Since the compiler would retain the type information, it could generate entities relying on dimensionality information that would use counters for detect end of multi-dimensional input streams for particular dimensions. Perhaps, more advanced compiler analysis could decide which version of last signal to use on individual basis. However, this would result in inconsistent representation of datatype information in generated hardware.

7.2 Future work

The next step of the project can focus on various aspects. For example, it can add more patterns to the Lift compiler to increase the range of applications that can be expressed or improve already existing patterns by designing optimisations. Alternatively, it can be explored how to utilise this project in the context of other applications.

7.2.1 New Functional Patterns

The project is currently still far from achieving the ultimate goal of providing high-level platform-agnostic framework for general programming. To get closer to this goal, new low-level patterns corresponding to high-level patterns already present in

Lift should be added to the project. This might include host side `Map`, `Reduce`, `Zip`, and a way for user to specify custom host side transformation (i.e. host side equivalent of `UserModule`). Furthermore, the project can be extended with more `UserModule` to provide a library implementations of common transformations (add, multiply) operating on common data types (int, float, double).

7.2.2 Optimisations

As hinted in Sections 6.3.4 and 7.1, the project might benefit from optimising implementation of already existing patterns. The range of possible optimisation spans host side as well as FPGA side of the system.

The main opportunity for host-side optimisation is to utilise memory hierarchy available to CPU. Currently, all data operated on lives in scratch memory, which is registered as IO device, thus all accesses bypass cache hierarchy and go directly to the main memory. This slows down all host side transformations while only data to be send or received to/from FPGA need to live in the scratch space. Compiler analysis could find temporary results that are not directly used for communication with FPGA, and thus could be allocated in virtual address space of the running process resulting in faster access due to cache hierarchy utilisation.

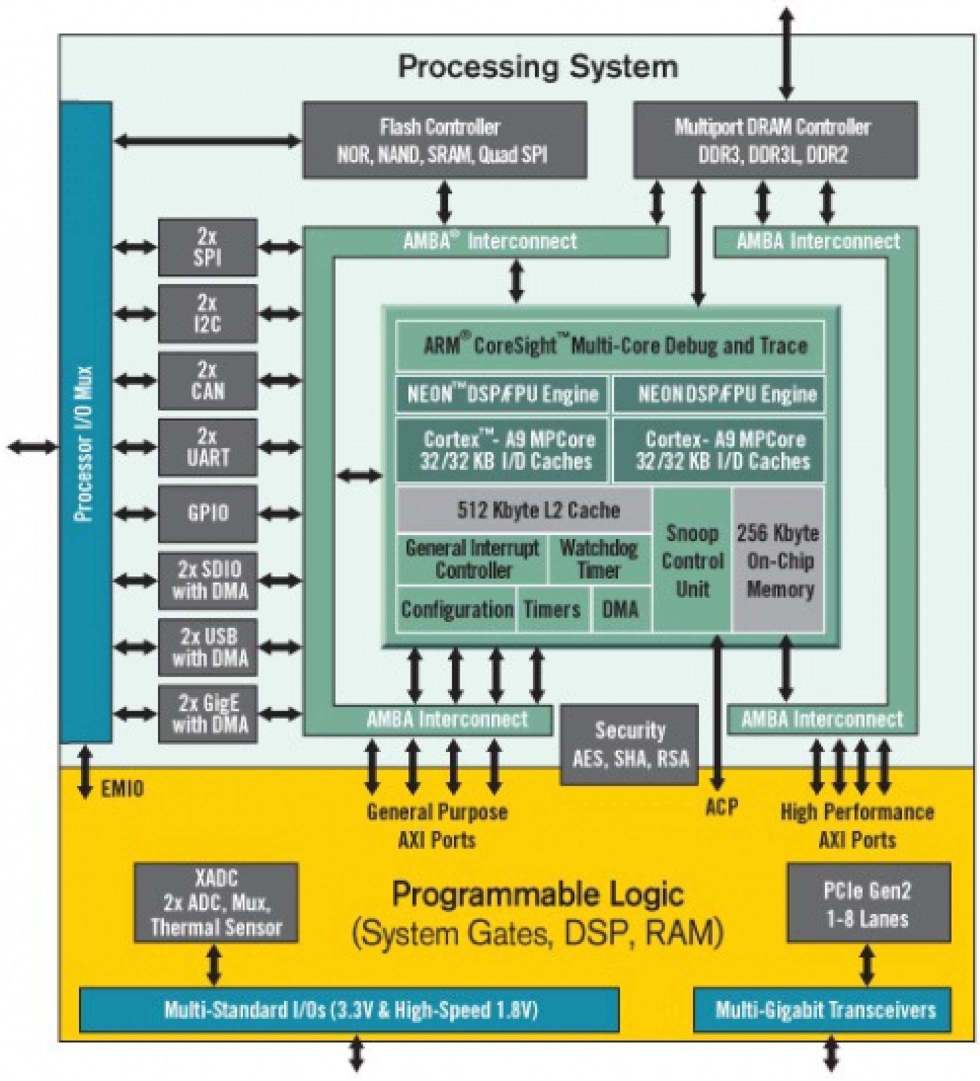
Based on results obtained in Section 6.3.4, data access can be the bottleneck of applications relying on data reuse. This finding suggests optimisations to `Buffer` module could improve the overall performance of the system. For example, accessing stored data could be parallelised so that several data items are available each clock cycle. Another optimisation might be forwarding of stored data. Currently, `Buffer` module starts producing data only after all incoming data is stored. Forwarding would allow producing partial data that is already stored before the whole data sequence is stored.

7.3 New Applications

The project should be evaluated for a wider range of applications. One of the advantages of FPGAs compared to GPUs and CPUs is that precision of data transformations can be customised in terms of data bit-width. This, in particular, can be useful for applications, where precision is not a significant requirement, such as neural networks. For these applications, this project could operate with 8-bit precision as opposed to minimal of 32-bit precision used in CPUs or GPUs. Furthermore, floating point data could be converted to fixed point or even integers, resulting in potentially much faster execution compared to GPUs.

Appendix A

Figures



Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
Processor I/O Mux
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
AMBA® Interconnect
AMBA Interconnect
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Figure A.1: ZYBO Soc architecture

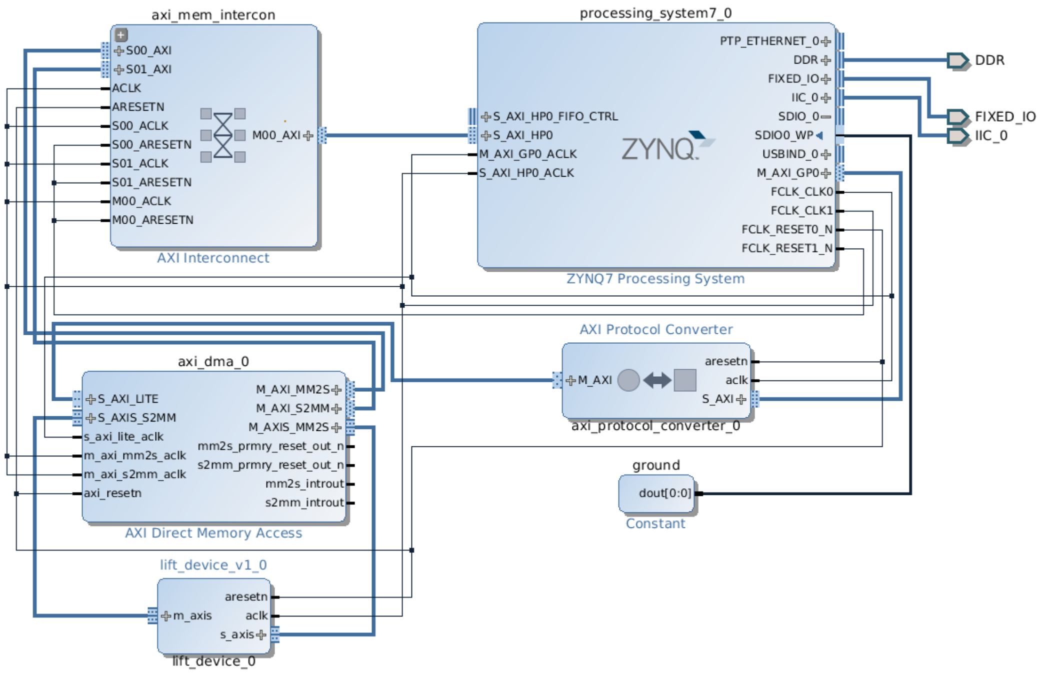
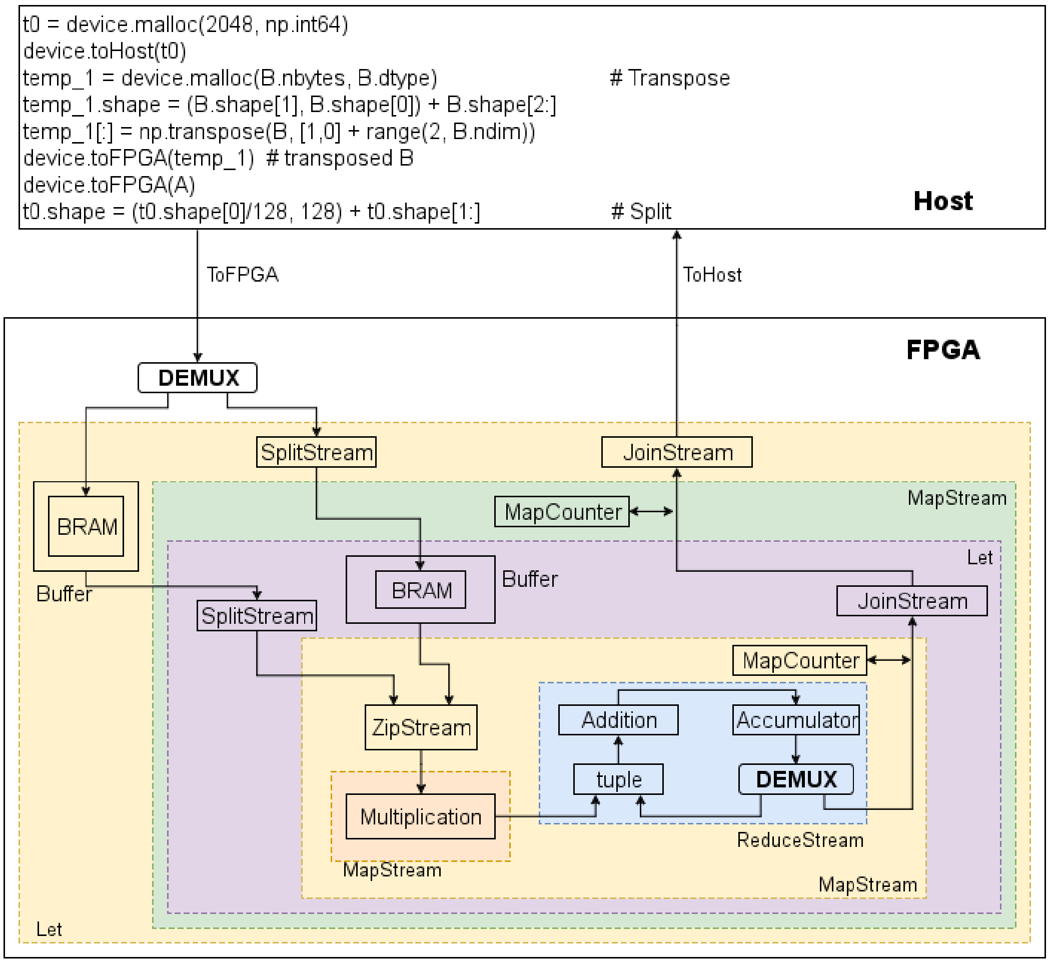


Figure A.2: Block design of the whole target system. Screenshot from Vivado [28]



```
t0 = device.malloc(2048, np.int64)
device.toHost(t0)
temp_1 = device.malloc(B.nbytes, B.dtype)              # Transpose
temp_1.shape = (B.shape[1], B.shape[0]) + B.shape[2:]
temp_1[:] = np.transpose(B, [1,0] + range(2, B.ndim))
device.toFPGA(temp_1)  # transposed B
device.toFPGA(A)
t0.shape = (t0.shape[0]/128, 128) + t0.shape[1:]       # Split
```

Figure A.3: Host code and FPGA block design for matrix multiplication. Host code specific to (2,128) * (128,128) matrix multiplication. In host code, Flatten pattern and type checking is omitted for verbosity issues.

Bibliography

[1] Lift: A functional data-parallel ir for high-performance gpu code generation.

[2] Booting Linux on the ZYBO. `http://www.dbrss.org/zybo/tutorial4.html`, 2016.

[3] Accellera. Systemverilog 3.0 accelleras extensions to verilog. 2002.

[4] Joshua Auerbach, David F Bacon, Ioana Burcea, Perry Cheng, Stephen J Fink, Rodric Rabbah, and Sunil Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference*, pages 271–276. ACM, 2012.

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225. ACM, 2012.

[6] David F Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.

[7] Brian Bailey, Felice Balarin, and Michael McNamara. *Tlm-driven design and verification methodology*. Lulu. com, 2010.

[8] David Bishop. Fixed point package users guide. *Packages and bodies for the IEEE*, pages 1076–2008, 2010.

[9] Thomas Bollaert. Catapult synthesis: a practical introduction to interactive c synthesis. In *High-Level Synthesis*, pages 29–52. Springer, 2008.

[10] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

[11] Wikimedia Commons. Field-programmable gate array, 2017. File: FPGA_cell_example.png.

[12] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[13] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From opencl to high-performance hardware on fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534. IEEE, 2012.

[14] Inc. Digilent. linux-Digilent-Dev. `https://github.com/Digilent/linux-Digilent-Dev`, 2016.

[15] Stephen A Edwards. High-level synthesis from the synchronous language esterel. In *IWLS*, pages 401–406. Citeseer, 2002.

[16] David Greaves and Satnam Singh. Designing application specific circuits with concurrent c# programs. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 21–30. IEEE, 2010.

[17] Pekka O Jäskeläinen, S Carlos, Pablo Huerta, and Jarmo H Takala. Opencl-based design methodology for application-specific processors. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 223–230. IEEE, 2010.

[18] MiguelR73. Zybo - AXI DMA inside embedded Linux. `http://www.instructables.com/id/Zybo-AXI-DMA-Inside-Embedded-Linux/`, 2016.

[19] Syed Waqar Nabi and Wim Vanderbauwhede. Using type transformations to generate program variants for fpga design space exploration. In *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*, pages 1–6. IEEE, 2015.

[20] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70. IEEE, 2004.

[21] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D Antonopoulos. Synthesis of platform architectures from opencl programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193. IEEE, 2011.

[22] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D Antonopoulos. Synthesis of platform architectures from opencl programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193. IEEE, 2011.

[23] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on*, pages 35–42. IEEE, 2009.

[24] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level func-

tional expressions to high-performance opencl code. *ACM SIGPLAN Notices*, 50(9):205–217, 2015.

[25] C Synphony. Compiler, Synopsys. *Inc*, 2014.

[26] Miljan VuletiĆ, Christophe Dubach, Laura Pozzi, and Paolo Ienne. Enabling unrestricted automated synthesis of portable hardware accelerators for virtual machines. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 243–248. ACM, 2005.

[27] Rick Weber, Akila Gothandaraman, Robert J Hinde, and Gregory D Peterson. Comparing hardware accelerators in scientific applications: A case study. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):58–68, 2011.

[28] Xilinx. Vivado Design Suite - HLx Editions. `https://www.xilinx.com/products/design-tools/vivado.html`. [Online; accessed 5-April-2017].

[29] Xilinx. Zybo Zynq-7000 ARM/FPGA SoC Trainer Board. `https://www.xilinx.com/products/boards-and-kits/1-4azfte.html`. [Online; accessed 26-January-2017].

[30] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. Autopilot: A platform-based esl synthesis system. In *High-Level Synthesis*, pages 99–112. Springer, 2008.