This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.
Abstract

Specialised accelerators deliver orders of magnitude higher energy-efficiency than general-purpose processors. Field Programmable Gate Arrays (FPGAs) have become the substrate of choice, because the ever-changing nature of modern workloads, such as machine learning, demands reconfigurability. However, they are notoriously hard to program directly using Hardware Description Languages (HDLs). Traditional High-Level Synthesis (HLS) tools improve productivity, but come with their own problems. They often produce sub-optimal designs and programmers are still required to write hardware-specific code, thus development cycles remain long.

This thesis proposes Shīr, a higher-level synthesis approach for high-performance accelerator design with a hardware-agnostic programming entry point, a multi-level Intermediate Representation (IR), a compiler and rewrite rules for optimisation.

First, a novel, multi-level functional IR structure for accelerator design is described. The IRs operate on different levels of abstraction, cleanly separating different hardware concerns. They enable the expression of different forms of parallelism and standard memory features, such as asynchronous off-chip memories or synchronous on-chip buffers, as well as arbitration of such shared resources. Exposing these features at the IR level is essential for achieving high performance.

Next, mechanical lowering procedures are introduced to automatically compile a program specification through Shīr’s functional IRs until low-level HDL code for FPGA synthesis is emitted. Each lowering step gradually adds implementation details.

Finally, this thesis presents rewrite rules for automatic optimisations around parallelisation, buffering and data reshaping. Reshaping operations pose a challenge to functional approaches in particular. They introduce overheads that compromise performance or even prevent the generation of synthesisable hardware designs altogether. This fundamental issue is solved by the application of rewrite rules.

The viability of this approach is demonstrated by running matrix multiplication and 2D convolution on an Intel Arria 10 FPGA. A limited design space exploration is conducted, confirming the ability of the IR to exploit various hardware features. Using rewrite rules for optimisation, it is possible to generate high-performance designs that are competitive with highly tuned OpenCL implementations and that outperform hardware-agnostic OpenCL code. The performance impact of the optimisations is further evaluated showing that they are essential to achieving high performance, and in many cases also necessary to produce hardware that fits the resource constraints.
Lay Summary

Many day-to-day computer and mobile apps, such as search engines and speech recognition, use so-called artificial intelligence to get the job done. At the same time, this technology is computationally intensive, which results in enormous power consumption, especially in data centres.

For this reason, data centres employ special devices, so-called FPGAs, which accelerate the same computations and perform them much more efficiently. However, FPGAs are many times more complex and time-consuming to program than conventional computers. Application developers need considerable expertise to write the detailed program code for FPGAs. This makes the development costs for such applications with artificial intelligence climb steeply.

This thesis addresses this problem by enabling developers to use a simple, abstract programming language for programming FPGAs. Only a few lines of code are then needed to describe the desired computation. However, the program code now lacks the many details that are still required for the FPGA to understand the instructions and execute them efficiently. In order to solve this, an automatic translation tool, called compiler, is provided by this thesis. It translates the abstract program code given by the developer step by step into the more detailed programming language intelligible to FPGAs. The compiler automatically determines the details so that the FPGA works as efficiently as possible.

In summary, this thesis shows how, starting from an abstract programming language, a compiler can be used to automatically generate efficient code for FPGAs.
Acknowledgements

This thesis marks the end of an intense four-year research period, which was packed with incredible experiences that will always be remembered. It was a great pleasure to meet and become friends with many nice people on this eventful journey. My research efforts were supported by countless individuals to whom I would like to express my deepest gratitude.

First and foremost, I want to thank my supervisor, Prof. Christophe Dubach, for the immense inspiration, tireless motivation and guidance. Many thanks for taking the time to extensively discuss ideas and share invaluable advice that will benefit me beyond my PhD.

When I started my PhD, I was fortunate to join the Lift team, a group of ambitious and motivated researchers: Michel Steuwer, Larisa Stoltzfus, Federico Pizzuti, Naums Mogers, Thomas Kœhler, Andrej Ivanis, Bastian Hagedorn, Bastian Köpcke, Lu Li, Martin Kristien, Toomas Remmelg and Adam Harries. I am grateful for the opportunity to have been part of this strong research group. Special thanks to Andrej for helping with the tedious setup of the FPGA server and Naums for providing the arithmetic library used in Shir. In addition, thanks to Larisa for the challenging climbing sessions and Thomas for the energetic table tennis matches.

I am proud that my work lays the foundation for many ongoing research directions of the new Shir team: Tzung-Han Juang, Ayan Chakraborty, Hamza Javed, Shakiba Bolbolian Khah, Jonathan Van der Cruysse, Zhitaol Lin, Ningchen Ma, Archisman Dey and Jacob Peng. It is a great pleasure to see their contributions to the Shir framework. A special thank you to Tzung-Han for the 2D convolution benchmarks and Ayan for implementing the onboard memory interface.

I would like to extend my thanks to all the people who make the School of Informatics at the University of Edinburgh a place where I feel welcome and supported. In particular, I am thankful for those I shared office 1.05 and social lunches with and who ensured a friendly work environment: Rui Li, Chris Vasiladiotis, Jörg Thalheim, Maurice Bailleu, Dimitra Giantsidi, Alexandr Maramzin and Jackson Woodruff.

Last but not least, I am sincerely grateful for my family and friends for supporting me in many different ways. Thanks to Dave and Sara, who made flat 45 feel like home very quickly. Many special thanks to my parents for their encouragement to pursue my dreams and that I could always rely on them. Finally, I thank Uta for her unconditional support, comprehension and motivation.
Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Parts of the material in this dissertation have been published in the following publications:

- **Christof Schlaak**, Tzung-Han Juang, and Christophe Dubach. [SJD22a]

- **Christof Schlaak**, Tzung-Han Juang, and Christophe Dubach. [SJD22b]

During the work on this thesis, I also co-authored the following paper:

- Tzung-Han Juang, **Christof Schlaak**, and Christophe Dubach. [JSD23]

*(Christof Schlaak)*
# Contents

1 Introduction ........................................ 1  
1.1 Challenges .................................................. 4  
1.2 Contributions ............................................... 7  
1.3 Structure ...................................................... 9  

I State of the Art ........................................... 11  
2 Hardware Design and Functional Languages ............. 13  
2.1 Digital Hardware Design With FPGAs .................. 13  
2.2 Functional Programming .................................. 21  
2.3 Term Rewriting Systems .................................. 24  

3 Related Work ............................................... 27  
3.1 Monolithic Template-Based Accelerators .............. 27  
3.2 Hardware Description Languages ...................... 30  
3.3 High-Level Synthesis ..................................... 35  
3.4 Multi-Level Representations .............................. 37  
3.5 Domain-Specific Synthesis ............................... 39  
3.6 Higher-Level Synthesis .................................... 40  

II Contributions ............................................. 49  
4 Multi-Level Representations ............................... 51  
4.1 Overview .................................................... 52  
4.2 Core Abstract Language .................................. 55  
4.3 Algorithmic Level ......................................... 59  
4.4 Architecture Level ........................................ 62
8.2 Critical Analysis ................................................. 146
8.3 Future Work ......................................................... 149

A VHDL Templates ......................................................... 153
   A.1 Overview ......................................................... 153
   A.2 Examples ......................................................... 155

Bibliography ................................................................. 163
Chapter 1
Introduction

Machine learning is a hot topic. Although the ideas of Artificial Intelligence (AI) root back to the 1950s [Tur50], machine learning has only more recently become practical due to improvements in computational performance. It is now ubiquitous in a wide range of application domains in our everyday lives. Computers can process speech and understand natural languages. Two decades ago, they started using computer vision to recognise faces [AM03]. Self-driving cars rely on AI technologies to navigate autonomously [DMDV20]. More recently, AI performs medical diagnosis by detecting tumours [LGN+17]. In 2022, an art competition was won by an artist who generated his hyper-realistic graphics using AI [Roo22].

By helping with pattern recognition and processing copious amounts of data, AI technologies already have a huge societal impact and it keeps rising. This trend is indicated by the growing proportion of venture capital invested in AI related start-ups over the past nine years [Tri21]. However, as deployment and complexity of AI use-cases grow, so does the demand for high computational performance and energy efficiency.

Unfortunately, the Dennard scaling [DGY+74] already ended in the mid-2000s and the end of Moore’s law [Moo06a, Moo06b] is approaching. The exponential performance gains due to shrinking physical dimensions and increasing transistor count in Integrated Circuits (ICs) are stagnating. This has led to the ‘multi-core era’, in which more and more cores are added to each processor for greater performance. However, even multi-core scaling will soon be limited due to the failure of Dennard scaling [EBA+11]. Machine learning is just one of many domains where novel ways to improve computing systems efficiency are required. To exploit the full potential of workloads from such domains, their parallel nature must be taken into account.
Chapter 1. Introduction

(a) Programmability and efficiency of different hardware platforms. FPGAs can be reconfigured to quickly deploy brand new algorithms.

(b) Exponential growth of annually published machine learning papers\(^1\). This number doubles every two years, which is similar to the trend in Moore’s law [Moo06b]. Graph inspired by [DPY18].

Figure 1.1: Despite being less efficient than ASICs, FPGAs are advantageous due to their reconfigurability, which is important to keep up with the ongoing trend of new machine learning technologies.

Hardware Accelerators

Graphics Processing Units (GPUs) shine with their massive parallel processing due to a high number of cores compared to Central Processing Units (CPUs). This allows them to achieve high performance while remaining efficient. Although originally designed to accelerate graphics, GPUs were quickly programmed to carry out general-purpose computing on them, known as General-Purpose Computing on Graphics Processing Units (GPGPU).

However, the efficiency of both CPUs and GPUs is still impeded by their general-purpose architecture and the problem of the memory wall [WM95], also called the von Neumann bottleneck [MT09]. In the von Neumann model, memory is physically separated from the processing unit, causing many data movements that cost a lot of energy. Accessing off-chip memory in a 45 nm technology requires \( \approx 1000 \times \) more power than a 32-bit floating point multiplication. Even when exploiting data locality with caches, this problem persists, since caches still take \( \approx 100 \times \) more power [Hor14].

\(^1\)Accumulated number of papers in the topics cs.LG (Machine Learning), cs.AI (Artificial Intelligence), cs.NE (Neural and Evolutionary Computing) and stat.ML (Machine Learning) from arxiv.org.
As depicted in Figure 1.1a, there are further candidates for accelerating computationally intensive algorithms. FPGAs and Application-Specific Integrated Circuits (ASICs) are special-purpose hardware platforms, tailored and fine-tuned to a workload from a specific domain. With their flexible dataflow architecture designs and distributed memory on the chip, they solve the von Neumann bottleneck. This allows them to achieve high energy efficiency and even beat GPUs [NVS+17].

Google’s Tensor Processing Unit (TPU) [JBB+17] is a prominent example for an ASIC accelerating machine learning applications. This coprocessor with a matrix multiply unit is about $15–30 \times$ faster and $30–80 \times$ more energy-efficient in comparison to GPUs and CPUs [JBB+17]. It is so successful that Google is already deploying the fourth generation of the TPU [JHYA+21] in their data centres.

FPGAs and ASICs differ in one important detail: While the former are less efficient than the latter due to their Look-Up Table (LUT) based logic implementation, they are reconfigurable. This is an immense advantage, because the designs of FPGAs can be updated after deployment in, for instance, a large-scale server cluster. Especially in the field of machine learning, which is changing constantly and rapidly, as shown in Figure 1.1b, the reconfigurability is important as it allows to quickly install the latest algorithms. In contrast, it takes at least two years [DPY18] to get a fresh ASIC to run a new workload. This detour is not only time-consuming due to the design, manufacture and deployment of a new ASIC but also expensive, because of their high non-recurring engineering costs.

With all these benefits, FPGAs have become the substrate of choice to implement accelerators. Microsoft augmented the servers in their data centres with FPGAs to offload computationally demanding workloads and profit from their power efficiency and low cost [PJM+14]. The deep learning tasks in their cloud computing platform Azure and their Bing ranking are accelerated based on Project Catapult [ORK+15] and Project Brainwave [CFO+18, FOP+18] to serve Deep Neural Networks (DNNs) at data centre scale.
1.1 Challenges

**Challenge 1 — FPGAs Are Hard to Program for Non-Experts**

The flexibility of FPGAs, however, comes with extra burden placed on the programmer. Designing new accelerators is a manual, time consuming and error-prone process. FPGAs offer many options for tweaking the performance but in return require hardware expertise and take costly development time to be configured properly. Current HDLs are not suitable for a rapid-development cycle and there is a lack of high-level abstractions for efficient hardware design. Languages such as Bluespec [Nik08] and Chisel [BVR12] reduce the amount of boiler plate code required, but remain low-level.

Traditional HLS tools [Int20, Xil21] try to simplify hardware development but, despite their name, they are still fairly low-level and require some hardware knowledge to achieve high performance. Many hardware concepts are still transpiring through the high-level abstractions offered by their languages. Furthermore, these toolchains are typically built using traditional software compiler passes that are not a good match for hardware optimisation. The ‘innocent-looking software code’ [GLC+20] makes it hard to predict what the hardware will look like at the end of the compilation process [NAT+20] and leads to slow designs [GLC+20] with a large performance gap [SWLR13, WBC13] compared to hand written HDL code. In order to make accelerators accessible for non-hardware experts and fully automate their design, high-level programming abstractions are needed, that hide all hardware details.

*Functional languages* are well suited for such abstractions, because they focus on the algorithm, not on the implementation. They specify what needs to be achieved but leave open how to get there [Hud89]. Contrary to imperative languages, no notion of sequencing of commands is predetermined. Moreover, since (pure) functional languages are free of side effects and hidden states, it is ‘almost embarrassingly easy’ [Ham94] to parallelise them for better performance.

Especially now, the functional programming paradigm shows itself to be particularly useful. By offering an alternative to conventional (software-like) von Neumann style programming, it helps to think outside the box of von Neumann hardware architectures [Bac78].

Existing frameworks, such as DNNWeaver [SPM+16], Aetherling [DFH+20] and Spatial [KFP+18], make programming FPGAs easier but sacrifice expressiveness and are restricted to certain application domains. DNNWeaver uses a parameterised
monolithic hardware design, which mitigates expressiveness, maintainability and modularity. According to [Bl19], a more fine-grained, flexible computing system is desirable instead. Aetherling is limited in their use-cases and simply cannot realise matrix multiplication, nor 2D convolution. Other approaches like Spatial raise the abstraction but do still not reach a hardware-agnostic level, so that they only work in the hands of hardware experts.

**Challenge II — It Is a Long Journey From Algorithm to Hardware Design**

Once the input algorithm is specified in a functional language, a compiler takes over to generate a hardware implementation from it. Nevertheless, designing such a compiler is anything but trivial.

A multi-level IR, in the spirit of MLIR [LAB+20] for instance, leads to an elegant compiler design and implementation by decomposing compilation into a gradual lowering process with manageable, simple steps. The input algorithm goes through lowering passes that introduce more and more implementation details, until a hardware description is obtained, that is detailed enough to be installed on the FPGA.

Each level deals with one particular design aspect only. This *orthogonalisation of concerns* leads to effective design exploration and efficient implementations [KNRSV00]. Furthermore, this enables easy extensions for novel optimisation techniques or hardware targets. Especially in the field of accelerators, the ability to adapt quickly is essential because their algorithms and workloads change rapidly. Developers can precisely pick the IR they want to operate on. Similarly, static analysis tools can choose the level of abstraction that provides all the details they need.

When it comes to describing low-level IRs, functional languages again turn out to be the ideal candidate, because mathematical functions and hardware circuits have much in common [She84]. A function has one or more parameters and a return value. Similarly, a hardware module can have input ports and an output port. A function call with an argument is comparable to a hardware module whose inputs are connected to further hardware modules (one for each argument). Just like mathematical functions, circuits are composable. For instance, the composition \( g \circ f \) can represent a module \( f \), whose output is directly connected to the input of module \( g \).

Recent years have witnessed a shift towards multi-level, functional approaches for high-performance computing. Delite [SBL+14], Lift [SFLD15] and Futhark [HSE+17] have demonstrated that high-level abstractions and high-performance can go hand in hand. More recently, Lift-hls [KBSD19] and Aetherling [DFH+20] have proven
capable of producing accelerator designs. These approaches expose design decisions in their functional IRs to exploit parallelism, by distinguishing between space and time types.

However, their proposed IRs fall short of the mark. They lack explicit support for memory operations and further device-specific optimisations, which severely restricts their use on real hardware. Aetherling, for instance, has only produced results in simulation with an overly simplified memory model.

Maximising performance requires to leverage all the available hardware features, such as on-chip and off-chip memories. Given that such concepts are absent from the above mentioned IRs, their associated compilers are unable to exploit these capabilities when aiming for high-performance FPGA designs.

**Challenge III — The Design Space of FPGAs is Vast**

The flexible hardware of FPGAs offers a wide range of optimisation possibilities around parallelisation, memory usage, special hardware features and others. This spans a vast design space, where manual optimisation is tedious. Automation is needed to explore it and make optimal choices for peak performance hardware designs.

An automatic system based on **rewrite rules** is a promising approach to define such optimisations in an exchangeable, maintainable and extendable way. Functional IRs enable easy pattern matching and are therefore particularly well suited for rewrite-based optimisations. That is why rewrite rules have found approval in the compiler community and are employed by Lift [SFLD15], Lift-hls [KBSD19], Aetherling [DFH+20] and Spiral [SP19] for example. Moreover, the multi-level structure of some approaches allows making design choices on many different levels of abstraction.

Despite the success of prior work, several challenges remain. While many approaches around rewriting optimisations do not target FPGAs at all, those who do lack important FPGA-specific optimisations regarding memory usage and data reshaping. The latter is required to express important workloads such as matrix multiplication or convolution. In particular, as we will see later in this thesis, **functional approaches in general** lack a systematic approach to deal with data reshaping operations efficiently. These deficits must be optimised away to achieve high-performance FPGA designs.
1.2 Contributions

In this section, it is briefly described how the previously identified challenges are addressed in the following technical chapters. This thesis makes the following research contributions, some of which are also published in [SJD22a, SJD22b]:

A Multi-Level IR Structure for Designing Hardware Accelerators

This thesis is centred around Shir, a framework with a functional, pattern-based, data-parallel language in the spirit of Lift [SFLD15], and a compiler. Shir contains common parallel patterns, such as Map and Reduce, combined with a rich type hierarchy. Furthermore, a novel multi-level IR structure for accelerator design is designed and implemented. The IRs are described in Chapter 4.

Starting with a higher-level, hardware-agnostic functional programming abstraction as the entry point, Shir allows even non-hardware experts to specify the input algorithm, which is later turned automatically into an FPGA implementation. It also serves as an intermediate language when targeted by other high-level tools. The next lower level IR introduces basic architecture design decisions, which are explicitly encoded in Shir’s strong type system. This covers different forms of parallelism in a similar style to Lift-hls and Aetherling.

Uniquely, Shir represents hardware memory concepts in a functional style in dedicated IRs. The novel ramarray type embodies data in memory. Furthermore, the IR supports synchronous data transfers for on-chip block memory, asynchronous transfers for onboard memories and asynchronous Direct Memory Access (DMA) for host memory. Such features are explicitly exposed in the IR, for example synchronicity is directly encoded in the function type as an effect. Access to shared memories is managed by arbiters. Common memory operations are made accessible via corresponding primitives to enable their efficient use.

Another IR at the bottom of the multi-level structure enables device-specific design decisions. Exposing such features at the IR level is essential to fully support the synthesis of high-performance hardware.

Mechanical Lowering Procedures From Algorithm to Hardware

Along with the multi-level IRs, a compiler has been developed that automatically lowers an input program through the multi-level IRs until finally HDL code is emitted.
Common hardware synthesis then turns this code into a bitstream for FPGAs. These lowering steps in the compiler are specified as mechanical procedures and do not require any user input. During compilation, data types are prepared for off-chip transfers, e.g., by packing and padding. Arbiters are automatically inserted to deal with shared resources. This entire process is defined and explained in Chapter 5.

**Rewrite Rules for Optimisation**

Rewrite rules for automatic optimisation have been specified, implemented and evaluated. They are designed to be convergent, so that they can be applied in simple fixed-point iterations, which ends when no further rewrite is possible. Rewriting takes place on different abstraction levels, once again leveraging the multi-level structure in Shir. Chapter 6 explains the rewrite rules.

One set of rewrite rules deals with parallelism by expanding the design on the FPGA and using as many hardware resources as available. Further rules are responsible for inserting memory buffers, parallelising data transfers between them and the computation, and for optimising DMA communication with off-chip memories to maximise performance.

The main causes of hardware inefficiencies associated with functional data reshaping operations are identified and addressed using a set of rewrite rules. The issues occur when repeating, transposing or sliding data and are related to needs for large on-chip buffers or complex signal wiring, in order to perform the reshaping operations. The impact of the rewriting is discussed and evaluated. The performance results show that without the optimisations presented, a functional approach either produces designs that do not synthesise due to resource constraints, or produces designs with abysmal performance.

**Evaluation on a Real FPGA**

The viability of the presented Shir framework and its optimisations has been demonstrated running stencil computations, matrix multiplication and 2D convolution on an Intel Arria 10 FPGA. Using a real FPGA instead of simulation requires more engineering efforts but puts the framework to the ultimate test, because physical timing phenomena start to play a crucial role and may make the generated hardware unfeasible. The results testify to correct and efficient hardware designs, which are competitive with those generated by Intel’s OpenCL HLS. They are presented in Chapter 7.
1.3 Structure

This chapter has introduced this thesis and motivated the research contributions. The rest of this thesis is organised as follows:

Part I — State of the Art

Chapter 2 provides the technical background about FPGAs and functional programming languages to understand this work.

Chapter 3 discusses related work about similar HLS approaches and how they compare to this thesis.

Part II — Contributions

Chapter 4 develops a multi-level IR structure that spans over a hardware-agnostic and multiple hardware-specific abstractions. Each level with its primitives and types is described separately, while their ability to express different hardware design decisions is displayed.

Chapter 5 describes a systematic compilation through the levels of IRs, until VHSIC Hardware Description Language (VHDL) code for the FPGA is generated. This process is based on multiple automatic compiler passes.

Chapter 6 explores optimisations based on an automatic rewriting system. The presented rewrite rules specify how the IR is modified to, e.g., parallelise the computation, insert memory buffers, optimise data reshaping and exploit device-specific features.

Chapter 7 evaluates the viability of this approach on matrix multiplication and 2D convolution, showing the effects of the optimisations presented. In order to obtain realistic results, all experiments are performed on a real FPGA.

Part III — Conclusions

Chapter 8 summarises the contributions of this thesis, critically analyses its limitations and proposes possible future extensions.
Part I

State of the Art
Chapter 2

Hardware Design and Functional Languages

This thesis deals with synthesising accelerators from a high-level functional language. To understand the rest of it, this chapter presents the required technical background.

First, Section 2.1 provides an overview of the targeted hardware platform, namely FPGAs. Understanding this type of integrated circuits is crucial for taking advantage of them properly. They come with many specialised features that must be exploited to achieve competitive performance.

Then, in Section 2.2, the paradigm of functional programming is introduced. More specifically, an extension of lambda calculus that supports type variables and subtyping relationships, namely ‘System F with Subtyping’, is described, because it forms the basis for the functional IRs in this thesis.

Last, Section 2.3 presents rewriting systems with their properties to illustrate how programs are optimised later in Chapter 6 of this thesis.

2.1 Digital Hardware Design With FPGAs

FPGAs are integrated circuits that are configured by the consumer after being manufactured, when the hardware is already in the field. The configuration describes the desired function, e.g., accelerate a certain computational task. FPGAs come with the great advantage that they are reconfigurable. This allows them to be updated, for instance, to fix bugs or to employ new algorithms. They can keep up with quickly changing demands.
Table 2.1: Comparison of CPUs to FPGAs.

<table>
<thead>
<tr>
<th>CPU</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>general-purpose</td>
<td>specialised</td>
</tr>
<tr>
<td>‘cheap’ software code</td>
<td>‘expensive’ hardware description</td>
</tr>
<tr>
<td>central register file</td>
<td>distributed memories</td>
</tr>
<tr>
<td>fixed pipeline</td>
<td>flexible dataflow</td>
</tr>
<tr>
<td>predefined data types</td>
<td>arbitrary precision</td>
</tr>
<tr>
<td>not power efficient</td>
<td>very efficient</td>
</tr>
</tbody>
</table>

When programming an FPGA, real physical connections with wires on the chip are modified according to the program’s needs. This is unlike CPU or GPU programming, where the hardware is fixed and only the instructions are changed. The *hardware description* code for FPGAs is at a much lower level than the software code. Further differences between CPUs and FPGAs are listed in Table 2.1.

FPGAs offer a flexible, highly intrinsically parallel structure to efficiently implement arbitrary workloads. However, this flexibility comes with the price that the *design space* is vast and hardware developers must *explore* it to find suitable implementations.

FPGAs are employed in high-throughput, high-performance tasks like digital signal processing, image and video processing, cryptography, and many other applications. In the design of ASICs, full system verification quickly becomes too complex for simulation. FPGAs are well suited to build a prototype for functional testing and for getting rid of bugs early on. This prevents reengineering and producing defective ASICs that would otherwise increase cost and Time-To-Market (TTM) [Ou05]. To keep the TTM short, FPGAs can also be integrated directly in the final product, so that the manufacturing of a custom ASIC is no longer necessary.

The energy efficiency of FPGAs is valuable for large-scale data centres that run compute intense workloads, like machine learning algorithms, as it considerably cuts the cost for electricity. Their reconfigurability furthermore allows to quickly deploy new efficient workloads, which is why they are already widely used in large numbers in data centres [CFO+18, FOP+18].
2.1. Digital Hardware Design With FPGAs

Figure 2.1: Structure of an Intel Arria 10 FPGA chip. Next to the core logic fabric, there are many specialised hard Intellectual Property (IP) blocks for efficient memories, ethernet communication and computation to make the overall chip more efficient. Figure adapted from [Int22c].

2.1.1 FPGA Chip Structure

The central building blocks of FPGAs are Configurable Logic Blocks (CLBs), also called Adaptive Logic Modules (ALMs) on Intel FPGAs. They consist of LUTs, registers and supplementary logic. Switch blocks configure the interconnect between these modules. All these ingredients allow the device to build any digital function, as long as it does not require more logic or memory than available on the device [FMM12].

Additionally, more specialised, hardened components are available on the chip next to the core logic blocks, as depicted in Figure 2.1. They are characterised by their high power efficiency, which is up to $10 \times$ higher [OVP+22] than a similar function in programmable logic. Digital Signal Processor (DSP) blocks enable efficient arithmetic operations on integers or floating point numbers. If the data precision is sufficiently low (18 bits or less), one DSP block in Figure 2.1 performs two integer multiplications at the same time. The Intel Arria 10, for instance, has 1518 of such DSP blocks and is therefore able to compute 3036 integer multiplications per clock cycle.

Moreover, there are M20K internal block Random Access Memory (RAM) units distributed on the chip. Each block can hold a total of 20,480 bits, hence the name, in different sizes, e.g., $512 \times 40$ bits, $2048 \times 10$ bits or $16,384 \times 1$ bit [Int22b].

The hard Ethernet IP supports the Ethernet protocol stack with data speeds from 10 G to 400 G.
Figure 2.2: Memories typically available in an FPGA system. Due to their sizes, access speeds and locations, the memories are used for different purposes in the system.

Another hard IP implements Peripheral Component Interconnect Express (PCIe) Gen 3 to communicate with the host. Further hard memory controllers allow to access the large off-chip memories, as will be explained in Section 2.1.2 more thoroughly. Phase-Locked Loops (PLLs) provide a clock source for the functions on chip.

In summary, modern FPGAs offer lots of specialised functions to increase the power efficiency of the device. When programming FPGAs, this must be taken into account and these functions must be exploited accordingly.

### 2.1.2 Memories in FPGA Systems

The flexibility of the CLBs allows their registers to be used as RAM cells. This kind of memory usually spreads out over multiple logic blocks and is therefore also called distributed RAM. Since each CLB only has a few registers, larger distributed RAMs occupy a lot of the FPGA’s programmable logic resources, which are then no longer available for other operations. For this reason, FPGA systems typically come with different kinds of dedicated memories for various purposes. Figure 2.2 depicts them and where they are located in the system.

The FPGA chip itself contains fast RAM blocks. This is the fastest of all the memories in the system, because data can be read or written at each clock cycle and very wide data widths are supported when multiple of these discrete memory blocks are combined. However, with a total amount of a few megabytes only, block RAM is not sufficient for larger intermediate results of the computation.

That is why some systems include a few banks of off-chip memory on the FPGA board. The Intel Arria 10, for instance, has two DDR RAM banks. Each delivers about 12 GB/s through a single shared channel for both read and write access. Thus, a total of up to 24 GB/s is reached if the data is efficiently distributed among these banks.
2.1. Digital Hardware Design With FPGAs

For FPGAs that are connected with a host via PCIe, the host RAM may be directly accessible using DMA. If present, this large but slow memory provides the input data for the FPGA and is the destination when writing back the computed results. The Core Cache Interface (CCI-P) carries 512-bit wide cache lines from the host machine’s CPU [Int22a]. All data must be reshaped and padded accordingly before being transmitted.

Contrary to the synchronous block RAM, off-chip communication to memories or the host machine works asynchronously. For asynchronous memory access, the FPGA’s memory controller sends requests to read or write in a non-blocking manner and some time later the memory responds with the desired data (or just an acknowledgement in the case of write). In the meantime, further requests can be sent until the queue is filled with pending requests (64 on the Intel Arria 10 [Int22a]). To maximise memory throughput, this queue should be filled as much as possible.

2.1.3 Hardware Description Languages

HDLs describe digital hardware systems. They cover different levels of abstraction, one of which is the Register-Transfer Level (RTL), which accurately models the flow of signals through logical operations between registers in hardware. The two most widely used languages are VHDL [IEE09] and Verilog [IEE06]. The former is strongly typed and strictly enforces correct typing, while the latter is more permissive. Both languages well support the FPGAs from the two major vendors Xilinx (now AMD) and Altera (now Intel). This thesis is confined to VHDL.

VHDL was originally created in 1983 by the US Department of Defense to document their ASICs at the time, before it became part of the IEEE 1076 standard [IEE09]. It turned out that these descriptions are accurate enough to allow for logic simulation and even for the creation of physical implementations of the circuit itself.

Unlike serial code, where the lines of code are sequentially executed one at a time, HDLs model a concurrent system. In such parallel logic languages, all of the lines of code are executed at the same time, which entails a completely different kind of programming compared to software.

2.1.4 Logic Synthesis

Once the hardware description is written in an HDL, it takes a few synthesis steps to generate the final bitstream to install on the FPGA, as seen in Figure 2.3. The hardware vendor provides the synthesis tools for that.
First, the logic synthesis converts the HDL into a gate-level netlist, which is a list of logic elements and interconnections that implement the described behaviour. At the same time, the logic operations are optimised and redundant logic is eliminated.

Next, the generic logic elements are mapped to technology specific gates, which are present on the target FPGA device. In the place and route phase, they are placed at a specific location on the chip and the wires to connect the elements are routed.

Before generating the actual bitstream from this resulting design, it is checked for timing violations using timing analyser tools supplied by the FPGA vendors. The following section explains what these errors are and how to fix them.

### 2.1.5 Propagation Delay

When a signal in hardware travels from one register to the next one, it does not arrive instantly from a physical point of view. A certain amount of time passes, which is referred to as the propagation delay.

The more complex the combinational logic for a signal between two consecutive registers, the larger the delay. The signal path with the largest delay in the entire circuit is the critical path. In order for the circuit to work correctly, the critical path delay must not exceed the clock period. Otherwise signal value changes may not reach the registers in time, which leads to unpredictable errors during runtime. In more mathematical words, the maximum clock frequency $f_{\text{MAX}}$ is limited by the inverse of the critical path delay: $f_{\text{MAX}} < \frac{1}{\text{criticalDelay}}$.

If the critical path in a design is too long for the clock frequency, the timing analysis tools will identify this timing violation, which must be resolved in one of two ways: Either the clock frequency is decreased, which impairs the throughput of the FPGA; Or the hardware engineer reduces the critical path delay using pipelining, as demonstrated in Figure 2.4. The latter is generally the preferred way, as it keeps the $f_{\text{MAX}}$ high, which allows to achieve high performance with the FPGA.
2.1. Digital Hardware Design With FPGAs

2.1.6 High-Level Synthesis

Coding in HDLs is time-consuming and requires hardware expertise, since these languages describe the design on a low, detailed level. To improve the productivity of developers, HLS raises the abstraction of the input description. Instead of timed RTL code, only a high-level behavioural specification is needed for this process. The input languages are typically subsets of C or its variants, e.g., C++, OpenCL [Mun09], but also MATLAB or other more academic programming languages.

HLS then automatically converts it into a digital hardware design that implements the desired behaviour. In the meantime, the HLS tools exploit spatial parallelism, pipelining and other optimisations to get the best performance out of the target platform. Further user provided constraints restrict area usage, delay times and hardware data types to guide this process. Pragma directives are commonly inserted in C-like code. For instance, the keywords #pragma HLS unroll in a for loop cause copies of the loop’s body in hardware, so that the entire loop is run concurrently.

Design flows profit from HLS, because they use common programming languages, that are accessible for non-hardware experts and allow easy arithmetic implementations. The developers are able to quickly create a larger number of implementations and try out different design choices to explore the optimisation space. Thus, the overall productivity is increased.
The following OpenCL code for HLS shows a hand-optimised implementation for matrix multiplication by Intel:

```cpp
#include "../host/inc/matrixMult.h"
#ifndef SIMD_WORK_ITEMS
#define SIMD_WORK_ITEMS 4
#endif

__kernel
// The dimension of the block used in the core computation
// How many output elements will be computed in a SIMD manner
void matrixMult( // Input and output matrices
    __global uchar *restrict C, __global uchar *A, __global uchar *B,
    // Widths of matrices.
    int A_width, int B_width)
{
    // Local storage for a block of input matrices A and B
    __local uchar A_local[BLOCK_SIZE][BLOCK_SIZE];
    __local uchar B_local[BLOCK_SIZE][BLOCK_SIZE];
    // Block index
    int block_x = get_group_id(0);
    int block_y = get_group_id(1);
    // Local ID index (offset within a block)
    int local_x = get_local_id(0);
    int local_y = get_local_id(1);
    // Compute loop bounds
    int a_start = A_width * BLOCK_SIZE + block_x;
    int a_end = a_start + A_width - 1;
    int b_start = BLOCK_SIZE + block_y;
    // Compute the matrix multiplication result for this output element.
    // Each loop iteration processes one block of the matrix.
    uchar running_sum = 0.0f;
    for (int a = a_start, b = b_start; a <= a_end; a += BLOCK_SIZE, b += (BLOCK_SIZE * B_width))
    {
        // Load the matrices to local memory.
        A.local[local_y][local_x] = A[a + A_width * local_y + local_x];
        B.local[local_x][local_y] = B[b + B_width * local_y + local_x];
        // Wait for the entire block to be loaded.
        barrier(CLK_LOCAL_MEM_FENCE);
        // Do the dot product accumulation within this block. Fully unroll the loop.
    }
}
```

---

1https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html
#pragma unroll
for (int k = 0; k < BLOCK_SIZE; ++k)
{
    running_sum += A_local[local_y][k] * B_local[local_x][k];
}
// Wait for the block to be fully consumed before loading the next block.
barrier(CLK_LOCAL_MEM_FENCE);

// Store result in matrix C
C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = running_sum;

2.2 Functional Programming

Functional programming is a programming paradigm for expressing computation by forming functions by abstraction and applying functions to arguments. Common examples of functional programming languages are Haskell [Mar10] and ML [MHMT97]. In contrast to imperative programming languages, they are declarative and state what to achieve instead of how to get the desired outcome [Hud89].

In functional programming, the functions are more mathematical functions than procedures. They are handled just like data: They can be passed as arguments, returned from functions or bound to names.

Functions that operate on other functions, by either receiving them as inputs or returning them as outputs, are referred to as higher-order functions. Common examples are map and reduce.

The function map takes both an unary function \( f \) and a list of \( N \) values \( i \) to apply the function to each element from the list:

\[
\text{map}(f, [i_1, i_2, \ldots, i_N]) \equiv [f(i_1), f(i_2), \ldots, f(i_N)]
\] (2.1)

The function reduce takes a binary operator \( f \), an initial element and a list of \( N \) values \( i \) to bring down the input list to a single value. In case of inserting the addition \((+)\) for the function and using 0 as the initial element, the sum of a list is computed:

\[
\text{reduce}(+, 0, [i_1, i_2, \ldots, i_N]) \equiv 0 + i_1 + i_2 + \cdots + i_N
\] (2.2)
Functional programming offers some advantages when pure functions are used. Whenever such functions are called with the same argument, the same result will always be returned. They have not side effects and no non-local mutable state is accessed. These attributes help to establish a safe design. More complex behaviour is safely composed by combining smaller modular functions. In contrast, a function that modifies a global state may cause errors, when integrating it into a larger system.

The theoretical foundation of functional programming languages is formed by lambda calculus, which is introduced in the next section.

### 2.2.1 Untyped Lambda Calculus

Developed in the 1930s by Alonzo Church, the lambda calculus [Chu32, Chu33] is a Turing complete, formal system of computation. As the grammar in Figure 2.5 indicates, lambda calculus is a notation centred around anonymous functions (abstractions) and their application. Despite its simplicity, it is expressive and flexible [AK21].

To disambiguate terms, expressions can be wrapped by parentheses. Apart from that, function applications are left associative, so that \( x y z \) is equal to \( ((x y) z) \). Abstractions always extend as far right as possible, if not restricted by parentheses. The expression \( \lambda x . y z \) does not apply \( z \) to the abstraction over \( x \). Instead, \( z \) is applied to \( y \) before creating the abstraction: \( \lambda x . (y z) \)

Abstractions bind a certain variable in its body, the right part after the dot. The left part is therefore also referred to as the binder. Variables in an expression \( e \) are unbound or free, if they are referenced in \( e \) but not enclosed in an abstraction within \( e \) that declares them. For instance, the variable \( y \) is free in the expression \( \lambda x . y \). Even if they are bound to an abstraction, which is outside of the considered scope of the expression, they are still (locally) free.
2.2. Functional Programming

2.2.2 System F With Subtyping

In [Chu40], Church assigned types to all the terms in lambda calculus. That is how the simply-typed lambda calculus was created.

The syntax is extended with a colon notation to denote that variable \( x \) has a certain type. The following example identity function requires an integer type \( IntT \):

\[
\lambda x : IntT \cdot x
\]

Whenever this function is called, the given argument must be of the same type \( IntT \). This restriction makes the lambda calculus computationally weaker, but allows for static analysis, for instance to guarantee that a program could evaluate successfully.

However, if a generic function is needed that works on different types, it must be redefined for each single one of them. System \( F \) [Gir71, Rey74, Gir86] solves this problem by introducing polymorphism with type variables to define functions that can be used with multiple types. Essentially, lambda calculus is brought to the type level with type abstractions and type applications. Type-level functions are expressed with an upper-case lambda: \( \Lambda \). For instance, the type of the above identity function can first be abstracted out with the type variable \( TV \):

\[
\Lambda TV \cdot \lambda x : TV \cdot x
\]

Then, this generic function is instantiated with a concrete type by applying a type as argument:

\[
(\Lambda TV \cdot \lambda x : TV \cdot x) \cdot IntT
\]

On this basis, System \( F \)-Sub (System \( F_{<} \)) [CMMS91] introduces a notion of subtyping. This enables flexible control over the types by building a type hierarchy and then restricting terms to certain parts of the type hierarchy only. A single abstraction can now be created for a selected group of different types — a subtree in the hierarchical tree of types.

In a type system, where \( IntT \) is a subtype of \( ScalarT \), a function’s integer return type can be viewed as a simple scalar (covariance). For the input type of a function, it is the other way around (contravariance) [Cas95]. For instance, if a function accepts all kinds of scalars for inputs, it can still be applied on more complex subtypes of scalar, e.g., an integer.

In this thesis, subtyping relationships are denoted with the superscript, e.g., \( T^U \) means that \( T \) must be a subtype of \( U \).
2.3 Term Rewriting Systems

Rewrite rules specify how and under which circumstances one term in a Term Rewriting System (TRS) is transformed into another term. This mechanism can be used to replace a term with a better, e.g., more efficient, one. After rewriting, the optimisation is explicitly encoded in the new term.

In this thesis, the long double arrow $x \rightarrow y$ denotes a rule to rewrite $x$ as $y$, while $x \Rightarrow^* y$ describes a sequence of rewrites leading from $x$ to $y$. Rules may also have an optional condition, written on top of the arrow, to restrict their application. The following rewrite rule replaces the primitive $\text{PrimitiveA}$ by $\text{PrimitiveB}$ if the input, which is fed into the primitive, is sufficiently small:

\[
\begin{align*}
\text{input} & \Rightarrow \text{PrimitiveA} & & \text{// required term} \\
& \text{if } \text{input} < 42 & & \text{// rewrite with condition} \quad (2.6) \\
\text{input} & \Rightarrow \text{PrimitiveB} & & \text{// rewritten term}
\end{align*}
\]

After rewriting with this example rule, everything apart from $\text{PrimitiveA}$ remains untouched. In case $\text{PrimitiveA}$ occurs multiple times in the given term and the condition is still satisfied, the rewrite rule can be applied more than once.

Convergent Systems

The optimisations that will be presented in this thesis, consist of at least one but often an entire set of rewrite rules. In the latter case, rewriting can be non-deterministic, as it is unclear in which order and how much the rules are applied. This is often unsuitable for optimisations, because it does not guarantee to yield an actual improvement.

In order to make the rewriting deterministic, either a fixed order must be provided for each input term and each rule set or, to avoid this complex step, it must be ensured that the outcome of the rewriting remains independent from the order of the rules. For this, the TRS must be terminating and confluent and therefore convergent \cite{BN98, DJ90, Klo93}.

2.3.1 Termination

A rewriting system is terminating (also called strongly normalising and noetherian), if no infinite chain of rewrites exists. All sequences eventually must terminate.
2.3. Term Rewriting Systems

For all children of a common ancestor, there exists a common descendent.

Figure 2.6: Confluence in a rewrite graph, where nodes capture a term’s state and edges represent rewrites. All the paths that diverge from a common ancestor ($t_0$) will eventually end at common descendent node ($t_3$) [BN98].

In such a system, the term’s normal form, where no rewrite is applicable any more, can always be reached after a finite amount of rewrite steps. Thus, an iterative rewrite process always yields a term in normal form, similar to a fixed-point iteration.

2.3.2 Confluence

A rewriting system is confluent, if its terms can be rewritten in multiple ways, e.g., different order of rule applications, and still yield the same result. The rewrite graphs of such systems, as visualised in Figure 2.6, contain the shape of a diamond. When deriving the terms $t_1$ and $t_2$ from a given term $t_0$ in such a system, a common term $t_3$ can always be found that joins $t_1$ and $t_2$: $\forall t_0, t_1, t_2 : (t_0 \Rightarrow^* t_1) \land (t_0 \Rightarrow^* t_2) \rightarrow \exists t_3 : (t_1 \Rightarrow^* t_3) \land (t_2 \Rightarrow^* t_3)$ [BN98]. Thus, a term in a confluent rewriting system has at most one normal form.
Chapter 3
Related Work

This chapter presents the state of the art relevant to this thesis and discusses how this thesis addresses the limitations of previous work. In particular, hardware design approaches and functional representations are covered.

The first section, 3.1, lists specialised implementations for accelerators based on coarse-grained hardware templates. Section 3.2 describes programming languages for hardware design. In Section 3.3, higher level approaches to synthesise hardware are presented. Then, Section 3.4 shows recent work on representations with multiple levels of abstraction. Related domain-specific languages are discussed in Section 3.5. Finally, Section 3.6 deals with closely related work. This last section also summarises the key differences between these and the features contributed by this thesis.

3.1 Monolithic Template-Based Accelerators

Current accelerator research is bursting with new and increasingly efficient implementations based on coarse-grained hardware templates. A large part of it targets machine learning workloads \([HXL+16, QWY+16, SCD+16, WYZ+17]\), and CNP \([FPHL09]\), DeepBurning \([WXH+16]\), FP-DNN \([GLX+17]\), PipeCNN \([WXJ17]\), FINN \([UFG+17, BPF+18]\), MAGNet \([VSW+19]\), to name just a few. The sheer volume of publications demonstrates the need for FPGA-based accelerators. This section takes a closer look on a selection of them.

**DNNWeaver** \([SPM+16, MCV+19]\) is a parameterised generator for deep learning accelerators for inference with DNNs. It handles various Convolutional Neural Network (CNN) configurations with different input matrix and weight sizes, as well as low data precision, by adjusting the parameters of the hand-optimised Verilog templates.
DNNWeaver’s hardware architecture consists of processing units that contain many processing elements and a dedicated buffer for convolutional weights. Each processing element implements multiply and multiply-add operations with its Arithmetic Logic Unit (ALU).

The input model is specified with Caffe [JSD+14], a framework for deep learning algorithms with a collection of reference models. This high-level specification is then translated into Instruction Set Architecture (ISA) code, which is then again compiled into an execution schedule. After that, the hardware templates are customised and optimised using a heuristic algorithm. The optimisations range from parallelisation (e.g., the number of processing elements in a processing unit) over data reuse in the layers to scheduling and they take the available resources on the target FPGA into account. DNNWeaver achieves independence from FPGA vendors, by first generating a generic DRAM interface for the design and later adding device-specific glue between this and the platform’s actual hardware interface.

Tabla [MPA+16] is a template-based framework to generate accelerators for the training phase of a class of machine learning algorithms. It exploits the fact that many learning algorithms can be expressed as solving an optimisation problem, that minimises an objective function using stochastic gradient descent. The user only has to specify this objective function in a high-level programming model as the input to this framework, together with the description of the target FPGA’s available features.Tabla then uses predefined templates designed by hardware experts to produce a set of Verilog files for synthesis.

fpgaConvNet [VB16, VB18] is another framework that maps CNNs onto FPGAs. The user provides a high-level description of the CNN’s layers and the target FPGA platform. The framework first turns this specification into an Synchronous Data Flow (SDF) graph, where each node represents a layer. By applying algebraic transformations on this graph, the design space is explored to find the design with the highest throughput. This optimisation process is based on simulated annealing and maintains the functional equivalence. It covers coarse-grained parallelism, where all the layer’s operations are run in parallel, and fine-grained parallelism, where the dot product is unrolled to be computed in a single clock cycle. Finally, the optimised graph is mapped to a set of five tuneable hardware building blocks: Sliding Window Block, Fork Unit, Convolution and Pooling Banks, Nonlinear Bank and Memory I/O Unit. A synthesisable Vivado HLS design is then generated from these blocks. The fpgaConvNet framework supports time sharing across the layers. The large CNN is
partitioned into several parts, which are all run separately on the FPGA. In between, the FPGA is reconfigured, while intermediate results are held in off-chip memory.

TVM [CMJ+18] takes a model from a high-level deep learning framework, such as TensorFlow [ABC+16], MXNet [CLL+15] and PyTorch [PGM+19], in the Open Neural Network Exchange (ONNX) format [BLZ+19] and generates optimised code for CPU, GPU and FPGA-based back ends. A graph rewriter transforms the program on the high level to generate variants of the implementation. Low operator-level optimisations are proposed by a schedule explorer and then automatically selected based on a machine learning cost model.

Regarding the FPGA back end, a Vanilla Deep Learning Accelerator prototype is targeted. Inspired by previous accelerators like the TPU [JBB+17], its fixed architecture is based on a tensor processor for efficient matrix multiplication and convolution.

fBLAS [DMdFLH20] implements the widespread Basic Linear Algebra Subprograms (BLAS) specification for FPGA. This enables the host system to offload common linear algebra operations, such as the dot product and matrix multiplication, to its FPGA and accelerate them. There is no need for the developer to write any hardware-specific code, a simple library call suffices. fBLAS consists of optimised OpenCL HLS modules, one for each of the BLAS routines. By tuning their parameters, they can be adapted to satisfy the performance requirements and resource constraints of the given FPGA. However, if the given algorithms or parts of them do not fit directly to the implemented functions, the performance degrades. Further optimisations for such algorithms remain inaccessible, because as a library-based approach, fBLAS is unable to think outside the 'subroutine box' and cannot address the performance deficits from a higher level.

More generic workloads can be accelerated with [YTT+08] and FPMR [SWY+10]. They bring Google’s programming model MapReduce [DG04] to FPGAs. The functional MapReduce model reduces the complexity of portable (independent of the back end) and efficient parallel implementations. The user does not have to deal with data distribution to the processors and their load balancing. Only the task itself must be specified as a map function and a reduce function in C code. No hardware expertise is necessary. The generated hardware design of FPMR consists of processors for map and reduce operations, a processor scheduler and memory controllers. While many programs can be expressed as MapReduce problems, SHIR aims for an even more flexible function approach that leaves processor-based FPGA implementations behind.
Similarly, Fleet [THZ20] takes a user-defined RTL specification of a serial processor and integrates it into a massively parallel streaming design for an FPGA. It automatically instantiates the required copies of the processor and the memory controllers and connects their inputs and outputs. It focuses on token-oriented stream processing.

Discussion

The performance of the above mentioned approaches is outstanding because they are specifically optimised for their targeted applications. However, their monolithic hardware design based on coarse-grained computational kernels limits expressiveness, maintainability and modularity [BI19]. Although the large-scale HDL templates can be configured by adjusting parameters, these approaches are overspecialised and restricted to certain application types. The fixed architecture in these designs complicate the integration of future optimisation techniques.

Research in this field is stuck in a rut [BI19]. Innovative research ideas do not stand a chance against the dominant programming model, if compared by performance only. Nevertheless, if the metrics expressiveness, maintainability and modularity are considered, novel language designs point in a challenging but promising direction.

The framework in this thesis, Shir, follows the path advocated in [BI19] with functional IRs and fine-grained multi-purpose templates, that can be composed to implement the desired behaviour. While the use-cases evaluated are currently limited, the flexible Shir approach is easily extendable to support further workloads in the future.

3.2 Hardware Description Languages

This section presents different languages designed for hardware design, reaching from established, commercially used languages to more modern, academic languages. A special focus is placed on functional languages at the end of it.

3.2.1 Traditional Languages

Formerly ‘very high level’ hardware description languages like VHDL [IEE09] and Verilog [IEE06] are nowadays considered low level. With these approaches, hardware is programmed close to the gates.
3.2. Hardware Description Languages

SystemVerilog [IEE18] extends Verilog with some more advanced features. The object-oriented programming paradigm enables inheritance and polymorphism. SystemVerilog supports constrained random testing, as well as assertions, and introduces a stronger type system than Verilog. Ports can be compactly grouped as interfaces and hardware modules can be parameterised.

SystemC [IEE12] supports higher-level design stages, such as system-level modelling, similar to SystemVerilog. Technically it is a C++ library but it may easily appear as a programming language. Based on C++, it inherits powerful language constructs such as templated classes and object-orientation. The simulation kernel of SystemC enables the design of testbenches and their simulation. To generate hardware, SystemC is compiled to Verilog using synthesis tools, such as Synthesizer [Mer08].

3.2.2 Non-Functional Languages

The IRs in Shir are rooted in functional languages. But before diving into related functional approaches for hardware description in Section 3.2.3, this section provides a brief summary of less related, other types of HDLs for the sake of completeness.

Lustre [PHP87, HCRP91] and [JYH08, HKM+08, SBWM17] introduce synchronous data-flow languages that can describe hardware systems. In [Edw02, BKS03], the synchronous programming language Esterel [BG92] is used to specify control-dominated, concurrent systems. Efficient combinational circuits are generated by extracting control dependence information from a control-flow graph description.

Lime [ABCR10] is a Java-compatible object-oriented programming language from the LiquidMetal [ABB+12] project at IBM. It inherits the strong typing and many other advantageous features from Java. Thus, design of complex hardware systems is simplified with high-level abstractions, parameterised generic classes, polymorphic methods and overloaded operators. Unlike Java, Lime provides control at bit-level granularity, common in HDLs. The compiler transforms Lime code into a task-based data-flow programming model, which realises coarse-grained pipelining and data parallelism. This graph is then turned into Verilog code, in case FPGAs are the targeted platform.

JHDL [BH98] is an object-oriented standard programming abstractions that represents hardware modules as Java classes. The meaning of Java’s new keyword for memory allocation is reinterpreted to the creation of a circuit. Whenever the programmers see this keyword, they know that area on the FPGA is occupied.
Similarly, MyHDL [JS15] leverages the features of an existing programming language, Python this time, to facilitate describing hardware.

### 3.2.3 Functional Languages

**Bluespec** [Nik08] builds on SystemVerilog and includes higher-order functions. Hardware behaviour is modelled based on states and their modification. The states are modified using *guarded atomic actions*. To apply such an action, certain conditions must be satisfied, which is specified by the guard. Since they are atomic, at no time there is a partially modified state visible. An action either updates all the relevant states at once or, if not possible, leaves all of them untouched.

**SAFL** [MS00] and its extension SAFL+ [SM01] are a Statically Allocated Parallel Functional Language. As such, they focus on static allocation for all the variables to fixed storage locations, which is well suited for hardware implementations. This approach tackles the von Neumann bottleneck by leaving dynamically-allocated storage behind. Functions in SAFL are always called by value and instantiated at most once. If a function is called multiple times, it is shared among all its calls with multiplexers and arbiters in hardware. For parallelisation, functions must be duplicated and distributedly assigned to these calls on the SAFL source code level.

These restrictions lead to some advantages for hardware design. First, SAFL’s strict evaluation simplifies parallelisation. Second, the occupied area of the circuit is directly determinable by the length of the program. Basic hardware design decisions, such as resource sharing and area-time trade-offs, are made straightforwardly accessible to a high level of abstraction, reducing development time compared to traditional HDL. Furthermore, an ordered, static schedule for function evaluation can be defined using let declarations. However, higher-order functions are not supported in SAFL and the available data types are limited. Once the SAFL program is specified, the FLaSH compiler [SM00] translates it into structural Verilog via intermediate control and dataflow graph representations.

**HML** [LL95] allows to design circuits with a concise notation in the programming language standard ML [MHMT97]. Its functional basis enables more abstract descriptions with parameterised values and functions. Since the parameters themselves can be functions, they surpass generics in VHDL. Additionally, the type system is able to detect hardware design rule violations at an early stage. The compiler hml2vhdl generates VHDL code from the given HML description.
Verischemelog [JB00] is an HDL based on Scheme [SDF+09]. It addresses the shortcoming of Verilog when it comes to generating hardware descriptions. Normally textual substitution is used in Verilog to achieve parameterised hardware modules. Verilog scheme provides a macro language for generating hardware designs, which are then compiled to Verilog for synthesis and simulation.

\(\mu\text{FP} [\text{She84}]\) is an extension of Backus’ FP [Bac81]. This language combines the behavioural description as well as the layout of a circuit in one specification. It supports combinational circuits and, by using a special \(\mu\) operator, finite state machines. Due to the algebraic properties inherited from FP, hardware designs in \(\mu\text{FP}\) remain concise and can be manipulated using algebraic laws, which preserves the original program semantics.

Wired [ACS05] addresses the issue that traditional HDLs veil low-level layout decisions leading to low quality hardware. Besides the overall behaviour specification, it also captures precise details of the layout. The size and position of the wires on potentially multiple metal layers (3D) is encoded in this (impure) functional representation. When it comes to the output formats, however, this approach falters, as only a visual representation is produced.

ForSyDe [SJ04] presents a methodology for Formal System Design. In the beginning of the design process, the abstract and formal model is specified in Haskell. ForSyDe’s process constructors cleanly separate computation and communication. A formal and transparent refinement process turns the specification into an implementation optimised for synthesis onto the chosen architecture. This happens in multiple transformation steps, which introduce the hardware details. While this approach incorporates some beneficial concepts similar to SHIR, such as the transparent lowering procedure that we will see in Chapter 5, many other features are missing. For the conversion into a netlist, only a limited subset of Haskell is supported, excluding e.g., let bindings and higher-order functions.

Traditional HDLs lack support for formal verification, so that certain circuit properties cannot be easily proved. Lava [BCSS98] is an HDL based on Haskell, that addresses this issue with more abstract and general descriptions of hardware. This brings functional language features, such as higher-order functions, monads, type classes and polymorphism to the world of hardware generation. Later, more modern functional programming techniques and custom data types are brought to this approach with Kansas Lava [GBK+09]. It also makes larger circuits possible, compared to Lava.
More recently, Chisel [BVR+12] emerged as a hardware construction language. Embedded in Scala [OAC+04], it inherits powerful capabilities, such as polymorphism and higher-order functions in its object-oriented and functional programming style. These advanced features simplify digital hardware development but the circuit is still described at the low RTL. Chisel contains a high-speed cycle-accurate software simulator in C++. High-quality Verilog code for synthesis and simulation is generated from Chisel code using the FIRRTL [LIB16, IKL+17] hardware compiler framework. FIRRTL optimises circuits in its own IR on the RTL, which also supports user-defined transformations. If coding in another language is preferred, Magma [Han22] can describe a circuit in Python. Again, FIRRTL converts this code to Verilog.

CλaSH [SKB10, BKK+10, Baa15] enables a developer to describe hardware circuits in the functional language Haskell [Mar10]. It offers a high-level abstractions with polymorphism and higher-order functions. In contrast to traditional HDLs, the clocks are implicit, which simplifies hardware design. Furthermore, testbenches can be generated and the defined program can be simulated. To translate the Haskell code into low-level VHDL, a term rewrite system is employed. This compilation is direct without any intermediate steps and representations. Thus, no IR exists to expose design decisions and to provide an interface for optimisations. The design space can therefore not be explored. For instance, pipelined and spatial parallelism are not represented in the data types, let alone any kind of memory usage.

Discussion

All in all, these approaches exploit the paradigm of functional programming to increase the level of abstraction. Some expose hardware details in their high-level language to provide a convenient yet flexible way for the developer to make low-level design decisions. Thus, the code size is reduced and the productivity is increased compared to traditional HDLs.

However, the languages remain relatively low-level and still require programmers to understand hardware concepts, a major obstacle for non-experts. To improve the productivity even more, SHIR raises the input abstraction to a hardware-agnostic level. This thesis moreover features a transparent multi-level compiler structure to gradually add implementation details, which most of the above approaches, except ForSyDe, do not leverage.
3.3 High-Level Synthesis

HLS is the process of automatically turning an abstract behavioural specification into an RTL description for hardware. Several C-to-FPGA synthesis frameworks have been proposed that compile C/C++ or OpenCL [Mun09] code with annotations into hardware. On the commercial side, these include Intel’s OpenCL SDK [Int20], Xilinx’s Software-Defined Development Environment for Acceleration (SDAccel) [GRDT16], Xilinx’s Vivado HLS [Xil21], Catapult [Bol08], as well as LegUp [CCA+11, CCA+13], which originated in academia.

There are many more academic, often open sourced approaches, for instance CHiMPS [PBD+08], Augh [PBMR14], Dwarv [YKB+07, NSO+12, NSQ+14] and Bambu [PF13, FCC+21]. The latter is able to target different devices from different vendors and automatically generates testbenches for simulation. It makes use of available OpenMP pragmas to extract coarse-grained parallelism.

These frameworks make a great step towards productivity. They raise the level of abstraction and allow even programmers without extensive hardware knowledge to design accelerators. Existing code for CPUs can be reused for HLS and then executed on an FPGA with a much higher efficiency.

However, extracting parallelism from such imperative descriptions is a great burden for HLS compilers. These software-oriented languages are not well suited to describe hardware with coarse or fine grained parallelism and distributed memory. At the same time, dynamic memory management, pointers and recursion in software code further complicate the HLS process. To achieve high performance, the compiler requires additional clues in the form of hardware-specific manual optimisations [NSP+16]. Developers have to annotate the code with pragma directives to fix certain hardware implementation details. On top of that, these directives are often vendor-specific, which hinders HLS code portability across different vendors.

More modern HLS approaches aim at addressing these issues. Roccc [VPNH10] and Kiwi [SG08] offer more convenient ways for the developer to explicitly communicate the parallel architecture by introducing dedicated constructs. Their generated code is competitive with a manual implementation and independent from the targeted device.

HeteroCL [LCH+19] helps the developer by decoupling algorithm and hardware-specific customisation. In the code, the bit-accurate data types with quantisation, computation architectures and memory usage is kept separate from the main computation. Its Domain Specific Language (DSL) is python-based and extended from the TVM
Chapter 3. Related Work

framework [CMJ+18]. The compilation process uses the Merlin compiler [CHP+16] and is fully automated for various platforms, including FPGAs. Generally HLS code is generated but HeteroCL features two advanced back ends, SODA [CCWZ18] for stencil patterns, and the PolySa [CW18] back end for systolic array architectures.

The portability of HLS code suffers from vendor-specific C dialects and pragmas. AnyHLS [ÖPGM+20] addresses this problem by raising the abstraction level with portable (higher-order) functional abstractions to describe the algorithm. The source code of AnyHLS is free from vendor-specific pragmas and separated from its hardware implementation. This approach extends the AnyDSL compiler framework [LBH+18] to perform partial evaluation and to generate optimised code for Intel and Xilinx HLS.

According to [GLC+20], the performance of hardware created by HLS is impaired by timing issues, which force the clock frequency to be lowered. Existing tools do not provide helpful feedback to improve the timing and the HLS code itself does not expose the required details for this. It is difficult for developers to notice the potentially disastrous impact of ‘innocent-looking software code on the timing of the synthesised hardware’ [GLC+20]. The authors identified broadcast structures with high-fanout signals and long critical paths as the origin of timing issues in HLS tools and present techniques to address them.

Dahlia [NAT+20] identifies further pitfalls in HLS approaches. Black-box heuristics in industrial HLS tools often lead to unpredictable results and allow too little insight to understand and fix a potential performance problem. A small modification of a parameter in the code can cause drastic changes to the hardware implementation. Design decisions that are supposed to improve the performance can counterintuitively degrade it. This makes hardware design difficult for the programmer.

In order to improve the predictability of HLS processes, Dahlia proposes a type system that restricts the programs to the ones with a predictable compilation only. Thus, the programmer regains control over design decisions. There is no randomness in Dahlia’s HLS and the source code directly indicates what the generated hardware will look like. This method limits the exploratory range in the design space, so that only a subset of architectures that are supported by HLS can be expressed. However, Pareto-optimal designs are still found.

A key mechanisms to achieve predictability in Dahlia is its time-sensitive affine type system. Consumable hardware resources, such as memory ports, are modelled in a way that they can be used at most once (affine) at a time (time-sensitive). Any temporally conflicting access to these resources must be avoided in the program code.
Even though the authors strongly argue against using a ‘legacy software language’, their tool still compiles to HLS C++ code. As mentioned above, it improves the predictability of HLS but the ‘results can still vary’ depending on what the black-box HLS tool decides [NAT+20].

**Discussion**

The above mentioned approaches are witness to the fact that there are general flaws in HLS with regards to predictability [NAT+20], portability [OPGM+20] and separation of concerns [LCH+19] (e.g., algorithm and optimisations). Furthermore, [JKLL20] mentions bugs and unsupported features. In comparison to hand-written HDL code, HLS approaches deliver lower performance [SWLR13, WBC13] and attain lower clock frequencies due to timing issues [GLC+20]. In [SWLR13], a hardware expert achieves 61% of the performance with one third of the effort using HLS, compared to manual coding in a HDL. A remaining performance gap of $3.8 \times$ for HLS approaches compared to hand written code is stated in [WBC13].

Shir takes a fundamentally different path. First of all, the problems emanating from legacy software languages are dodged by using a functional IR. As discussed before in Chapter 1, functional programming is better suited to describe hardware systems, due to their commonalities. In addition, Shir’s compiler targets low-level RTL code to become independent from the monolithic, closed-source HLS tools once and for all. To separate algorithmic specification, optimisations and hardware-specific aspects, Shir features a multi-level IR. It raises the entry-point programming abstraction to a hardware-agnostic level, usable for non-hardware experts. Device-specific and vendor-specific details are introduced as late as possible, on the lower levels of the multi-level IR. Each of the levels exposes different hardware details and enables rewrite rules to automatically optimise the design. This optimisation process is transparent and the developers can comprehend how a certain implementation was created. Finally, Shir is able to deliver competitive performance despite its high-level abstraction, as the evaluation of this thesis shows.

## 3.4 Multi-Level Representations

Compilers can quickly become complex and unmanageable. To simplify their design, the compilation flow can be broken down into a series of more manageable lowering steps through multiple IRs. A structure like this is extensible and makes individual
parts, e.g., optimisations, reusable among various frameworks, reducing the development costs of compilers.

LLHD [SKGB20] is a three-level IR spanning over the entire low-level design flow of digital circuits. This project aims at providing a foundation for novel HDLs, moving away from the monolithic, vendor-specific, proprietary implementations of System-Verilog and VHDL language standards. LLHD is able to capture behavioural, structural and netlist descriptions and comes with the corresponding lowering passes between these IRs. The work in this thesis addresses a higher level part of the accelerator design process. Shir currently targets VHDL but future work can interface with other LLHD-based back ends.

Recently, MLIR [LAB+20, LAB+21] has been initiated as a subproject of the LLVM compilation framework [LA04]. It provides a unifying compiler infrastructure with multiple IRs on different abstractions levels to simplify compiler development. To create a new IR for this infrastructure, a custom dialect with its own set of operations and types must be defined. An IR is lowered by applying transformations in Multi-Level Intermediate Representation (MLIR) that turn one dialect into another one. There already are a couple of MLIR dialects, for instance an LLVM IR to generate CPU and GPU code, an IR for TensorFlow [ABC+16] graphs, an HLO (High Level Operations) IR as input for the XLA (Accelerated Linear Algebra) compiler, an IR for structured control flow, an affine dialect for polyhedral optimisations and a ‘linalg’ dialect for operations on tensors and buffers.

**Discussion**

Shir’s infrastructure shares many ideas of MLIR. Similar to MLIR’s tensors, Shir models high-level data in multidimensional arrays. Both compilers lower these into more hardware-related types like vectors for SIMD tasks or ramarrays (memrefs in MLIR) for buffering. The development of Shir started before MLIR has gained traction. The MLIR infrastructure was therefore not available as basis of Shir’s IRs. Nevertheless, the contributions in this theses could as well be ported to MLIR by adding new dialects and lowering passes to the framework. The work [LSS21] has proven the viability of ‘integrating a functional pattern-based IR into MLIR’. The project is based on the functional language Rise [HLK+20], a spiritual successor of Lift [SFLD15], with common data-parallel patterns.

In the end, building on the MLIR infrastructure remains an implementation detail, orthogonal to the novel concepts introduced by Shir.
3.5 Domain-Specific Synthesis

This section presents HLS frameworks for applications from a particular domain only.

Spiral [PMJ+05, MFHP12, SP19] is a domain-specific hardware generation framework for linear signal processing transforms. It presents a high-level mathematical formalism to represent the input algorithm. By rewriting or transforming it, the design space is automatically explored and the algorithm is finally compiled into synthesizable Verilog code for FPGAs and ASICs. Similar to Shir, it supports a similar notion of stream-based data to enable pipelining and it applies rewrite rules for optimisation.

There are numerous domain-specific approaches to accelerate image processing on FPGAs, for instance Darkroom [HBD+14], Rigel [HDD+16] and topically HalideHLS [PBY+17]. The latter is based on the popular image processing programming language Halide [RKBA+13]. It presents an end-to-end system from domain-specific Halide code to FPGAs, extending the available two target platforms CPU and GPU with a third one. Apart from the bitstream, this approach also generates the software program to interface with the accelerator. HeteroHalide [LCC20] shares the main idea but offers a more up-to-date implementation with an improvement in throughput. For the compilation, it employs HeteroCL [LCH+19], previously mentioned in Section 3.3, as an IR to generate the hardware designs for the different back ends.

Discussion

The principal difference of these approaches in comparison to Shir is that they are not extensible. The integration of the latest optimisations for novel algorithms is complicated. On the other side, Shir aims for a domain-extensible IR to support high-performance accelerators for various workloads. Its multi-level IR and the rewriting concept facilitate the extension with new abstractions and rewrite optimisations, as well as back ends for a wider range of hardware platforms. As the authors of [KS21] have demonstrated, domain-extensible approaches even manage to outperform domain-specific ones, when provided with additional optimisations in the form of composed rewrite rules.
3.6 Higher-Level Synthesis

This section details the most closely related work, which starts at a higher level of abstraction than HDLs and traditional HLS approaches, is not limited to a specific application domain and is independent of black-box HLS toolchains.

Glenside [SLL+21] introduces a pure functional IR with access patterns to express data layout transformations. This addresses the identified gap in current machine learning IRs that term rewriting only works well on pure functional high-level representations. The authors claim that low-level, typically impure, memory usage and optimisations, such as operator fusion, hamper rewriting techniques. In fact, low-level IRs and rewrite rules for optimisations function well together, as shown by SHIR. Also hardware-level memory details can be extensively expressed in a pure functional IR. SHIR’s multi-level approach furthermore cleanly separates the rewrite optimisations on different levels of abstraction.

Glenside circumvents the issues of implicit repetitions by avoiding name binding, as in lambda abstractions, and higher-order functions altogether. This, however, prevents the programmer from using well-known functional patterns, e.g., Map, and puts the additional task of choosing the right access patterns on the programmer’s shoulders. Moreover, this requires the definition of specialised operators like cartProd (similar to SkelCL’s allpairs [SFAG14]), reduceSum and dotProd in the Glenside IR. In contrast, the IRs in SHIR support lambdas and higher order functions, which enable the programmer to specify arbitrary programs on any dimensional data in a familiar way with only a few common functional primitives. Operators, such as the dot product, can be created on the fly by combining generic Map, Mul, Reduce and Add primitives. No additional specialised operator is needed. Furthermore, Glenside does not generate hardware designs but rewrites the program to match a predefined architecture, not exploiting the flexibility of the FPGA. They neither discuss how expensive operations like transposition could be achieved in hardware, nor show any performance results.

The advantages of a strongly typed functional IR for hardware design have recently been shown by TyTra, Lift-hls and Aetherling. Similar to SHIR’s vector and stream types, they feature space- and time-aware types to express parallelism and pipelining on a high level of abstraction. Moreover, they raise the abstraction level for the programmer and offer a hardware-agnostic entry point. Rewrite rules optimise their hardware implementations.
TyTra [NV15, NV16, NV19, VNU19] presents a compiler framework that automatically turns a given application into an optimised hardware design. To explore the space of FPGA-specific optimisations, it applies type transformations, hence the name TyTra. These transformations produce program variants with different performance and resource utilisation.

TyTra contains a functional coordination language, TyTra-CL, that is used to describe the netlist of a given input program [VNU19]. It includes declarations of functions and input vectors, while their implementation is not required. TyTra-CL is a subset of Haskell but it uses dependent vector types. It offers common higher-order functions, such as \texttt{map} and \texttt{fold}. This raises the abstraction level for the programmer or other high-level language compilers that target TyTra. The work in [NV19] particularly contributes an automated flow to compile legacy Fortran code into an optimised FPGA-based accelerator, where the implementation is automatically pipelined and vectorised.

TyTra’s types are capable of expressing various design variants with different kinds of parallelism. For example, the functionality of a \texttt{map} primitive can be interpreted in three different ways, with a parallel, sequential or pipelined implementation. Similarly, a reduction operation is either implemented sequentially or as a tree. The resulting designs of TyTra’s transformations are correct by construction and provably implement the identical functionality.

In order to estimate the area utilisation and throughput of the generated variants, they are first compiled from TyTra-CL to the TyTra-IR, a strongly and statically typed intermediate language. This IR is based on the LLVM IR and contains extensions for FPGA-specific details. Computation is described using Static Single Assignments (SSAs). A cost model now parses and analyses the compiled TyTra-IR variants to obtain quick performance estimates before synthesis. Code generators, such as the TyTra Back-End Compiler (TyBEC), emit Verilog code for hardware synthesis from the best design variant.

The applications of TyTra focus on streaming programs and originate from the scientific computing domain, such as numerical weather prediction and climate simulation. In contrast to Shir, algorithms, such as matrix multiplication and convolution, are not evaluated, leaving open how the data reshaping challenges therein are tackled.

Lift-hls [KBSD19] targets FPGAs from the data-parallel, functional language Lift [SFLD15]. Its evaluation focuses on matrix multiplication. This application brings many challenges, as shown and addressed in Chapter 6. To circumvent the data repetition issue of functional IRs, Lift-hls implements a work-around for matrix mul-
tiplication, but does not address this issue generally. It cannot repeat data without buffering it in on-chip memory and, thus, cannot implement a low-area matrix multiplication version, where the input data is directly streamed from host RAM only, for example. In contrast, **SHIR** provides a generic solution for data reshaping based on automated rewrite rules. It reaches more different points in the design space of matrix multiplication, as the evaluation in Section 7.3 will show. Furthermore, **SHIR** is able to transpose and slide data, as needed for tiling and convolution, which is not supported by Lift-hls at all.

Some limited support for memory operations is provided by Lift-hls. To move data to and from the FPGA, it relies on an ad-hoc approach and does also not discuss how this is achieved. Based on the LetStream primitive on-chip buffering can be realised, however, there is no way to express what kind of memory is used and how resources like memory are shared among multiple clients. In contrast to **SHIR**, Lift-hls does not automatically identify data reusage in the IR and insert buffers where needed.

The authors of this related work do not discuss how to *automatically* vectorise the dot product and how to scale the memory width accordingly, although this is anything but trivial as the many rewrite steps in Sections 6.1 and 6.3 indicate. Furthermore, tiling is achieved by the host only, whereas **SHIR** accumulates the tiles on the FPGA itself in a reduction of a 2D stream. The vectorisation in Lift-hls remains limited but reasonable, considering the size of the FPGA. However, on more powerful devices, a higher level of parallelism is possible, which in turn brings its own set of problems. A large reduction tree, for example, causes long propagation delays, too long to be handled in a single clock cycle, so that pipelining is required. As we will see in Section 6.7, an optimisation in **SHIR** addresses this issue automatically to enable a parallel dot product on huge vectors.

All in all, despite the many commonalities, **SHIR** is more sophisticated and flexible then Lift-hls with regards to handling memory and optimising the implementation. Performance issues are more generically addressed in **SHIR**. The set of evaluated algorithms is larger and is run on a more powerful but also more complex Intel Arria 10 FPGA, which is found in industry use-cases.

**Aetherling** [DFH+20] generates hardware designs with different throughput-area trade-offs by choosing between sequences over space (SSeq) and over time (TSeq). This approach starts with a hardware-agnostic data-parallel input language, which is transformed into their space-time-aware IR using rewrite rules. Within this step the design is either fully parallelised or fully sequentialised. For partially-parallel
implementations, an additional set of rewrite rules is involved, which operates on the input language. That way, hardware design decisions are encoded in the input specification. In contrast, SHIR’s algorithmic representation does not contain any information about parallelisation. These details are only introduced at the lower level IRs, that are made for this very purpose.

Aetherling builds on statically-scheduled hardware modules and does not support variable-latency operators and therefore any kind of asynchronous or blocking communication. However, these are necessary when targeting a real FPGA, because the timing of the DMA interface to exchange data between host memory and accelerator is non-deterministic.

Similar to SHIR, this approach inserts registers into signal paths with long propagation delays to maintain high clock rates. Nevertheless, the authors do not discuss how exactly this is achieved.

Aetherling has presented results on a larger set of applications compared to SHIR. However, they have only demonstrated results in simulation using a simple memory model and have not discussed how memory would be handled on a real hardware platform. In contrast, this thesis evaluates matrix multiplication and 2D convolution, which bring some major challenges, for example, the repetition and transposition of streams. Moreover, all the designs produced are run on a real FPGA.

**Delite** [SBL+14] is a compiler framework that aims at simplifying the development of Scala-embedded DSLs. Existing DSLs include OptiML for machine learning, OptiQL for data querying, OptiGraph for graph analysis and OptiMesh for mesh computation. Delite provides reusable common components such as functional parallel patterns, optimisations and code generators, as well as constructs to define custom IRs. The output of the compilation is C++, OpenCL or related program code, which inherits HLS-related issues, as mentioned above in Section 3.3.

Nevertheless, [GLN+14] extends Delite by an automated methodology ranging from high-level programs in DSL to FPGA bitstreams. Given an input program based on parallel computation patterns, a hardware design with multiple kernels is generated. These are integrated into a fixed system architecture template, which dynamically schedules them and dynamically manages the system’s memories. The template hides the memory implementation, which prevents memory-specific optimisations. This is unlike SHIR, which exposes them in its IR and is able to optimise them by applying high-level rewrite rules.

The scheduling of the generated kernels is controlled by a soft-core processor,
which occupies additional logic on the FPGA. This contrasts with SHIR, which relies on a static schedule, so that no such controller is required. The architecture template used in [GLN+14] furthermore contains a globally shared on-chip buffer for all the kernels, which creates a von Neumann-style bottleneck. Apart from that, an architecture with one central, large on-chip memory undermines the advantages of the block RAM’s distributed nature. This may lead to routing problems for larger designs.

Further extensions of Delite are presented in [KDP+16, PKB+16]. Their FPGA back end emits optimised MaxJ [Tec11] code, a low-level Java-based hardware generation language for FPGAs. The work [PKB+16] in particular introduces a compiler with transformation rules to automatically pipeline and parallelise operations or to insert on-chip buffers for locality. The extension in [KDP+16] allows to lower the high-level parallel constructs from Delite into Delite Hardware Definition Language (DHDL), an embedded domain-specific language in Scala. DHDL exposes details about parallelism, locality and memory in an internal dataflow graph IR. This graph is analysed to provide quick area and throughput estimates. Each node in the dataflow graph represents a certain functionality, which is associated to a parameterisable architectural template. These templates implement basic operations, memory access and controllers for pipelining and parallelisation. The approach is evaluated on multiple algorithms including the dot product and matrix multiplication.

Nevertheless, the work [KDP+16] differs from SHIR in the following aspects. Delite has more coarse-grained hardware templates. However, finer building blocks can be more flexibly composed to realise different implementations of the algorithm. For example, to load and store tiles in Delite, the primitives TileLd and TileSt are used, while SHIR achieves this behaviour based on the combination of more standard primitives Map, Read and Write. Small templates for more fundamental operations enable more generic rewrite optimisations. In case there was a rewrite rule to optimise all Map implementations, it would also affect the tiling in SHIR. However, in Delite, the TileLd primitive in Delite would require an additional, specialised rewrite rule.

Spatial [KFP+18] ties in with Delite in that it provides a higher level of abstraction for performance-oriented hardware accelerator design. However, it is more tailored to improve the productivity of hardware developers instead of automating the entire hardware design flow all the way from the algorithm. The language of Spatial is concise and contains hardware-specific abstractions, including interfacing with the host machine and memory management through a library of on-chip and off-chip memory templates.
3.6. Higher-Level Synthesis

The compiler of Spatial generates optimised Chisel [BVR+12] code, ready to be synthesised for various FPGA architectures of different vendors but also the Plasticine [PZK+17] Coarse-Grained Reconfigurable Array (CGRA). During this compilation, the implementation is automatically pipelined and memory access is parallelised by using multiple banks for example. The design space is automatically explored by the machine learning framework HyperMapper [BNZ+16].

In comparison to approaches based on Delite and Spatial, SHIR is more transparent, exposing various hardware details on different levels of abstraction in a multi-level IR. The functional foundation of these IRs facilitates employing rewrite rules for optimisation. The type system of SHIR exposes fundamental design decisions by distinguishing between stream, vector and ramarray types, which Delite and Spatial do not offer. Furthermore, SHIR presents a mechanical lowering procedure, while Delite does not discuss how arbitrary hardware-agnostic algorithms are mapped to its coarse-grained hardware templates.

Summary

As shown in Figure 1.1b in the introduction, the workloads for accelerators are rapidly evolving. The number of publications in this research area is growing exponentially. It is not surprising that several contributions evince similarities with SHIR. Some of the related work, such as Aetherling [DFH+20], was even developed at the same time. They share the same goal to improve the productivity of accelerator design with higher-level (often functional) approaches.

Besides many similarities, there are also differences to SHIR. Previous related work comes short of providing at least one of the following features:

**Number of IR Levels** A variety of IRs enabling rewrite rules (or similar mechanisms) to optimise the design on different levels of abstraction.

**Hardware-Agnostic IR** A hardware-agnostic entry point for the user or front end, making the framework accessible for non-hardware experts or tools.

**Stream- & Vector-Like Types** A representation of streams and vectors (or a similar division of types) to express pipelining and parallelism on the type level.

**Memory-Aware IR** A representation capable of expressing access to the diverse memories found on common FPGA platforms.
Table 3.1: Features of Shir, the framework developed in this thesis, and closely related work.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Number of IR Levels</th>
<th>HW-Agnostic IR</th>
<th>Stm &amp; Vec Types</th>
<th>Memory-Aware IR</th>
<th>Stream Repetition</th>
<th>Transpose &amp; Slide2D</th>
<th>Stm-Vec Conversions</th>
<th>FPGA Evaluation</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shir</td>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Glenside</td>
<td>1</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>TyTra</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Lift-hls</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Aetherling</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Delite</td>
<td>1</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Spatial</td>
<td>1</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>— manual —</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Optimisations** Automatic rewrite rules to parallelise the design, insert buffers and address the data reshaping issues present in functional IRs.

**FPGA Evaluation** An evaluation using a real FPGA device taking physical effects, such as signal propagation delay, into account proving that the approach is viable. This includes the implementation of ways to communicate with the host machine, e.g., via DMA.

**Applications** A set of evaluated algorithms that involves data reshaping and requires the above mentioned optimisations for decent performance.

Table 3.1 provides a feature table that shows the limitations of state-of-the-art related approaches for higher-level synthesis.

As can be seen, Glenside [SLL+21] has a single IR for the hardware-agnostic description of the algorithm. To generate an implementation for matrix multiplication, which is not run on an FPGA, it introduces a dedicated primitive to circumvent the challenge of creating the cartesian product of streams, but does not solve it generically. Instead of naively computing convolutions, it maps them to matrix multiplications only.
3.6. Higher-Level Synthesis

TyTra [NV15] has a sophisticated type system to express pipelining and parallelism. The compiler targets a real FPGA. However, the workloads from SHIR are not supported.

Lift-hls [KBSD19] introduces a two-level structure with one hardware-agnostic IR for the algorithm and another one closer to the hardware. It manages to implement matrix multiplication on an FPGA with a non-generic work-around for the data repetition. The challenges around transposition or sliding windows are left untouched, which is why convolution cannot be realised.

Similarly, Aetherling [DFH+20] comes with space and time data types and an analogous division into two abstraction levels. It has not support for repetition of streams, transposition and is therefore unable to encode matrix multiplication or 2D convolution. Its use-cases are centred around much simpler one-dimensional convolution, which are, however, only run in simulation.

Delite [KDP+16] provides none of the data reshaping optimisations but has some limited representation of memory in its IR. It compiles matrix multiplication for FPGA but does not evaluate convolutions.

Spatial [KFP+18] is able to express data reshaping operations, but since there are no hardware-agnostic primitives, it is up to the user to perform all the optimisations manually. Identically, memory operations are made available in the IR but they require hardware expertise to handle them efficiently. Spatial already includes key performance decisions at its highest abstraction level, where SHIR only focuses on the algorithm itself.
Part II

Contributions
Chapter 4

Multi-Level Representations

The main goal of this thesis is to produce optimised FPGA code from an input specification in a high-level functional language. First of all, this requires the definition of such a language but also an automated compilation process with optimisations to generate the desired hardware design.

As motivated in Chapter 1, a multi-level IR approach simplifies the compiler design by applying the principle of separation of concerns. Each IR has a certain function in the compiler flow. Depending on its abstraction level, different features are exposed. The input algorithm is specified on a very high-level hardware-agnostic abstraction, where no hardware design decisions have yet been made. Only the lower level abstractions gradually introduce hardware concepts, such as pipeline or spatial parallelism, buffering strategies and how to exploit special hardwired functions on the target platform. Once new features are exposed in an IR, they can be used and optimised for the given use-case. The functional design of the IRs enables easy transformations, for lowering and optimisation purposes, using simple rewrites. The output after the compilation is VHDL code suitable for FPGA synthesis. This kind of low-level representation provides the flexibility needed and enables the compiler to specify all the details in the hardware design.

In a more extensive accelerator design process, the input specification of the compilation can be provided by a high-level framework or a tool that maps existing legacy code to parallel patterns, as in [GRS+18]. The algorithmic specification then acts as an intermediate language and is therefore also considered to be an IR.

In this thesis, the overall framework with its IRs and the compiler are referred to as Shir. The compiler flow through Shir’s multi-level IRs is depicted in Figure 4.1. Each of these IRs is individually presented in this chapter.
Figure 4.1: IRs in the Shir compiler from algorithmic code to VHDL code. The complex compilation is split into many simple and more manageable steps (solid edges), that will be shown in Chapter 5. The first IRs employ the functional programming paradigm, where rewrite-based optimisations (dashed self-loop edges) are easily supported. Chapter 6 will explain these optimisations.

First, Section 4.1 gives an overview of how a simple example program is represented on Shir’s various levels of abstraction. Then, the Shir core language is introduced in Section 4.2, which all of the IRs in the Shir compiler are based on. After that, the individual IRs are presented in Sections 4.3 to 4.7, going from high-level abstractions to lower level ones. Section 4.8 deals with dataflow graphs, a more hardware-like representation, and another IR for actual VHDL code generation. Section 4.9 summarises the design choices and benefits of Shir’s multi-level IR structure.

4.1 Overview

Before diving into the IRs, a brief example illustrates how a simple program ‘sum of an array’ is represented at the first three abstraction levels.

At the Algorithmic Level, programs are represented in a hardware-agnostic way using common built-ins such as Map, Reduce or Zip. They can be generated from external tools or manually written by a programmer. Here, the sum of an array is as in expression (4.1), where Input<\text{IntT}(32), N> simply represents an input array of \( N \) 32-bit integers which will be allocated in memory.

When the input algorithm is provided to the compiler, it is first lowered to the Architecture Level, where architecture design decisions are made. These affect the performance, for instance, if the level of parallelisation is increased. More specifically, the Abstract Memory Level exposes memory operations. The ‘sum of an array’ example on this level is shown in expression (4.2).
4.1. Overview

\[ \text{Input}<\text{IntT}(32), N> \]
\[ \Rightarrow \text{Reduce}(\text{Add}, \text{ConstInt}<0>) \]

(a) Algorithmic Level.

\[ \text{Let } \text{mem} = \text{MemAlloc}<\text{HostRamT}, \text{IntT}(32), N> \text{ in} \]
\[ \text{Counter}<0, 1, N> \]
\[ \Rightarrow \text{MapStm}(\text{Read}(\text{mem})) \]
\[ \Rightarrow \text{ReduceStm}(\text{Add}, \text{ConstInt}<0>) \]

(b) Abstract Memory Level.

\[ \text{Let } \text{rdCtrl} = \lambda r. r \Rightarrow \text{ReadHostMemCtrl} \text{ in} \]
\[ \text{Counter}<0, 1, N/16> \]
\[ \Rightarrow \text{ReadAsync}(\text{rdCtrl}, \text{ConstInt}<0>) \]
\[ \Rightarrow \text{MapStm}(\text{SplitVec}<32> \Rightarrow \text{VecToStm}) \]
\[ \Rightarrow \text{JoinStm} \]
\[ \Rightarrow \text{ReduceStm}(\text{Add}, \text{ConstInt}<0>) \]

(c) Hardware Memory Level.

Figure 4.2: Representations at three different levels for the program ‘sum of an array’.
As a first step in expression (4.2), host RAM for the program’s input is allocated. A counter generates a stream of addresses from 0 to $N-1$. Then, a sequential implementation of map, namely $\text{MapStm}$, is used to call the $\text{Read}$ memory function for each address. This function reads the memory $\text{mem}$ at the given indices. The $\text{MapStm}$ outputs a stream of data that is reduced to produce the sum. For the sake of simplicity, the details about the program’s output are omitted in this example, but a similar process takes place where explicit write operations are performed. While no specific hardware details are yet exposed in the IR, memory operations are now explicit. This abstraction makes it easy to introduce memory usage into the design and enables simple memory-related optimisations, as Section 6.2 will show.

The Hardware Memory Level is a lower IR level and exposes all the hardware features directly, as shown in expression (4.3), allowing an easy translation to VHDL code. During lowering, a $\text{ReadHostMemCtrl}$ function is inserted in place of the abstract $\text{MemAlloc}$ concept. This read controller is directly connected to the DMA engine on the FPGA. The $\text{ReadAsync}$ expression takes the read controller and the base address $\text{ConstInt<0>}$ of the input data allocation in memory to process the incoming stream of offsets produced by the counter. The static value for the base address is determined by the compiler in a separate step.

Compared to the previous level, the Hardware Memory Level has two major differences: First, $\text{ReadAsync}$ asynchronously requests data from the hardware DMA controller. As explained in background Section 2.1.2, this controller responds at a later time with the desired data. Since the controller accepts multiple pending requests, an asynchronous read takes a stream of offsets as an input, which are requested concurrently.

Second, as is the case with many accelerator devices, the granularity of a transfer between host and device has a fixed size. This can, for example, be a 512-bit wide vector, a cache line of the host’s CPU. To read $N$ 32-bit values, for example, only $N/512$ cache lines are needed. The $\text{ReadAsync}$ primitive returns them in a stream. As seen in expression (4.3), this data is reshaped with a $\text{SplitVec}$ into pieces of 32 bits. Finally, a $\text{JoinStm}$ is used to merge the two outer dimensions of the stream and produce a single stream of 32-bit integers that are fed into the reduction.

This example has shown how different levels of abstraction for the same program expose different hardware details. While higher levels are more concise and easier to (re)write, they only offer limited control about the resulting hardware implementation. Once a certain higher IR is optimised, it is lowered to the next representation, to make
more hardware details accessible, which are then again optimised. That way, the compilation and optimisation processes are kept as simple as possible, because they operate only on the features that are necessarily exposed in a specific IR. For instance, due to this multi-level IR design, the programmer does not have to deal with low-level concepts like cache lines, when defining the algorithm or when a simple buffering strategy is employed.

4.2 Core Abstract Language

The implementation of the Shir framework is embedded in Scala \[OAC^{+}04\]. This specific programming language is not required but it provides modern features like pattern matching that help implementing compiler passes and rewriting mechanisms, which are needed in Chapters 5 and 6.

The core of the Shir language and the high-level algorithmic primitives are inspired by Lift \[SFLD15\], a functional, data-parallel language based on typed lambda calculus. The major difference is the support of subtyping, in the style of System $F$ with subtyping, System $F_{<:}$ \[CMMS91\], as shown in Section 2.2.2. This allows to constraint the use of data types and their composition. As will be explained later in Sections 4.4.2, 4.4.3 and 4.5, certain nestings of types must be forbidden. Such restrictions are easy to implement via subtyping. Furthermore, this mechanism enables building up new constructions within the language, while the unified type system confirms that the implementation’s types are correct.

Core Types

All types in Shir are implemented as classes in Scala. Their constructors are referred to as type constructors. The object oriented manner of the type implementations facilitates extensions to new types, without requiring any modification of the core type system. When dealing with subtyping relationship, the type constructors’ arguments are all covariant except for function types, where it is contravariant in its input type (see Section 2.2.2).

Figure 4.3 shows the Shir core type hierarchy. The root of the hierarchy is AnyT. Classes implementing this trait represent any type. To denote a more specific type, a subclass of this trait must be implemented instead. These subclasses are separated into value types and meta types, which are explained below.
Data and function types both are value types. Hence, they are taken as input to, or returned by, functions. Common data types, like integer, represent the main values for the computation, i.e., data at runtime. These specific types are not part of the core language and only introduced in the IRs in Sections 4.3 and 4.4.

The function type constructor is $\text{FunT}(\text{inT}, \text{outT})$ where $\text{inT}$ is the function input type and $\text{outT}$ its output type. The superscript denotes a subtyping relationship, i.e., $\text{inT}$ must be a subtype of $\text{ValueT}$. From now on in this thesis, the function type is represented as: $\text{inT} \rightarrow \text{outT}$.

To represent meta-information embedded inside other types, the meta type is used. In the core language, the only concrete meta type is $\text{NatT}$, the type representing natural numbers. As we will see in the following sections, this type is used to specify the length of arrays (or other collections). Unlike value types, meta types have static values that are known during compile time.

Generic (or templated) functions are implemented using type-level functions. The type-function type constructor takes a type variable $TV$, subtype of $T$, and a return type $U$ where the type variable can appear: $\text{TypeFunT}(TV, U)$. The corresponding short form notation is $TVT \rightarrow U$.

Going beyond defining functions with specific types, type-functions allow to generalise these to work with a variety of types. Instead of defining a concrete function that, for instance, takes an integer and returns an integer, $\text{integer} \rightarrow \text{integer}$, a more generic function can now be expressed: $TV^{DataT} \rightarrow TV \rightarrow TV$. In plain English this means: Given a type variable $TV$, which is a subtype of $DataT$, a function is obtained that takes something of type $TV$ and returns something of the same type.
4.2. Core Abstract Language

Figure 4.4 shows the grammar for the core expressions of the Shir language. These expressions Param, Lambda, FunCall and Let are traditionally found in any implementation of lambda calculus. Lambda is an anonymous function that has a Param as argument and an Expr as body where Param can appear. FunCall is simply a function call to a lambda. Let binds an expression to a Param that is shared among all of the Param’s appearances in the body.

For example, the lambda abstraction \( \lambda x . x \) implements the identity — what goes in, comes out. The \( x \) is the parameter’s identifier. Once a function is defined, whether it is anonymous or not, it can be called. With \( arg \) as the argument of such a function call, \( (\lambda x . x)(arg) \) is created. In case a lambda is called, beta reduction is applied directly (in the construction of the expression), which replaces all occurrences of the lambda’s parameter in the body with the argument expression of the function call. After this reduction, the function call and the lambda are removed from the expression without changing the semantics. For the above example this process simply returns \( arg \).

To create generic expressions, the TypeLambda is used. Instead of specific types, type variables are placed in the body of the TypeLambda. This allows the types to be instantiated later using a TypeFunCall with a specific type. When a TypeLambda and
a `TypeFunCall` collide in an expression, the type lambda’s type variable is substituted by the call’s argument type, similar to the above mentioned reduction.

To exemplify this, the identity abstraction is brought forth again, this time with annotated types (after the colon): \( \lambda x : \text{IntT}(32) . \ x : \text{IntT}(32) \). This expression is restricted to 32-bit integer types but a type lambda makes it generic: \( \Lambda TV . \ \lambda x : TV . \ x : TV \). Now, a type-function call instantiates this generic function with any desired type, e.g., an 8-bit integer: \( (\Lambda TV . \ \lambda x : TV . \ x : TV)<\text{IntT}(8)> \). The obtained function in turn can then be called with an integer value \( i \): \( (\Lambda TV . \ \lambda x : TV . \ x : TV)<\text{IntT}(8)> (i) \).

As defined in the grammar, only a single input parameter is defined per function. Nevertheless, functions with multiple parameters can be imitated by either taking a tuple as the input or returning another function, that requires the second argument, after the first argument was provided (currying). In order to keep function specifications concise, a smart constructor automatically generates the currying. This way, the developer has the comfort of creating and calling functions with multiple parameters, e.g., \( f(\text{arg1}, \text{arg2}, ...) \), while behind the scenes a nesting of single parameter functions do the job. Likewise, this simplification is applied to type-functions with multiple type variables and their calls.

The `Primitive IR` node represents built-in function calls or built-in constants. They are treated similarly to generic lambda abstractions but are represented only by an identifier, e.g., `Id` for the identity function and `Add` for addition. Just like in the identity example above, primitives are first instantiated with a specific type and are then called with the input argument: `Id<\text{IntT}(8)> (i)`.

Since the argument of the type-function call usually corresponds to the type of the argument, e.g., `Id<\text{type of } i> (i)`, the trivial type-function call can be omitted for tidier expressions: `Id (i)`.

Furthermore, to avoid (reverse) composition and nested functions that are hard to read, the `reverse function application` as in OCaml [MM22] (also `pipe-forward operator`) is defined for the `\( \triangleright \)` symbol. For example, the following expression first applies \( f_1 \), then \( f_2 \) and finally \( f_3 \) to the input \( i \): \( i \triangleright f_1 \triangleright f_2 \triangleright f_3 \). If read using the English standard of left to right, this notation indicates the flow of the input data through the functions, which is particularly suitable for the description of hardware circuits.

In `Shir`, each expression has a (return) type. When building expressions, their types do not have to be explicitly provided, a type variable (`TypeVarT`) can be used instead of a concrete type. Again, the superscript notation indicates whether a type variable must be a subtype of another type.
The type checker for SHIR uses unification and a constraint solver, where subtyping relationships are expressed as constraints. During type checking, type variables will eventually be replaced by a concrete type, if the program is correctly typed. If the type system rules are violated, an error is thrown.

For example, given the expression \((\lambda x: TV1. x: TV2)(i: IntT(32))\), the type checker determines that \(IntT\) must be a subtype of \(TV1\) due to the contravariant relation of argument type of the call and input type of the function. In addition, \(TV1\) and \(TV2\) must be equal because they refer to the same parameter.

4.3 Algorithmic Level

This section presents the high-level algorithmic extensions to the core language. It provides a hardware-agnostic entry point to the SHIR compiler and can be handwritten or targeted from other compilers and frameworks.

The focus of this Algorithmic Level lies on the what, rather than the how things are implemented. This frees up the programmer from having to worry about the underlying hardware specifics to achieve performance. Instead, the compiler is responsible for providing performance, given a high-level input program.

Types

The high-level algorithmic types are shown in Figure 4.5, outside of the dashed box. The array type stores both the element type \(T\) and the array length \(N\), as seen in its type constructor \(ArrayT(T^{DataT}, N^{NatT})\). From now on, the shortcut syntax \([T]_N\) is used...
to represent array types. For tuple types, *TupleT*, the short form is \(<\text{type}_1,\text{type}_2,\ldots>\). Float types in compliance with the IEEE standard 754 [IEE08] are supported in SHiR.

In contrast to related languages, such as Lift [SFLD15] and Rise [HLK+20], the integer type stores the number of bits used to represent the integer value. The allowed bit-widths are not limited to powers of two, the SHiR compiler supports arbitrary precision. The integer type constructor reflects this flexibility: *IntT*(\text{numBits}^{\text{NatT}}). This enables efficient area usage especially on FPGAs, because they are re-configurable and not constrained by a specific operator bit-width.

**Primitives**

The algorithmic IR primitives, listed in Figure 4.6, are common high-level functional primitives. They act as a shortcut to a specific combination of SHiR’s core expressions (e.g., function, type-function, function call, ...), which determines the primitive’s (return) type. For addition, subtraction, multiplication, division and identity, the primitives \text{Add}, \text{Sub}, \text{Mul}, \text{Div} and \text{Id} are used. The \text{ConstInt} primitive takes a static \text{NatT} that specifies its constant output value. \text{Tuple} builds a tuple from two separate input expressions of types \text{T}_0 and \text{T}_1. \text{Select} takes a static natural number \text{N} to extract the element at this index from a given tuple.

The \text{Map} primitive requires a function, \(T \rightarrow U\), to process the incoming array \([T]_N\). Similarly, \text{Reduce} takes a function, this time with two separate inputs \(U \rightarrow (T \rightarrow U)\), an initial value \(U\) and then an array \([T]_N\) to produce a single value \(U\). To keep the expressions for higher-order functions tidy, a short form is introduced. Instead of \text{Map}(\lambda x. f(x))\), the lambda can be omitted: \text{Map}(f).

The \text{Slide} primitive implements a sliding window of size \(S_W\), which is moved in steps of \(S_S\) over the input array \([T]_N\). At each step the window is emitted, so that a nested array of arrays \([T]_{S_W}(N-S_W)/S_S+1\) is created. This is similar to the stencil operation in prior works of Lift [HSS+18, SHS+19].

\text{Zip} transforms two collections of elements \([T]_N\) and \([U]_N\) into a single one with pairs. For this, the ingoing arrays must have the same length. To split an array into chunks of size \(M\), the \text{Split} primitive is used. The length of the original array must be a multiple of \(M\), otherwise the type checker will issue an error message. The opposite operation is \text{Join}, which flattens a given array, effectively merging two dimensions into a single one.

The \text{Input} primitive represents the input data coming from memory. It supports multidimensional data, specified by \(D_1\ldots D_N\).
Add: $T^{\text{ScalarT}} \rightarrow \langle T, T \rangle \rightarrow T$ \hfill (4.4)

Sub: $T^{\text{ScalarT}} \rightarrow \langle T, T \rangle \rightarrow T$ \hfill (4.5)

Mul: $T^{\text{ScalarT}} \rightarrow \langle T, T \rangle \rightarrow T$ \hfill (4.6)

Div: $T^{\text{ScalarT}} \rightarrow \langle T, T \rangle \rightarrow T$ \hfill (4.7)

Id: $T^{\text{DataT}} \rightarrow T \rightarrow T$ \hfill (4.8)

ConstInt: $N^{\text{NatT}} \rightarrow T^{\text{IntT}} \rightarrow T$ \hfill (4.9)

Tuple: $T_0^{\text{DataT}} \rightarrow T_1^{\text{DataT}} \rightarrow T_0 \rightarrow T_1 \rightarrow \langle T_0, T_1 \rangle$ \hfill (4.10)

Select: $N^{\text{NatT}} \rightarrow T_0^{\text{DataT}} \rightarrow T_1^{\text{DataT}} \rightarrow \langle T_0, T_1 \rangle \rightarrow T_N$ \hfill (4.11)

Map: $T^{\text{DataT}} \rightarrow U^{\text{DataT}} \rightarrow N^{\text{NatT}} \rightarrow (T \rightarrow U) \rightarrow [T]_N \rightarrow [U]_N$ \hfill (4.12)

Reduce: $T^{\text{DataT}} \rightarrow U^{\text{DataT}} \rightarrow N^{\text{NatT}} \rightarrow$

\hspace{2cm} $(U \rightarrow (T \rightarrow U)) \rightarrow U \rightarrow [T]_N \rightarrow U$

Slide: $S_W^{\text{NatT}} \rightarrow S_S^{\text{NatT}} \rightarrow T^{\text{DataT}} \rightarrow N^{\text{NatT}} \rightarrow$

\hspace{2cm} $[T]_N \rightarrow [[T]_{S_W}]_{(N-S_W)/S_S+1}$ \hfill (4.14)

Zip: $T^{\text{DataT}} \rightarrow U^{\text{DataT}} \rightarrow N^{\text{NatT}} \rightarrow \langle [T]_N, [U]_N \rangle \rightarrow [(T, U)]_N$ \hfill (4.15)

Split: $M^{\text{NatT}} \rightarrow T^{\text{DataT}} \rightarrow N^{\text{NatT}} \rightarrow [T]_N \rightarrow [[T]_M]_{N/M}$ \hfill (4.16)

Join: $T^{\text{DataT}} \rightarrow M^{\text{NatT}} \rightarrow N^{\text{NatT}} \rightarrow [[T]_M]_N \rightarrow [T]_{M \cdot N}$ \hfill (4.17)

Input: $T^{\text{DataT}} \rightarrow D_1^{\text{NatT}} \rightarrow \ldots \rightarrow D_N^{\text{NatT}} \rightarrow [[T]_{D_1}, \ldots]_{D_N}$ \hfill (4.18)

Figure 4.6: Algorithmic primitives and their types. These definitions are read from left to right: First, the type variables ($T$ and $U$) are specified for generic primitives. $N$, $M$, $S$ and $D$ represent natural numbers, while $D_1$ to $D_N$ denote dimensions. They are concatenated with the $\rightarrow$ symbol and have a superscripted super type, e.g., $T^{\text{ScalarT}}$ means that $T$ is a subtype of $\text{ScalarT}$. After that, the input value types are listed, separated by a $\rightarrow$ symbol. The rightmost value type is the return type of the primitive.
Figure 4.7: Streamed data (left) is passed element by element, over time, through function \( f \) of \( \text{MapStm} \). \( \text{MapVec} \) creates multiple instances of \( f \) to process a vector (right) in parallel at the cost of more area on the FPGA.

### 4.4 Architecture Level

Producing high-performance hardware from high-level algorithmic primitives is challenging. The compiler must make many choices regarding parallelisation and buffering strategies. To address this challenge, it is typical to use further IR levels to encode implementation choices.

The Architecture Level introduces a notion of timing with its new collection types, explained in the following sections. For instance, choices about whether computation is run in parallel (over \( \text{space} \)) or sequential (over \( \text{time} \)) are encoded directly in the IR. Figure 4.7 shows these two cases using the example of \( \text{Map} \) primitives. For example, TyTra \([15, 16, 19]\), Lift-hls \([19]\) and Aetherling \([DFH+20]\) are approaches that similarly distinguish between two major collection types. On top of that \( \text{SHIR} \) provides a more sophisticated type hierarchy to restrict nesting of types and introduces a third novel collection type for memory, as will be described in Section 4.5.

The Architecture Level remains functional, facilitating the application of transformations that crucially affect the resulting hardware design and performance without needing to operate on a low HDL-like level. Figure 4.8 shows the core language types, this time extended with all the architecture types.
4.4.1 Scalars and Tuples

On the Architecture Level, each type has a bit width allowing to statically analyse the memory usage and other characteristics of a design. Scalar types are similar to the Algorithmic Level, with the addition of LogicT, which represents a single bit. Subtypes of BasicDataT ensure that data is always available in a single clock cycle. This contrasts with streams and ramarrays, which require multiple cycles to read their entire content, as we will see in the following sections.

The tuple type manifests itself in two variants with different restrictions for its element types. The TupleT type is flexible and allows to contain streams, tuples and basic data. The BasicTupleT is more restrictive and only accepts immediately accessible elements of type BasicDataT. Whenever the Tuple primitive is created, the return type is determined by its elements and is either TupleT or BasicTupleT.

Primitives that operate on scalars (Add, Sub, Mul, Id, ConstInt), as well as Select are similar to the ones in Figure 4.6. Additionally, a bitwise shift operator is introduced:

\[
\text{Shift}: R^{\text{LogicT}} \rightarrow N^{\text{NatT}} \rightarrow T^{\text{IntT}} \rightarrow T \rightarrow T
\]  

(4.19)

It shifts to the right, if the logic value \( R \) is set to 1, otherwise to the left. The static number \( N \) determines by how many positions the bits are moved. The Architecture Level, however, has no division primitive (\( \text{Div} \)), since it is not necessarily supported by the targeted hardware platform. Section 5.1 will explain how this problem can be circumvented in some cases by using \( \text{Shift} \) instead.
4.4.2 Stream

Type

The stream type models flow of data in general and pipelining. A new element is produced at each clock cycle unless the stream is stalled. Once data has been consumed, it cannot be recalled. The type constructor \( \text{Stream}T(\text{NonRamArray}T, N\text{Nat}T) \) represents a sequence of \( N \) elements of type \( T \), where \( T \) can be any data type but a ramarray. This is because ramarrays represent data in memory, which must be read first in order to operate on it. From now on, the short-form notation \( \mathbb{Z}[T]_N \) is used for stream types.

Primitives

The high-level algorithmic primitives are refined for streams. For example, \( \text{MapStm} \), \( \text{ReduceStm} \), \( \text{SlideStm} \), \( \text{ZipStm} \), \( \text{SplitStm} \) and \( \text{JoinStm} \) all operate on streams, as the suffix \( \text{Stm} \) indicates. These new primitives are listed in Figure 4.9, where more details about their type signature is provided. In the following, some implementation details of stream-based primitives are revealed.

\( \text{MapStm} \), for instance, instantiates the given function and feeds the individual elements of the stream into it, one at a time. Depending on the function, this generates a pipeline in hardware. \( \text{ReduceStm} \) generates an accumulator, which accumulates all incoming elements of the stream using a given function (e.g., \( \text{Add} \)). This flexible primitive will also be used later to write a stream of data to memory. \( \text{SlideStm} \) is implemented in hardware by feeding the input stream into a shift register and sending out all the register’s contents as a vector, whenever a new input value arrives. This process is illustrated in Figure 4.10.

\( \text{DropStm} \) removes the first \( F \) and the last \( L \) elements from a given stream. \( \text{ConcatStm} \) takes two streams and combines them into one long stream of \( N+M \) elements. Both input streams must have the same element type.

\( \text{StmToVec} \) converts a stream of \( N \) basic data elements into a vector. In the hardware implementation, a shift register is filled element by element from the stream. Once it is full, all its contents are emitted in parallel as a vector. This behaviour is similar to the \( \text{SlideStm} \) implementation in Figure 4.10 with a fixed window size of \( N \) and a step size of \( N \), except that the returned vector is not wrapped in a stream.

\( \text{Repeat} \) takes basic data or a stream and repeats it \( N \) times. This primitive plays a vital role later for the optimisation in Section 6.4.1.
4.4. Architecture Level

MapStm: $\mathcal{T}^{\text{NonRamArray}} \rightarrow U^{\text{NonRamArray}} \rightarrow N^{\text{Nat}} \rightarrow$

\[
(T \rightarrow U) \rightarrow \mathcal{E}[T]_N \rightarrow \mathcal{E}[U]_N
\]

ReduceStm: $\mathcal{T}^{\text{NonRamArray}} \rightarrow U^{\text{Data}} \rightarrow N^{\text{Nat}} \rightarrow$

\[
(U \rightarrow (T \rightarrow U)) \rightarrow \mathcal{E}[T]_N \rightarrow U \rightarrow U
\]

SlideStm: $S_W^{\text{Nat}} \rightarrow S_S^{\text{Nat}} \rightarrow T^{\text{BasicData}} \rightarrow N^{\text{Nat}} \rightarrow$

\[
\mathcal{E}[T]_N \rightarrow \mathcal{E}[\mathcal{E}[T]_{S_W}(N-S_W)/S_S+1]
\]

ZipStm: $\mathcal{T}^{\text{NonRamArray}} \rightarrow U^{\text{NonRamArray}} \rightarrow N^{\text{Nat}} \rightarrow$

\[
\langle \mathcal{E}[T]_N, \mathcal{E}[U]_N \rangle \rightarrow \mathcal{E}[(T,U)]_N
\]

SplitStm: $M^{\text{Nat}} \rightarrow \mathcal{T}^{\text{NonRamArray}} \rightarrow N^{\text{Nat}} \rightarrow \mathcal{E}[T]_N \rightarrow \mathcal{E}[\mathcal{E}[T]_{M/N}]_{M/N}$

\[
(4.24)
\]

JoinStm: $\mathcal{T}^{\text{NonRamArray}} \rightarrow M^{\text{Nat}} \rightarrow N^{\text{Nat}} \rightarrow \mathcal{E}[\mathcal{E}[T]_{M}]_N \rightarrow \mathcal{E}[T]_{M\cdot N}$

\[
(4.25)
\]

DropStm: $\mathcal{T}^{\text{NonRamArray}} \rightarrow N^{\text{Nat}} \rightarrow F^{\text{Nat}} \rightarrow L^{\text{Nat}} \rightarrow$

\[
\mathcal{E}[T]_N \rightarrow \mathcal{E}[T]_{N-F-L}
\]

ConcatStm: $\mathcal{T}^{\text{NonRamArray}} \rightarrow N^{\text{Nat}} \rightarrow M^{\text{Nat}} \rightarrow$

\[
\langle \mathcal{E}[T]_{M}, \mathcal{E}[T]_N \rangle \rightarrow \mathcal{E}[T]_{M+N}
\]

StmToVec: $\mathcal{T}^{\text{BasicData}} \rightarrow N^{\text{Nat}} \rightarrow \mathcal{E}[T]_N \rightarrow \mathcal{E}[T]_N$

\[
(4.28)
\]

Repeat: $N^{\text{Nat}} \rightarrow \mathcal{T}^{\text{NonRamArray}} \rightarrow T \rightarrow \mathcal{E}[T]_N$

\[
(4.29)
\]

Counter: $C_0^{\text{Nat}} \rightarrow C_S^{\text{Nat}} \rightarrow D_1^{\text{Nat}} \rightarrow \ldots \rightarrow D_N^{\text{Nat}} \rightarrow T^{\text{Nat}} \rightarrow$

\[
\mathcal{E}[\ldots \mathcal{E}[T]_{D_1}, \ldots]_{D_N}
\]

\[
(4.30)
\]

Figure 4.9: Architecture Level primitives operating on streams.
Figure 4.10: Stream-based slide with window size $S_W$ of 2 and step size $S_S$ of 1. The incoming stream is fed into the internal shift register, first into $r_2$, then $r_1$. Once filled, its contents are sent out as a vector. Then, $S_S$ more elements are consumed from the input and shifted into the registers, before producing the next output vector. The blue dotted lines show which input data creates which output vector for the three steps.

Counter emits a stream of incrementing integers, starting at $C_0$ with step size $C_S$. This primitive is useful for generating memory addresses as seen in the preceding example of Section 4.1. To keep the FPGA resource usage low, the compiler allocates just enough bits for the counter’s integer data type to express the specified value range. As seen in the type definition in expression (4.30), multiple dimensions $D_1$ to $D_N$ can be defined for the Counter. The output type then becomes a stream of streams (or more deeply nested), creating an $N$-dimensional structure of incrementing integers. This feature helps when certain sequences of values must be repeated, as needed later in Section 6.4.1.

4.4.3 Vector

Type

The vector type is similar to the tuple type, with the key difference that the vector’s elements must all be of the same type. A vector is ideal for spatial parallelism, a common strategy to improve performance. The type constructor $\text{VectorType}(T^{\text{BasicData}}, N^{\text{Nat}})$ creates a vector type with support for parallel access to all its $N$ elements of type $T$ in a single clock cycle. This property limits the elements of vectors to those that are instantaneously accessible (basic data). From now on, the short-form notation $\text{VEC}[T]^N$ is used to represent vector types.
4.4. Architecture Level

MapVec: $T^{BasicDataT} \mapsto U^{BasicDataT} \mapsto N^{NatT} \mapsto$

$(T \mapsto U) \mapsto \mathbb{B}[T]_N \mapsto \mathbb{B}[U]_N$

(4.31)

SlideVec: $S_W^{NatT} \mapsto S_S^{NatT} \mapsto T^{BasicDataT} \mapsto N^{NatT} \mapsto$

$\mathbb{B}[T]_N \mapsto \mathbb{B}[\mathbb{B}[T]_{S_W}]_{(N-S_S)/S_S+1}$

(4.32)

ZipVec: $T^{BasicDataT} \mapsto U^{BasicDataT} \mapsto N^{NatT} \mapsto$

$\langle \mathbb{B}[T]_N, \mathbb{B}[U]_N \rangle \mapsto \mathbb{B}[\langle T, U \rangle]_N$

(4.33)

SplitVec: $M^{NatT} \mapsto T^{BasicDataT} \mapsto N^{NatT} \mapsto \mathbb{B}[T]_N \mapsto \mathbb{B}[\mathbb{B}[T]_{M/N}]_{N/M}$

(4.34)

JoinVec: $T^{BasicDataT} \mapsto M^{NatT} \mapsto N^{NatT} \mapsto \mathbb{B}[\mathbb{B}[T]_{M}]_N \mapsto \mathbb{B}[T]_{M\cdot N}$

(4.35)

DropVec: $T^{BasicDataT} \mapsto N^{NatT} \mapsto F^{NatT} \mapsto L^{NatT} \mapsto$

$\mathbb{B}[T]_N \mapsto \mathbb{B}[T]_{N-F-L}$

(4.36)

ConcatVec: $T^{BasicDataT} \mapsto N^{NatT} \mapsto M^{NatT} \mapsto$

$\langle \mathbb{B}[T]_M, \mathbb{B}[T]_N \rangle \mapsto \mathbb{B}[T]_{M+N}$

(4.37)

VecToStm: $T^{BasicDataT} \mapsto N^{NatT} \mapsto \mathbb{B}[T]_N \mapsto \mathbb{B}[T]_N$

(4.38)

VecToTuple: $T^{BasicDataT} \mapsto N^{NatT} \mapsto \mathbb{B}[T]_N \mapsto \langle T, \ldots, T \rangle$

(4.39)

TupleToVec: $T^{BasicDataT} \mapsto N^{NatT} \mapsto \langle T, \ldots, T \rangle \mapsto \mathbb{B}[T]_N$

(4.40)

Figure 4.11: Architecture Level primitives operating on vectors.
**Primitives**

The common high-level algorithmic primitives also have a counterpart for vectors, which end on the suffix *Vec*: e.g., *MapVec*, *SlideVec*, *ZipVec*, *SplitVec* and *JoinVec*, as listed in Figure 4.11.

*MapVec* exploits spatial parallelism by instantiating the given function once for each vector element. *SlideVec* simply generates wires in hardware to rearrange the data to create a vector of vectors. This logic operation involves no states.

*DropVec* removes the first $F$ and last $L$ elements of a vector, while *ConcatVec* concatenates two vectors. Additionally, the *VecToStm* primitive converts a vector into a stream of data, by emitting the vector’s elements one after the other. *VecToTuple* converts a vector into a tuple to feed the data into tuple-based operations like *Add*. The *TupleToVec* primitive turns a tuple, where all the elements have the same type, into a vector.

There is no explicit primitive for *ReduceVec*. Instead, the same functionality is achieved with a smart constructor of the same name, which automatically generates an efficient reduction tree from a combination of *MapVec* and *VecToTuple*. The tree performs $N-1$ operations in $\log_2 N$ steps. Given an *input* with a vector type of eight elements and an *Add* function, this constructor will return the following expression:

\[
\text{input} \triangleright \text{SplitVec}<2> \triangleright \text{MapVec} (\text{VecToTuple} \triangleright \text{Add}) \\
\triangleright \text{SplitVec}<2> \triangleright \text{MapVec} (\text{VecToTuple} \triangleright \text{Add}) \\
\triangleright \text{VecToTuple} \triangleright \text{Add} \quad // \text{reducing the remaining two elements}
\]

The reduction operator in a reduction tree must be associative in order to compute the correct result. The current implementation of SHIR leaves it up to the user to ensure that this is adhered to. Future extensions can help the compiler prevent generating erroneous hardware designs, by adding information about the associativity of functions to the IR.
4.5 Abstract Memory Level

Memory plays a key role in achieving high-performance hardware designs. Thus, a compiler for accelerators must be able to express diverse ways of buffering and memory usage in general.

This section introduces the Abstract Memory Level for interfacing with memory. With this new level of abstraction, the IR becomes memory-aware, enabling optimisations around memory, as will be demonstrated later in Section 6.2.

Types

The type constructor \( RamArrayT(T^{BasicDataT}, N^{NatT}, ML^{MemLocT}) \) represents a ramarray of \( N \) elements of type \( T \), which can only be accessed one at a time, however, random-order access is possible. From now on, the short-form notation \( \frac{\alpha}{\beta}[T]_N^{ML} \) is used for ramarray types.

As seen in the type definition, this collection type cannot contain any other elements than basic data types, i.e., scalars, basic tuples or vectors, because the elements must always be read or written as a whole within a single clock cycle. Data structures like streams for example would not fulfil this requirement as they take multiple cycles to be transmitted. Shir features a flat memory model and its type system forbids nesting \( RamArrayT \). Multi-dimensional data is stored in memory by joining (i.e., flattening) the data before writing to memory and then splitting it after reading.
Let expressions are not shown for simplicity.

The ramarray’s MemLocT indicates which of the memories in the FPGA system, see Figure 2.2, is chosen to hold the data. The available options are host RAM (HostRamT), off-chip onboard DDR RAM (OnboardRamT) or on-chip block RAM (BlockRamT). Since the onboard RAM is partitioned into banks, an identifier is provided with the type constructor, e.g., OnboardRamT(0) for the first bank. An FPGA chip can realise many different block RAMs in various shapes (width and depth). The block RAM type therefore contains an identifier to distinguish between multiple block RAM instances. The host RAM type, on the other side, is a singleton, because there is only one such memory in an FPGA system.

**Primitives**

The primitives on this Abstract Memory Level, shown in Figure 4.12, enable allocation of memory, reading from and writing to it on a high level of abstraction, similar to other programming languages.

MemAlloc allocates memory space for \( N \) elements of type \( T \) and returns a ramarray. A MemLocT is required, to identify the memory used. In the following example, 1024 8-bit integers are allocated in host memory:

\[
\text{MemAlloc}\langle\text{HostRamT},\text{IntT}(8),1024\rangle
\]

Given a ramarray, the Read primitive returns an element at the specified address of type \( A \). The bit-width of the address depends on the ramarray’s length and is equal to \( \lceil \log_2 N \rceil \).

The Write primitive enables the reverse operation: writing a new element at a given position in a ramarray. Similar to the concept of monads in functional programming [Wad95], this primitive returns the updated ramarray. However, the later generated hardware implementation updates the data in-place.
Thus, an interesting code pattern is enabled: By wrapping Write into ReduceStm, an entire stream can be buffered in a ramarray. In practice, this is beneficial when the ramarray resides on-chip and the stream is read multiple times. The following expression implements this important use-case, where the ingoing stream input is stored in a ramarray and then read again:

\[
\text{Let } \text{allocatedBuffer} = \text{MemAlloc<memLocType, dataType, N> in (4.46)}
\]

\[
\text{Let } \text{updatedBuffer} = (\text{input}, \text{Counter<0, 1, N>}) \triangleright \text{ZipStm}
\]

\[
\triangleright \text{ReduceStm}(\lambda \text{mem. \lambda data. data} \triangleright \text{Write(mem), allocatedBuffer}) \text{ in Counter<0, 1, N> \triangleright MapStm(\lambda addr. addr} \triangleright \text{Read(updatedBuffer))}
\]

Figure 4.13 illustrates the corresponding block diagram. The compiler does not expect the user to write such code. Instead, it will be inserted automatically as part of the optimisations presented in Section 6.2.

In case multiple onboard memory banks are available to the system, data can be stored in interleaved memory. The data is distributed alternately on the banks, improving the throughput when accessing consecutive addresses. An onboard RAM buffer with interleaving is expresses as follows in SHIR:

\[
\text{Let } s = \text{input} \triangleright \text{SplitStm<2> \triangleright MapStm(StmToVec \triangleright VecToTuple) in (4.47)}
\]

\[
\text{Let } \text{obram0} = \text{MemAlloc<OnboardRamT(0), ..., N/2> in}
\]

\[
\text{Let } \text{obram1} = \text{MemAlloc<OnboardRamT(1), ..., N/2> in}
\]

\[
\text{Let } \text{buffered0} = \langle s \triangleright \text{MapStm(Select<0>), Counter<0, 1, N/2>}\rangle
\]

\[
\triangleright \text{ZipStm} \triangleright \text{ReduceStm}(\lambda \text{mem. \lambda data. data} \triangleright \text{Write(mem), obram0}) \text{ in Counter<0, 1, N/2> \triangleright MapStm(\lambda addr. addr} \triangleright \text{Read(buffered0)},
\]

\[
\text{Let } \text{buffered1} = \langle s \triangleright \text{MapStm(Select<1>), Counter<0, 1, N/2>}\rangle
\]

\[
\triangleright \text{ZipStm} \triangleright \text{ReduceStm}(\lambda \text{mem. \lambda data. data} \triangleright \text{Write(mem), obram1}) \text{ in Counter<0, 1, N/2> \triangleright MapStm(\lambda addr. addr} \triangleright \text{Read(buffered1)},
\]

\[
\langle \text{Counter<0, 1, N/2> \triangleright MapStm(\lambda addr. addr} \triangleright \text{Read(buffered0)}),
\]

\[
\text{JoinStm}
\]

The input stream of N elements is first reshaped into a stream s of tuples. Then, memory regions on two onboard RAM banks are allocated. The first half of the tuple stream s is extracted using MapStm(Select<0>) and then written to the first memory bank. Likewise, the other bank is filled with the second half of the tuple. After that, both buffers are read and the resulting streams are reshaped into the original format.
Table 4.1: Mapping of low-level memory features to existing functional concepts.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Functional Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>function</td>
</tr>
<tr>
<td>memory access</td>
<td>function call</td>
</tr>
<tr>
<td>shared memory</td>
<td>shared function</td>
</tr>
</tbody>
</table>

4.6 Hardware Memory Level

The Abstract Memory Level from the previous section allows to easily interface with memory. However, this IR cannot be realised in hardware due to abstract concepts like ramarray types. The Abstract Memory Level must therefore be lowered to a more feasible implementation, which is expressed on the Hardware Memory Level.

There are similarities between memory access and functional concepts are shown in Table 4.1. For example, a reading access can be viewed as calling a memory function with an input address, to obtain the data at this address as the return value. Reusing existing functional concepts for interaction with memory simplifies the implementation of memory features in the compiler.

Types

At the Hardware Memory Level, memory is represented as functions and the ramarray types disappear. Depending on the type of memory involved, on-chip versus off-chip, these functions are either synchronous (data is returned immediately) or asynchronous (requested data is returned later and not necessarily in order). To capture this in the type system, the IR is expanded with a new function type. Its type constructor is $ArchFunT(inT^{ValueT}, outT^{ValueT}, c^{CommT})$. The short-form notations $inT \xrightarrow{s} outT$ for synchronous communication (where $c$ is $SyncT$) and $inT \xrightarrow{a} outT$ for asynchronous communication ($c$ is $AsyncT$) will be used from now on.
4.6. Hardware Memory Level

\[ \text{BlockRam}: T_{\text{BasicData}} \rightarrow A_{\text{Int}} \rightarrow WE_{\text{Logic}} \rightarrow \langle T, A, WE \rangle \xrightarrow{s} T \] (4.48)

\[ \text{ReadSyncMemCtrl}: T_{\text{BasicData}} \rightarrow A_{\text{Int}} \rightarrow WE_{\text{Logic}} \rightarrow \langle T, A, WE \rangle \xrightarrow{s} T \] (4.49)

\[ \langle \langle T, A, WE \rangle \xrightarrow{s} T \rangle \xrightarrow{s} A \xrightarrow{s} T \]

\[ \text{WriteSyncMemCtrl}: T_{\text{BasicData}} \rightarrow A_{\text{Int}} \rightarrow WE_{\text{Logic}} \rightarrow \langle T, A \rangle \xrightarrow{s} T \] (4.50)

\[ \langle \langle T, A \rangle \xrightarrow{s} T \rangle \xrightarrow{s} A \xrightarrow{s} \langle T, A \rangle \xrightarrow{s} T \]

\[ \text{ReadSync}: T_{\text{BasicData}} \rightarrow A_{\text{Int}} \rightarrow \langle T, A \rangle \xrightarrow{s} T \] (4.51)

\[ \langle A \xrightarrow{s} T \rangle \xrightarrow{s} A \xrightarrow{s} \langle T, A \rangle \xrightarrow{s} T \]

\[ \text{WriteSync}: T_{\text{BasicData}} \rightarrow A_{\text{Int}} \rightarrow \langle T, A \rangle \xrightarrow{s} T \] (4.52)

\[ \langle \langle T, A \rangle \xrightarrow{s} T \rangle \xrightarrow{s} A \xrightarrow{s} \langle T, A \rangle \xrightarrow{s} T \]

\[ \text{OnboardRam}: B_{\text{Nat}} \rightarrow T_{\text{BasicData}} \rightarrow A_{\text{Int}} \rightarrow R_{\text{Int}} \rightarrow WE_{\text{Logic}} \rightarrow \langle T, A, R, WE \rangle \xrightarrow{s} \langle T, R \rangle \] (4.53)

\[ \langle \langle T, A, R, WE \rangle \xrightarrow{a} \langle T, R \rangle \rangle \xrightarrow{s} \langle A, R \rangle \xrightarrow{a} \langle T, R \rangle \]

\[ \text{ReadOnboardMemCtrl}: T_{\text{BasicData}} \rightarrow A_{\text{Int}} \rightarrow R_{\text{Int}} \rightarrow WE_{\text{Logic}} \rightarrow \langle T, A, R, WE \rangle \xrightarrow{a} \langle T, R \rangle \] (4.54)

\[ \langle \langle T, A, R, WE \rangle \xrightarrow{a} \langle T, R \rangle \rangle \xrightarrow{s} \langle A, R \rangle \xrightarrow{a} \langle T, R \rangle \]

\[ \text{WriteOnboardMemCtrl}: T_{\text{BasicData}} \rightarrow A_{\text{Int}} \rightarrow R_{\text{Int}} \rightarrow WE_{\text{Logic}} \rightarrow \langle T, A, R \rangle \xrightarrow{a} R \] (4.55)

\[ \langle \langle T, A, R, WE \rangle \xrightarrow{a} \langle T, R \rangle \rangle \xrightarrow{s} \langle A, T, R \rangle \xrightarrow{a} R \]

\[ \text{ReadHostMemCtrl}: T_{\text{BasicData}} \rightarrow A_{\text{Int}} \rightarrow R_{\text{Int}} \rightarrow \langle A, R \rangle \xrightarrow{a} \langle T, R \rangle \] (4.56)

\[ \text{WriteHostMemCtrl}: T_{\text{BasicData}} \rightarrow A_{\text{Int}} \rightarrow R_{\text{Int}} \rightarrow \langle A, T, R \rangle \xrightarrow{a} R \] (4.57)

\[ \text{ReadAsync}: T_{\text{BasicData}} \rightarrow N_{\text{Nat}} \rightarrow A_{\text{Int}} \rightarrow R_{\text{Int}} \rightarrow \langle A, R \rangle \xrightarrow{a} \langle T, R \rangle \] (4.58)

\[ \langle \langle A, R \rangle \xrightarrow{a} \langle T, R \rangle \rangle \xrightarrow{s} A \xrightarrow{s} [A]_N \xrightarrow{s} [T]_N \]

\[ \text{WriteAsync}: T_{\text{BasicData}} \rightarrow N_{\text{Nat}} \rightarrow A_{\text{Int}} \rightarrow R_{\text{Int}} \rightarrow \langle A, T, R \rangle \xrightarrow{a} A \] (4.59)

\[ \langle \langle A, T, R \rangle \xrightarrow{a} R \rangle \xrightarrow{s} A \xrightarrow{s} [\langle T, A \rangle]_N \xrightarrow{s} A \]

Figure 4.14: Memory-related primitives on the Hardware Memory Level. Types named \( A \) denote integer-based types for addresses, and \( B \) memory bank ids. Types \( R \) represent integer-based types for request ids, which map the responses in asynchronous communication to the previously sent requests.
Figure 4.15: Block diagram on the Hardware Memory Level for buffering a stream $s$ in block RAM. The block RAM function is shared among the two controllers. Due to the synchronous fashion of block RAM access, the controllers are connected to the ReadSync and WriteSync primitives. In this memory representation, the MemAlloc (from the Abstract Memory Level) is replaced by a ConstInt, that provides the base address of the allocated memory region. Blue boxes are primitives and memory related ones are dashed. Let expressions are not shown for simplicity.

Primitives for Synchronous Block RAM

At this level, there is no abstract MemAlloc primitive but a more specialised BlockRam primitive, shown in Figure 4.14, to model synchronous on-chip RAM as functions. When called, this primitive returns a function that accepts a piece of data, an address and a write enable flag which chooses between read and write access. This functional design is in line with its resulting hardware implementation.

Additionally, memory controllers are introduced, which extract only a certain capability of the memory interface. The ReadSyncMemCtrl primitive takes a BlockRam and provides an interface which only allows to read data. The write enable flag is not accessible any longer from the outside and is fixed to false internally. The WriteSyncMemCtrl extracts a write interface, setting the write enable to true internally. The provided interface from WriteSyncMemCtrl takes a value and an address and returns the same value after writing.

ReadSync uses the ReadSyncMemCtrl to read a value at a certain address from the block RAM memory, which is behind the memory controller. This primitive calculates the actual requested address by adding the given base address and given offset, both subtypes of integer. The WriteSync primitive works in a similar way, but instead takes a tuple of data and offset as input.
The block diagram for buffering a stream on this lower level of memory representation is shown in Figure 4.15. The SHR code is as follows:

\[
\text{Let } \text{mem} = \lambda \text{data} \cdot \lambda \text{addr} \cdot \lambda \text{we} \cdot \text{data} \triangleright \text{addr} \triangleright \text{we} \triangleright \text{BlockRam in} \quad (4.60)
\]

\[
\text{Let } \text{readCtrl} = \lambda \text{addr} \cdot \text{addr} \triangleright \text{ReadSyncMemCtrl}(\text{mem}) \text{ in}
\]

\[
\text{Let } \text{writeCtrl} = \lambda \text{dataAddr} \cdot \text{dataAddr} \triangleright \text{WriteSyncMemCtrl}(\text{mem}) \text{ in}
\]

\[
\text{Let } \text{updatedBaseAddr} =
\quad \langle \text{input}, \text{Counter}<0,1,N> \rangle \triangleright \text{ZipStm} \triangleright \text{ReduceStm}(\lambda \text{ba} \cdot \lambda \text{data} \cdot \text{data} \triangleright \text{WriteSync}(\text{writeCtrl}, \text{ba}), \text{ConstInt}<0>) \text{ in}
\]

\[
\text{Counter}<0,1,N> \triangleright \text{MapStm}(\lambda \text{addr} \cdot \text{addr} \triangleright \text{ReadSync}(\text{readCtrl}, \text{updatedBaseAddr}))
\]

**Primitives for Asynchronous Host RAM**

On the Hardware Memory Level, onboard RAM banks are represented as functions through the \text{OnboardRam} primitive. Its constructor takes a natural number (\(B\) in the definition) to identify which memory bank is used. Similar to block RAM, a pair of memory controllers, \text{ReadOnboardMemCtrl} and \text{WriteOnboardMemCtrl}, extracts the read or write capabilities of this function by setting the write enable flag correspondingly.

From the FPGA point of view, the notion of host RAM is always present in the system and there is no need to allocate it. Therefore, there is no counterpart to the \text{BlockRam} or \text{OnboardRam} primitive on the Hardware Memory Level. However, the \text{ReadHostMemCtrl} and \text{WriteOnboardMemCtrl} primitives provide memory controllers for accessing host memory. During hardware generation, these controllers are connected to the DMA interface of the FPGA which talks to the host RAM via the PCIe bus. This method of communication is explained in Section 2.1.2 in more detail.

Each asynchronous memory access must specify a unique request id (\(R\) in the definition) to link a response to the requested read or write operation. The asynchronous read controllers, \text{ReadOnboardMemCtrl} and \text{ReadHostMemCtrl}, provide an asynchronous function, which takes a tuple of address and request id and returns a tuple of data and request id. Likewise, the write controllers, \text{WriteOnboardMemCtrl} and \text{WriteHostMemCtrl}, provide an asynchronous function, that takes a tuple of address, data and request id and just returns the same request id, when the data has been written to the address in memory.
Chapter 4. Multi-Level Representations

The hardwired interfaces to memory depend on the FPGA DMA specifications. The Intel Arria 10, used in this thesis, transfers cache lines of 512 bits between the FPGA chip and host RAM or a bank of onboard RAM. Smaller pieces of data must be packed and possibly padded with zeros to fill an entire line. For 7-bit values each cache line contains 73 elements \(\left\lfloor \frac{512}{7} \right\rfloor \) with 1 padding bit \((512 \mod 7)\). As will be introduced in Section 5.1, an automatic conversion generator in ShIR is able to create a combination of splits, joins and other reshaping primitives to convert cache lines into any desired type of data with arbitrary precision and back.

The primitives for reading and writing are refined to enable concurrent memory requests. They take the corresponding memory controller and a base address as input. In addition, \texttt{ReadAsync} consumes a stream of addresses, while \texttt{WriteAsync} consumes a stream of tuples of address and data. Without waiting for the responses of previous requests (non-blocking), these asynchronous primitives send new requests to memory one by one, as soon as a new value from the input stream arrives. Thus, throughput between the host and the FPGA is maximised using multiple in-flight requests. The input stream’s length of \texttt{ReadAsync} and \texttt{WriteAsync} determines the maximum number of possible parallel requests. Once the requested data is loaded, the host sends a response with the payload attached. \texttt{WriteAsync} returns the base address, when all the write operations to this memory region are completed and confirmed by the memory controller.

While this IR design looks complicated on the surface, it directly maps to hardware concepts and provides great flexibility in terms of expressible hardware designs.

4.7 Device-Specific Level

The flexible, programmable logic is a key component that all FPGAs have in common. However, many devices also offer specialised functions in hardened gates, as described in Section 2.1.1. These excel at a specific purpose and are up to \(10 \times\) more power efficient \([\text{OVP}^{+} 22]\) than soft implementations on programmable logic. The Intel Arria 10, for instance, contains dedicated, hardwired transceivers for efficient DMA via PCIe and off-chip RAM access. Furthermore, more than 1500 DSPs perform efficient integer and floating point multiplications. These dedicated hardware functions optimise area, speed and power consumption of essential computation and data movement, while the FPGA’s programmable logic remains available for application-specific functions.
In order to exploit such device-specific features, the Shir compiler flow provides an additional abstraction at the lower level end of its functional IRs. To represent specialised functions on this Device-Specific Level, new primitives are introduced, like Mul2Add for example:

\[
\text{Mul2Add} : \mathbb{T}\mathbb{IntT} \rightarrow \langle \langle T, T \rangle, \langle T, T \rangle \rangle \rightarrow T
\]  

(4.61)

It leverages a configuration of Intel’s DSPs, where two lower precision multiplications are performed simultaneously and the two results are summed up in a single DSP block. Two tuples of integers are consumed to produce a single output value. This increases the number of possible parallel multiplications in a single clock cycle by using the DSPs more efficiently.

The Device-Specific Level serves as a general entry point for all optimisations that only apply to the targeted FPGA. It is easily extensible with new device-specific primitives, for instance to exploit other specialised hard IPs and better support systolic arrays.

### 4.8 Hardware Back End

Even from the lowest functional level in Shir, a few lowering steps are still necessary to finally generate HDL code. This process does not involve any more rewrite optimisations, because all the design decisions have already been made on the previous, higher levels of abstraction. Now, the functional world is left behind to reach more hardware-like representations. To simplify the lowering, it is divided into a dataflow and a VHDL IR, both described in this section.

#### 4.8.1 Dataflow Level

On the Dataflow Level, a Shir program is represented as a directed graph with producer-consumer patterns, similar to many traditional models of computation, such as SDF [LM87]. It consists of nodes, to represent operations on data, and directed edges to model the flow of data between nodes. The nodes can contain subgraphs to create a node hierarchy. This helps to organise the overall structure of the design by grouping certain parts of the program into a node, similar to what a human hardware designer would program in an HDL. Moreover, neatly generated hardware code facilitates debugging. Their visual appearance of dataflow graphs is similar to the previously encountered block diagrams in Figures 4.13 and 4.15 for example.
Chapter 4. Multi-Level Representations

Figure 4.16: Outgoing and ingoing handshake ports for bidirectional synchronisation. Each port is composed of four signals: data, last, valid and ready. They are directed towards the consumer, except for the ready signal, which is send by the consumer to notify the producer that the current data has been received.

Special behaviour nodes are defined for the related built-in primitives of the Architecture Level. These nodes have a fixed behaviour that specifies how the ingoing data is processed to produce the output.

To model communication between nodes, edges connect an outgoing port of one node to an ingoing port of another one. Each port has an Architecture Level type assigned, to provide information about what kind of data is transmitted. Only ports with opposing directions and with the same type can be connected via edges. Ports can be composed to form more complex ports.

Most commonly used in Shir’s dataflow graphs is the composed handshake port, which implements a simple handshake protocol, similar to [KBSD19]. As visualised in Figure 4.16, a handshake port consists of multiple signals: The payload data, a signal for the last flags and common handshaking signals to determine whether the data produced is valid and whether the consumer is ready to receive data. The last signal indicates when a stream’s last element, which in turn can also be a stream, is being transmitted. In case of multi-dimensional streams, this signal consists of multiple bits, one for each dimension. If only basic data, such as vectors, tuples or scalars, is exchanged, this signal is not in use.

According to the handshake protocol, data flows between components only when the valid and ready signals are both active. That way, the communication between
nodes is locally synchronised, resulting in a dynamic scheduling. There is no centralised scheduler, the control over interactions between nodes is distributed. A dynamic schedule supports asynchronous features such as host RAM access. While this requires some additional control logic, no negative impact was observed on the overall throughput or latency, because the control signals are only 1 bit wide and the control logic remains very simple. However, in the future, classical dataflow analysis can be used to create static schedules, as demonstrated in [CJC+20]. This is not necessary in Shir, because the scheduling does not pose a performance problem and satisfactory results are achieved with fully dynamic designs, as the evaluation will show.

In general, the dataflow follows a data push approach, where producers feed their outputs to their consumers, instead of consumers querying for their input data. As soon as a node has valid inputs, it processes the data to generate a result.

Besides the handshake port, there are other composed ports: request ports and response ports deal with asynchronous communications. A peculiarity of this is that the response protocol does not include a ready signal. This design decision is based on the concept of how asynchronous memories work in hardware. Therefore, a node must only make a request, when it is ready to receive the response, i.e., when it can directly process or buffer the incoming payload. Otherwise the response might get lost. The hardware templates, as will be introduced shortly, of nodes with such ports are carefully designed to ensure this behaviour, e.g., by adding an internal buffer.

In order to connect the generated design to the FPGA’s hardwired off-chip memory interfaces, special onboard RAM and host RAM ports are defined. When these ports are exposed on the top level of the dataflow graph, the later HDL code generation stage can easily connect them to the actual hardware ports.

### 4.8.2 VHDL Level

For the purpose of generating the code for hardware synthesis, the VHDL Level is introduced. This representation is very close to actual VHDL code due to its notion of modules with generics, ports and inner (sub-) modules, as well as signals and connections.

Each module in this IR includes the name of a template file to specify its behaviour. Shir contains more than 30 fine granular, composable VHDL templates, see Appendix A, to realise the user’s desired behaviour for the FPGA. There are complex templates like ReadAsync and WriteAsync with several hundred lines of code, but also
more simple ones, for example identity and integer addition.

The architecture section of these template files can contain any VHDL code. This provides all the flexibility and control needed to specify any desired behaviour of the FPGA, including device-specific features. Note that the Shir framework is not bound to VHDL code. An extension to support Verilog code, for instance, is easily possible.

The templates share a uniform port structure to guarantee their interoperability and allow them to be easily plugged together to achieve the desired behaviour. Each template has an ingoing clock and reset signal. Moreover, all the signals of the ports are defined, which are similar to the Dataflow Level. For instance, the four ports data, last, valid, ready are again used to implement the handshaking protocol. The valid and ready signals are std_logic bits, while the last signal is a std_logic_vector for the possibly multiple dimensions of a stream. The last port can also have a negative bit range (in VHDL: -1 downto 0) to create no wire at all, in case basic data is transmitted.

The behaviour of the templates is coded in such a way that the critical paths, see Section 2.1.5, are neither too short, wasting cycles and adding some latency to the overall design, nor too long, throttling the clock frequency or even breaking the design. Sequential combinational operations are combined and processed in one cycle to reduce cycle count. More complex operations may contain internal state machines and have registered inputs and outputs, e.g., Mul and Mul2Add. This adds some latency to the overall design but is necessary for the implementation on an FPGA.

The VHDL templates are independent of Shir and can be integrated into other frameworks. They can even be simulated standalone without any modifications, which is helpful for testing and debugging them.

4.9 Summary

The core Shir language is standard, similar to typed lambda calculus, augmented with support for generics and subtyping. These features allow easy extensions of this language. Based on this core, multiple IRs with specialised types and primitives are introduced. The Algorithmic Level defines a functional language with common parallel patterns, such as Map and Reduce, to specify the input program.

For hardware synthesis, this high-level abstraction is then lowered to the Architecture Level. On this level, three major collection types, depicted in Figure 4.17, express hardware design decisions, that crucially affect the performance. The primitives from
the Algorithmic Level are refined accordingly. In the context of hardware generation, a `MapStm` produces a sequential process in *time*, consuming and producing a stream of data with potential for pipeline parallelism. In contrast, `MapVec` processes data in *space*, resulting in spatial parallelism with the function *f* duplicated in hardware for every element of the vector.

Furthermore, on the Abstract Memory Level, a new ramarray type represents data in memory. In combination with `Read` and `Write` primitives, this allows to specify arbitrary memory usage, for example input data from host RAM and on-chip buffers for faster repeated data access. That way all the available kinds of memory on the FPGA, as depicted in Figure 2.2, can be exploited.

Since this abstraction is too high to map it directly to hardware modules, the lower Hardware Memory Level is introduced. Here, memory is represented as functions and thus, in a more compatible way for hardware synthesis. According to the different types of memory available on an FPGA, a notion of synchronous and asynchronous access is added in this IR. The Device-Specific Level allows to make the best possible use of specialised hardened functions on the specifically targeted FPGA.

Finally, to emit HDL code, the lowest functional IR is turned into a dataflow graph and then a VHDL IR. These representations add hardware level features, like handshaking to synchronise the communication between the components in the design. VHDL offers all the flexibility needed to achieve high-performance hardware.

All in all, the multi-level IR design of the **Shir** compiler allows to easily apply transformations of the program on the level that best suits the optimisation at hand. By separating different aspects of hardware design into individual IRs, the compiler structure remains clean and easily extensible.

The following chapter deals with the lowering process, *i.e.*, how a higher IR is automatically turned into a lower IR.
Chapter 5

Lowering Passes

The previous chapter has shown the multi-level IRs of Shir, where each abstraction exposes different details to work with. While hardware-specific IRs are great at expressing hardware choices, a mechanism is still required to go from one level to the next lower level until a form suitable for the generation of an efficient accelerator is reached. This chapter presents the compiler passes in Shir that automatically deal with these lowering steps. They are visualised as solid edges in Figure 4.1 from the previous chapter.

Once again, the multi-level IR approach helps to separate the concerns. Each lowering pass focuses only on one particular task in the compilation, while the optimisations are left aside for the rewrite rules in the next chapter. By taking one step at a time, the individual passes remain more manageable compared to a monolithic compilation that tries to do too much at once.

All these passes share a common approach to reach the next lower level: The given IR tree of one level is traversed and each expression is transformed in a deterministic way. This entire process is transparent from the programmer and does not require any user input.

The rest of this chapter is organised as follows. Section 5.1 describes the first lowering step, which takes the hardware-agnostic, algorithmic specification and introduces architecture design choices. After that, the memory expressions are refined in Section 5.2. In Section 5.3, a dataflow graph is generated from the lowest level functional IR. Section 5.4 turns this representation into VHDL code and describes how the host machine interacts with FPGA to start the computation. Finally, Section 5.5 concludes this chapter.
5.1 Architecture Design Choices

The initial IR in Shir is an algorithmic specification provided by the user or high-level tool. It is hardware-agnostic and no implementation details are defined yet.

The first step in the compiler is to lower this expression into a piece of Architecture IR. On this level, basic optimisations, such as parallelisation and memory usage, can be expressed, as Chapter 6 will show later. In order to perform this lowering step, the Shir compiler traverses the IR and automatically replaces each occurrence of an algorithmic expression by an equivalent expression from the Architecture Level. This procedure does not require any manual input from the user and is solely captured in the lowering instructions $L_A$, see Figure 5.1.

The recursive lowering starts in expression (5.1), where the provided root of the algorithmic specification is wrapped in an expression that allocates memory in host RAM and writes the $N$ elements returned by expression root back to it. Depending on the type of root, this might require multiple nested maps, which is represented by the `MapStm*` keyword. Data that is to be written back to the host must be based on the cache line size. Shir can generate the data type conversion from the given type of root to a cache line based type automatically, as will be explained later in this section.

In most cases there is a simple one to one translation, as in expressions (5.2) to (5.4) and (5.7) for example. Division gets a little more complicated, since not all FPGAs come with a hardwired implementation of this operator. Instead, the lowering step in expression (5.5) replaces it with a right Shift operator, if the divisor is a power of two. For this, the divisor must be statically known. If the compiler finds a `ConstInt` primitive at the second position in the input tuple, it is able to extract the divisor $N$ at compile time and transform it into a number of bits for the Shift operator. In all other cases, where the divisor is unknown or not a power of two, an exception is thrown in expression (5.6) and the compilation is aborted, because the target platform does not support the specified design. For future work, to support other FPGAs with division units, the corresponding primitive can be implemented on the Architecture Level.

Generally, the primitives are always lowered to their stream-based counterpart, e.g., `Map` becomes `MapStm` in this lowering process. Thus, the very initially generated lower-level design will always be a fully stream-based, minimal area implementation with no parallel computation. However, once the Architecture Level is reached, optimisations can take effect as described in Section 6.1. They rewrite the IR and trade in the FPGA’s area for performance.
5.1. Architecture Design Choices

\[ \text{Init}[\text{root}] = \text{Let } \text{mem} = \text{MemAlloc}(<\text{HostRamT}, \ldots, N> \text{ in} \quad (5.1) \]
\[ L_A[\text{root}] \triangleright \text{MapStm}^*(\lambda \text{data} \cdot \text{data} \triangleright \text{Write}(\text{mem})) \]
\[ L_A[\text{Add}] = \text{Add} \quad (5.2) \]
\[ L_A[\text{Sub}] = \text{Sub} \quad (5.3) \]
\[ L_A[\text{Mul}] = \text{Mul} \quad (5.4) \]
\[ L_A[\langle e, \text{ConstInt}<N> \rangle \triangleright \text{Div}] = L_A[e] \triangleright \text{Shift}<1, \log_2 N> \quad (5.5) \]
\[ // \text{assert } N \text{ is a power of two} \]
\[ L_A[\text{Div}] = \emptyset \quad // \text{otherwise throw exception} \quad (5.6) \]
\[ L_A[\text{Id}] = \text{Id} \quad (5.7) \]
\[ L_A[\text{ConstInt}<N>] = \text{ConstInt}<N> \quad (5.8) \]
\[ L_A[\langle e1, e2 \rangle] = \langle L_A[e1], L_A[e2] \rangle \quad (5.9) \]
\[ L_A[\text{Select}<N>] = \text{Select}<N> \quad (5.10) \]
\[ L_A[\text{Map}(f)] = \text{MapStm}(L_A[f]) \quad (5.11) \]
\[ L_A[\text{Reduce}(f, e)] = \text{ReduceStm}(L_A[f], L_A[e]) \quad (5.12) \]
\[ L_A[\text{Slide}<S_W, S_S>] = \text{SlideStm}<S_W, S_S> \triangleright \text{MapStm}(\text{VecToStm}) \quad (5.13) \]
\[ L_A[\text{Zip}] = \text{ZipStm} \quad (5.14) \]
\[ L_A[\text{Split}<N>] = \text{SplitStm}<N> \quad (5.15) \]
\[ L_A[\text{Join}] = \text{JoinStm} \quad (5.16) \]
\[ L_A[\text{Input}<T, D_1 \ldots D_N>] = \text{Let } \text{mem} = \]
\[ \text{MemAlloc}<\text{HostRamT}, \ldots, D_1^* \ldots D_N^*> \text{ in} \]
\[ \text{Counter}<0, 1, D_1^* \ldots D_N^*> \]
\[ \triangleright \text{MapStm}^*(\lambda \text{addr} \cdot \text{addr} \triangleright \text{Read}(\text{mem})) \quad (5.17) \]

Figure 5.1: Lowering \( L_A \) of Algorithmic Level to Architecture Level primitives with a flat memory representation (explained in Section 4.5). Expressions are represented by \text{root}, \text{e}, \text{e1}, \text{e2} and \text{f}. \text{T} is a Value type, while \text{N}, \text{D}_1 and \text{D}_N are Nat types. Type parameters are omitted in most cases, because they are equal on both sides of the equation.
When it comes to the SlideStm primitive, it is lowered differently because of its hardware implementation. As seen in Figure 4.10, the generated output is a stream of vectors. These inner vectors must be converted into streams, by using MapStm(VecToStm) in expression (5.13), to fit the other parts (and their types) of the stream-based, lowered design.

The Input primitive in expression (5.17) is lowered to an expression, that reads the input data from host RAM. If this data has multiple dimensions, nested MapStm are required, as indicated by MapStm*. Depending on the specified input type, a conversion from a cache line based type to this desired data type may become necessary. Again, SHIR generates the conversions automatically for these cases, as will be explained in the following paragraphs. For clarity, this conversion step is omitted in Figure 5.1.

The number of cache lines required for the input data, does not necessarily have to match the number of input elements, due to the conversion. That is why the counter in expression (5.17) counts up to \(D_1 \times \ldots \times D_N\), which are derived from the original dimensions \(D_1 \ldots D_N\) but not always equal. Due to the flat memory representation, the memory allocation considers the product of these dimensions \(D_1 \times \ldots \times D_N\), when calculating how many elements are read.

**Cache Line Conversion**

As we have seen, it is often necessary to transform data into another format. Generally, this happens when a hardwired interface (e.g., to memory) is involved in the FPGA application. These hardware constraints dictate a specific format (e.g., 512 bits), which does most likely not match the desired data types (e.g., 32 bits). For these use-cases, an automatic conversion generator reshapes the given data into the desired format by creating a combination of the primitives Split, Join, Drop, Concat and conversions between vectors and streams, namely StmToVec and VecToStm.

Figure 5.2 provides an example of how a single cache line of 512 bits is converted into a stream of 3-bit values, where the primitives DropVec, SplitVec, VecToStm reshape the data. Likewise, a conversion in the opposite direction is generated using the primitives StmToVec, JoinVec and ConcatVec. The latter is used to pad the vector with two more zero elements to achieve a total length of 512 bits again.

In real world applications, the amount of input data usually exceeds a single cache line of 64 bytes, so that a stream of cache lines is transferred to or from the host. The conversion generator takes this into account and wraps the primitives into MapStm to process cache line by cache line.
5.2 Lowering Memory Expressions

Prior to the lowering from Abstract Memory Level to Hardware Memory Level, all ramarrays in the given expression are mapped into memory regions, defined by a base address and a length. In case there is more than one allocation in the same memory (e.g., multiple inputs from host RAM), the required memory regions are laid out one after another, by mapping an incremented base address to each of these allocations.

Data is always aligned to the cache lines, i.e., a defined memory region invariably starts at the beginning of a cache line. This is a common compiler design decision for performance reasons (not only for FPGA but also CPU targets for instance). Multi-dimensional streams are stored in host RAM so that the start of the innermost stream is aligned to the cache line. This simplifies navigation in memory, like jumping back to the beginning of a stream to reread data, which is for example needed when

Figure 5.2: Converting the cache line of $CL$ bits (here 512) on the left into a stream of $\lfloor \frac{CL}{N} \rfloor N$-bit integers on the right. In this example, $\exists[Inst(3)]_{170}$ is the desired type. First, the 2 padding bits ($512 \mod 3$) are removed. The red dashed lines indicate what has changed in the first two conversion steps.

On the software side, host RAM data must be laid out accordingly. Based on the given IR, the SHIR compiler generates memory images, where the input data is already put into the right format for the FPGA design. These images are stored as binary files on the hard drive to be loaded into the RAM before executing the application.
Chapter 5. Lowering Passes

Figure 5.3: Mapping (blue lines) of multiple \texttt{MemAlloc} expressions from the IR (left) to regions in host RAM (right). Each memory allocation is aligned to the 512-bit wide cache lines. Dashed red lines indicate unused memory due to lesser amounts of data (\texttt{mem2}) or data with non-power-of-two precision (\texttt{mem3}).

multiplying matrices, as we will see in the following chapters.

Figure 5.3 shows a piece of IR with three memory allocations in host RAM. The first one maps 16 64-bit elements to two cache lines. The second one, \texttt{mem2}, takes only 160 bits in total, so that the remaining 352 bits of the cache line are unused. In the third allocation, \texttt{mem3}, the elements have a precision of 52 bits to fit 18 elements in two cache lines instead of only 16 64-bit elements. This leaves a rest of 44 bits (512 \text{ mod } 52) in each cache line. The automatic conversion generator, introduced in the previous section, takes this padding into account when packing data into memory or unpacking data from memory.

After memory allocation, off-chip memories are initialised as described in Figure 5.4. First, one read and one write memory controller for the host RAM are added as in expression (5.18). These controllers are instantiated in any case, because host memory is always present in the context of a hardware accelerator. Then, in expression (5.19), a pair of memory controllers is added recursively for each onboard RAM bank that is used in the given expression.

In order to keep track of memory controllers and use these shared functions again at a later step in the lowering process, read controllers are put into the map container \texttt{rdCtrls} and write controllers are put into \texttt{wrCtrls}. The helper function \texttt{put(m, id, mc)} stores the memory controller \texttt{mc} in map \texttt{m}, while \texttt{get(m, id)} returns the previously stored memory controller with id \texttt{id}. Furthermore, to keep the rules short, \texttt{memId(e)} represents the id of a \texttt{MemLocT} of the \texttt{ramarray} type of the expression \texttt{e}. The shortcut \texttt{memLoc(e)} returns the \texttt{MemLocT} of the \texttt{ramarray} type of the expression \texttt{e}.
5.2. Lowering Memory Expressions

\[\text{Init}[\text{root}]\] (5.18)
\[
= \text{Let } rdCtrl = \lambda r. r \triangleright \text{ReadHostMemCtrl} \text{ in }
\]
\[
\text{Let wrCtrl} = \lambda w. w \triangleright \text{WriteHostMemCtrl} \text{ in }
\]
\[
\text{put}(rdCtrls, \text{HostRamId}, rdCtrl);
\]
\[
\text{put}(wrCtrls, \text{HostRamId}, wrCtrl);
\]
\[
\text{InitOnboardRam}[\text{root}]
\]

\[\text{InitOnboardRam}[e] \text{ if } \text{mem occurs in } e\] (5.19)
\[
\text{with memloc}(\text{mem}) = \text{OnboardRamT}(\text{bankId})
\]
\[
\text{and } \text{memId}(\text{mem}) \notin rdCtrls
\]
\[
= \text{Let } obram = \lambda p. p \triangleright \text{OnboardRam}(\text{bankId}) \text{ in }
\]
\[
\text{Let } rdCtrl = \lambda r. r \triangleright \text{ReadOnboardMemCtrl}(obram) \text{ in }
\]
\[
\text{Let wrCtrl} = \lambda w. w \triangleright \text{WriteOnboardMemCtrl}(obram) \text{ in }
\]
\[
\text{put}(rdCtrls, \text{memId}(\text{mem}), rdCtrl);
\]
\[
\text{put}(wrCtrls, \text{memId}(\text{mem}), wrCtrl);
\]
\[
\text{InitOnboardRam}[e]
\]

\[\text{InitOnboardRam}[e] \quad // \text{else (all used banks are initialised)}\] (5.20)
\[
= \text{L}_M[e]
\]

Figure 5.4: Initialising off-chip memory as a preparation for the memory lowering pass \(L_M\) in Figure 5.5. put, get, memId and memloc are helper functions to keep this description short and readable. Additional if-conditions restrict the application of certain rules. The remaining lower case terms are expressions, where \(\text{mem}\) is a MemAlloc expression.
Chapter 5. Lowering Passes

\[ L_M[\text{Let } p = \text{mem in body}] \text{ if memId(\text{mem})} \notin \text{rdCtrls} \] (5.21)

= Let \( \text{bram} = \lambda p . p \triangleright \text{BlockRam} \) in

Let \( \text{rdCtrl} = \lambda r . r \triangleright \text{ReadSyncMemCtrl}(\text{bram}) \) in

Let \( \text{wrCtrl} = \lambda w . w \triangleright \text{WriteSyncMemCtrl}(\text{bram}) \) in

\[ \text{put}(\text{rdCtrls}, \text{memId(\text{mem})}, \text{rdCtrl}); \] 
\[ \text{put}(\text{wrCtrls}, \text{memId(\text{mem})}, \text{wrCtrl}); \] 

Let \( L_M[p] = L_M[\text{mem}] \) in \( L_M[\text{body}] \)

\[ L_M[\text{MemAlloc}...] = \text{ConstInt}<\text{baseaddr}> \] (5.22)

\[ L_M[\text{MapStm}(\lambda \text{addr} . \text{addr} \triangleright \text{Read(\text{mem})})] \text{ if memloc(\text{mem})} \neq \text{BlockRamT} \] (5.23)

= \text{ReadAsync}(\text{get}(\text{rdCtrls}, \text{memId(\text{mem})}), L_M[\text{mem}])

\[ L_M[\text{ReduceStm}(\lambda m . \lambda d . d \triangleright \text{Write(m, \text{mem})})] \text{ if memloc(\text{mem})} \neq \text{BlockRamT} \] (5.24)

= \text{WriteAsync}(\text{get}(\text{wrCtrls}, \text{memId(\text{mem})}), L_M[\text{mem}])

\[ L_M[\text{Read(\text{mem})}] \text{ if memloc(\text{mem})} \neq \text{BlockRamT} \] (5.25)

= \text{Repeat}<1> \triangleright \text{ReadAsync}(\text{get}(\text{rdCtrls}, \text{memId(\text{mem})}), L_M[\text{mem}]) \triangleright \text{JoinStm}

\[ L_M[\text{Write(\text{mem})}] \text{ if memloc(\text{mem})} \neq \text{BlockRamT} \] (5.26)

= \text{Repeat}<1> \triangleright \text{WriteAsync}(\text{get}(\text{wrCtrls}, \text{memId(\text{mem})}), L_M[\text{mem}]) \triangleright \text{JoinStm}

\[ L_M[\text{Read(\text{mem})}] \text{ if memloc(\text{mem})} = \text{BlockRamT} \] (5.27)

= \text{ReadSync}(\text{get}(\text{rdCtrls}, \text{memId(\text{mem})}), L_M[\text{mem}])

\[ L_M[\text{Write(\text{mem})}] \text{ if memloc(\text{mem})} = \text{BlockRamT} \] (5.28)

= \text{WriteSync}(\text{get}(\text{wrCtrls}, \text{memId(\text{mem})}), L_M[\text{mem}])

\[ L_M[\text{other}] = \text{other} \text{, with } L_M \text{ applied on children of } \text{other} \] (5.29)

Figure 5.5: Lowering \( L_M \) of Abstract Memory Level to Hardware Memory Level primitives. \( \text{put}, \text{get}, \text{memId} \) and \( \text{memloc} \) are helper functions and \( \text{mem} \) is a \( \text{MemAlloc} \) expression.
After all the required functions for off-chip memories are added to the expression, the actual lowering pass starts. The Shir compiler traverses the given expression in a recursive way and the lowering instructions $L_M$ in Figure 5.5 are followed. If none of the first lowering instructions are applicable, the very last one, expression (5.29), ensures that the recursive descent of $L_M$ is continued. Some of these contain additional if-conditions, that must hold to allow the substitution. Again, no manual user input is required for this procedure.

Whenever $\mathtt{Let}$ is encountered with a $\mathtt{MemAlloc}$ for a memory location, that has not been visited before during traversal, the rule expression (5.21) is applied. This creates a block RAM and the corresponding read and write memory controllers, which are put in the $\mathtt{rdCtrls}$ and $\mathtt{wrCtrls}$ maps. The block RAM function is shared among the two controllers. In expression (5.22), the $\mathtt{MemAlloc}$ itself is replaced by a $\mathtt{ConstInt}$ that returns the allocated base address. This address is defined in the mapping of memory regions, as described in the very beginning of this section.

The following rules lower the abstract $\mathtt{Read}$ and $\mathtt{Write}$ primitives to the more specific $\mathtt{ReadAsync}$ and $\mathtt{WriteAsync}$ for asynchronous memory and $\mathtt{ReadSync}$ and $\mathtt{WriteSync}$ for synchronous memory. In expressions (5.23) and (5.24), the substitution results in expressions for reading and writing with concurrent memory requests. This requires the former $\mathtt{Read}$ and $\mathtt{Write}$ expressions to be nested in a $\mathtt{MapStm}$, resp. a $\mathtt{ReduceStm}$. If this is not the case, an inefficient implementation of only one read or write request at a time is generated, as in expressions (5.25) and (5.26). For this case, the primitive $\mathtt{Repeat}$ is used to generate an address stream with a length ($N$) of 1 from the given single input address.

After these steps, the lowering to the Hardware Memory Level is completed, without requiring any user input. The simple and abstract expression of memory has been mechanically translated into a refined representation based on functions. Former local occurrences of off-chip memory allocations in the IR have been replaced by function calls to globally defined memory controllers. This simplifies connecting the generated off-chip memory ports to the real hardware interface on the FPGA at a later stage. In the next lowering step a dataflow graph is generated to approach a more hardware-like representation, before finally emitting VHDL code.
5.3 Dataflow Graph Generation

In this section, the low-level functional IR is transformed into a dataflow graph. This representation resembles the actual hardware design on the FPGA, as wires, ports and modules are exposed. There is an almost one-to-one mapping between functions from the IR and nodes in the graph. Again, for the lowering, a recursive bottom-up approach is employed: The leaves of the IR tree are turned into nodes first. Then, their parent expression is processed. A graph for this parent expression is created, involving the child nodes, which are wired together. This process continues until the root node of the expression is reached and the entire nested graph of the program is generated. This output is similar to the block diagrams we encountered previously in Figures 4.7, 4.13 and 4.15.

At this stage, the functional input IR must guarantee the following properties: First, no high-level expression from the Algorithmic or Abstract Memory Level must be present in the expression. Second, parameter definitions must only occur directly in a lambda abstraction. Third, the body of a lambda or Let must use the defined parameter at least once. Otherwise a disconnection might appear in the generated hardware implementation. The compiler checks that these properties hold to catch for any possible implementation bug.

Arbitration of Shared Resources

If only one parameter access occurs in the body of a Let, its argument is directly connected to the body using plain wires in hardware. Whenever the parameter is used multiple times, a resource, e.g., memory, is shared among all the places where the parameter is accessed, the clients.

Shared resources can be accessed by only one client at a time. During generation of the dataflow graph, such sharing is detected and arbiter nodes are inserted. A round robin scheduling strategy is used to fairly distribute access among the clients. There are three types of arbiters in ShIR, as shown in Figure 5.6.

The first one distributes shared data to its clients and makes sure they all receive it, even if they are not ready at the same time. For this, it maintains a list of clients that have received the current incoming piece of data, which is only consumed if all the clients are in this list. Especially in the case where the clients have different timing characteristics, they must be arbitrated.
The following example expression shows such a scenario, where \( f \) and \( g \) have internal pipelines of different length. A simple wire connection would not be sufficient and a distributor logic between the shared resource and the clients is required:

\[
\text{Let } \text{shared} = \text{Counter}<0, 1, N> \text{ in } \langle \text{shared} \triangleright f, \text{shared} \triangleright g \rangle \triangleright \text{Mul} \tag{5.30}
\]

A similar pattern has occurred before in expression (4.47), where a stream is reshaped and shared among two write operations to different memory banks.

The second arbiter is used when synchronous functions, such as block ram and its synchronous read and write controllers, are shared. Each call to the function must return before the next call, from possibly another client, is accepted. Only the currently selected client is connected to the resource’s data and handshake signals. This arbiter is required in expression (4.60) from the previous chapter, for example.

The last of the three arbiter types deals with shared asynchronous functions, such as onboard memory or host RAM controllers. For best performance, this arbiter must not grant access exclusively to one client only. It must be able to flexibly take requests from all its clients to maximise the request queue of the shared function. The implementation of the arbiter therefore contains a queue to keep track of which client has sent which request. Once the response arrives from the shared function, the information in this queue is used to forward the response to the right client.
An asynchronous arbiter is needed, for example, to connect a shared read host memory controller to the reading clients. This happens when the high-level program on the Algorithmic Level contains more than one Input expression, or multiple ReadAsync expressions occur on the Hardware Memory Level after lowering, as in the following expression:

\[
\text{Let } \text{rdCtrl} = \lambda r . r \triangleright \text{ReadHostMemCtrl} \text{ in } \text{Counter} < 0, 1, N > \triangleright \text{ReadAsync}(\text{rdCtrl}, \ldots) \triangleright \text{MapStm}(\ldots \text{Counter} < 0, 1, N > \triangleright \text{ReadAsync}(\text{rdCtrl}, \ldots) \triangleright \ldots)
\]

All in all, these arbiters are necessary modules to implement resource sharing in hardware, as specified by corresponding high-level Let expressions. They are essential to enable the memory usage as modelled in the SHIR IR.

### 5.4 Hardware Design Generation

Based on the information from the dataflow graph, the generation of actual VHDL code is straightforward. The required VHDL code templates are loaded from a database. See Appendix A.1 for a complete overview table and Appendix A.2 for some examples. All the graph’s edges are translated into VHDL statements and written into a ‘wrapper’ file at the top level of the design hierarchy.

**Interfacing to Host**

The generated top-level wrapper VHDL file contains the entire hardware design, while only the ports of the read and write controllers for host RAM and the onboard memory banks are exposed. These ports are connected to the FPGA’s DMA engine and the hardware interface of the DDR memory, as visualised in Figure 5.7.

On the software side of the host machine, a small C program loads the initial memory image into a huge\(^1\) pinned memory page in the host RAM. This program communicates with the FPGA using the Open Programmable Acceleration Engine (OPAE) [LLC17]. The OPAE C Application Programming Interface (API) makes the accelerator’s resources accessible and provides associated operations. The base address of the initial memory region is sent through this interface to the FPGA via Memory-Mapped I/O (MMIO). Now, all the requested addresses from the FPGA, are offset by

---

\(^1\)On Linux, memory pages up to 1 GB are made available by enabling so-called huge pages.
5.5 Summary

Thanks to the multi-level structure of the compiler, the complex task to generate low-level hardware code without user guidance, is divided into multiple simple and manageable lowering passes. This chapter has introduced them and explained how they work based on a concise list of lowering instructions, which are mechanically followed when traversing the input IR. However, this process introduces performance inefficiencies, which require optimisation. The following chapter will explain what these inefficiencies are and how they are addressed using an automated rewrite process.
Hardware accelerators are used to run workloads more efficiently compared to solutions implemented in software. FPGAs offer an ideal platform for this due to their reconfigurability and flexibility (Section 2.1), which allows them to adapt to constant workload changes.

However, with great design options comes great responsibility to make the right design choices. The available FPGA resources must be exploited in the best way possible to maximise the performance. The design space for hardware accelerators is huge and it is challenging to find the best implementation.

With so many design choices, manual optimisation is too exhaustive and time-consuming. Hence, an automated process is needed to modify the IR and eventually yield the optimised hardware design without any user input.

Rewriting systems, as introduced in Section 2.3, are a good match for such optimisation processes, especially for functional IRs as in Shir. The optimisations are encoded as rewrite rules, capable of drastically changing the hardware, even on a structural level. Moreover, rewrite rules keep the optimisations simple and easily extensible for new algorithms and new target hardware platforms. The rewriting approach is sound, because applying the rewrite rules does not change the original program semantics but only the manner of implementation. This guarantees correctness by construction. Since each rewrite is type-preserving, any combination of rewrites will preserve program semantics and result in an expression that is synthesisable.

The lowering process of the previous chapter has only targeted a single point in the design space, while no optimisations were applied. In this chapter, the designs are modified more severely to improve their performance.
Figure 6.1: Optimisation steps in the Shir compiler, represented as boxes. The order is indicated by solid arrows. Each optimisation consists of one or more rewrite rules and has a dedicated section in this chapter. The dashed arrows are lowering steps, as explained in the previous chapter.

Each section presents one specific step in the optimisation flow of Shir. Since there are dependencies between these steps, they must be executed in order, as depicted in Figure 6.1. In each step, a set of rules is applied in a fixed-point iteration (explained in Section 2.3) — as much as possible, until no further rewrite is possible.

First, Section 6.1 introduces rewrite rules to parallelise the design. More of the FPGA’s compute resources are used for better performance results. In Section 6.2, fast buffers are automatically inserted into the IR whenever data is reused. These two optimisation steps introduce bottlenecks, due to conversions between stream and vector data types, that are addressed in Section 6.3 to improve the performance. This dependency is the reason, why the first three rewrite steps must be ordered as in Figure 6.1.

Section 6.4 first identifies the problems created by data reshaping in the context of hardware synthesis and then shows how to make these problems literally disappear by applying rewrite rules. In Section 6.5, the communication to off-chip memories, which is crucial for the performance of accelerators, is optimised. Section 6.6 exploits specialised hardened functions on the FPGA that are specific to certain models only. Then, Section 6.7 deals with timing issues due to long signal paths on the FPGA and how to fix them. This step must come last in the optimisation flow, because it requires that the design has already settled and no major structural changes are made. Section 6.8 explains why the rewriting process is always guaranteed to terminate and has a deterministic outcome. Finally, Section 6.9 summarises the optimisations presented in this chapter.
6.1 Spatial Parallelism

The compute power of FPGAs lies in their highly parallel structure. Generally, the more resources on the chip are exploited, the better the performance.

When the Architecture Level is reached, as previously explained in Chapter 5, the initial expression defines a stream-based design with minimal area usage, using as few FPGA resources as possible. In this case, the dot product is expressed as follows:

\[
\langle \text{inputA, inputB} \rangle \rightarrow \text{ZipStm} \rightarrow \text{MapStm(Mul)} \\
\rightarrow \text{ReduceStm}(\lambda a. \lambda b. \langle a, b \rangle \rightarrow \text{Add, ConstInt<0>})
\]  

Problem Statement

As the block diagram in Figure 6.2a indicates, not many adders and multipliers are employed. The hardware implementation processes element by element. For design space exploration the compiler must be able to trade in available resources on the FPGA for increased throughput with a more parallelised computation.

Optimisation

The compiler achieves this automatically by substituting stream-based primitives with their vector-based counterparts, as in this rewrite rule for mapping operators:

\[
\text{MapStm}(f) \\
\rightarrow \text{if } f \text{ contains computation} \rightarrow \text{StmToVec} \rightarrow \text{MapVec}(f) \rightarrow \text{VecToStm}
\]  

This rule replaces a \text{MapStm} with a function \( f \) by a \text{MapVec}, where many instances of \( f \) are created to process all the elements in parallel at the same time, as explained in Section 4.4.3. To maintain correctness of types in the IR after rewriting, the input stream must be converted accordingly, using \text{StmToVec} primitives. Similarly, a \text{VecToStm} turns the resulting vector back into a stream. Note, that the rule is only triggered if some computation operations, e.g., addition, subtraction or multiplication, are found within function \( f \). These operations must not be nested in another \text{MapStm}. Thus, in case there is a nesting of \text{MapStms}, the rule’s application is restricted to only the innermost mapping operator. If the function contains only data type conversions, the mapping operator remains untouched, because no parallelisation is needed.
Chapter 6. Optimisation Using Rewrite Rules

(a) Fully stream-based dot product. The reduction contains an accumulator (box labelled ‘accum.’) and produces a valid result only after at least \( n \) cycles.

(b) Parallel implementation. Multiple parallel multipliers and adders are instantiated. This design can produce a valid output every cycle at best.

(c) Partially parallel implementation. The inner vector-based dot product is wrapped in an outer sequential operation on streams.

Figure 6.2: Block diagrams for dot product implementations with different levels of parallelism. Blue boxes are primitives.
Reductions are parallelised using this rule:

\[
\text{ReduceStm}(f, e) \quad \text{if } f \text{ contains computation}
\]

\[
\text{StmToVec} \triangleright \text{ReduceVec}(f, e) \quad // \text{ reduction tree smart constructor}
\]  

After applying these two rules (6.2) and (6.3) on the initial dot product in expression (6.1), the following expression is obtained:

\[
\langle \text{inputA}, \text{inputB} \rangle \triangleright \text{ZipStm}
\]

\[
\triangleright \text{StmToVec} \triangleright \text{MapVec} \langle \text{Mul} \rangle \triangleright \text{VecToStm}
\]

\[
\triangleright \text{StmToVec} \triangleright \text{ReduceVec} \langle \lambda a. \lambda b. \langle a, b \rangle \triangleright \text{Add}, \text{ConstInt}<0> \rangle
\]

This implementation uses more adders and multipliers in parallel, somewhat similar to Figure 6.2b. However, the produced design is suboptimal due to the three inserted conversions between streams and vectors which take multiple clock cycles. Section 6.3 will discuss in more detail how further rewrite rules can optimise all these bottlenecks away. Nevertheless, at this point it is already obvious that the VecToStm followed by a StmToVec are two unnecessary steps, which cancel each other out and can therefore simply be removed.

The level of parallelism is further adjustable: Mixed stream and vector designs are possible by first reshaping the input stream and then only partially parallelising the computation. In order to process \(N\) values in parallel, the input is split into chunks of \(N\) elements in advance. The overall computation of the dot product then happens in a pipelining (stream) of parallel computations on vectors, as visualised in Figure 6.2c:

\[
\langle \text{inputA} \triangleright \text{Split}<N> \triangleright \text{MapStm} \langle \text{StmToVec} \rangle, \quad // \text{ reshape inputs and}
\]

\[
\text{inputB} \triangleright \text{Split}<N> \triangleright \text{MapStm} \langle \text{StmToVec} \rangle \quad // \text{ create tuple}
\]

\[
\triangleright \text{ZipStm} \triangleright \text{MapStm} \langle \text{ZipVec} \rangle
\]

\[
\triangleright \text{MapVec}(\text{Mul}) \quad // \text{ inner parallel multiplication}
\]

\[
\triangleright \text{ReduceVec}(\lambda a. \lambda b. \langle a, b \rangle \triangleright \text{Add}, \text{ConstInt}<0>) \quad // \text{ inner reduction tree}
\]

\[
\triangleright \text{ReduceStm}(\lambda a. \lambda b. \langle a, b \rangle \triangleright \text{Add}, \text{ConstInt}<0>) \quad // \text{ outer stream reduction}
\]

In conclusion, parallelisation is essential for high-performance computation. Different levels of parallelism are easily achieved in the SHIR compiler due to powerful stream and vector types in combination with a set of simple rewrite optimisations.
6.2 Data Reusage

Similar to caches in CPUs and GPUs, memories on FPGAs come in different sizes and access speeds. In some scenarios, where certain pieces of data are frequently used, it makes sense to buffer them in smaller but faster memories. On FPGAs, on-chip RAM is used for such buffers. Whenever data is reread multiple times, they avoid repeatedly requesting that data from slower memories like host RAM.

**Problem Statement**

The lowering process in Chapter 5 only generates a simple implementation that always reads all input data from host memory. For high-performance designs, the compiler must automatically identify expressions that imply repeated reads and insert buffers into them.

A reuse access pattern generally occurs in expressions with a lambda that uses an unbound parameter (explained in Section 2.2.1) in its body, similar to the following example, where $f$ is any binary operation on two streams:

$$\lambda \text{streamA} \cdot \text{matrixB} \triangleright \text{MapStm}(\lambda \text{streamB} \cdot f(\text{streamA}, \text{streamB}))$$  \hspace{1cm} (6.6)

**Optimisation**

In order to create more efficient implementations, ShIR detects these specific combinations of expressions automatically and inserts buffers by applying the following rewrite rule:

$$\lambda \text{streamA} \cdot \text{matrixB} \triangleright \text{MapStm}(\lambda \text{streamB} \cdot f(\text{streamA}, \text{streamB}))$$

\hspace{1cm} if \text{streamA and streamB are streams and not yet buffered} \hspace{1cm} (6.7)

\begin{align*}
\lambda \text{streamA} \cdot \text{matrixB} \triangleright & \text{MapStm}(\lambda \text{streamB} \cdot \\
& f(\\
& \text{Let } \text{bram} = \text{MemAlloc}<\text{BlockRamT}, \ldots, \text{N}> \text{ in} \\
& \quad \text{Let } \text{buffered} = \langle \text{streamA}, \text{Counter}<0, 1, \text{N}> \rangle \triangleright \text{ZipStm} \\
& \quad \quad \triangleright \text{ReduceStm}(\lambda \text{mem} \cdot \lambda \text{data} \cdot \text{data} \triangleright \text{Write}(\text{mem}), \text{bram}) \text{ in} \\
& \quad \quad \quad \text{Counter}<0, 1, \text{N}> \triangleright \text{MapStm}(\lambda \text{addr} \cdot \text{addr} \triangleright \text{Read}(\text{buffered})) \\
& \quad , \text{streamB})
\end{align*}
function MultiDimStreamOperation(s)
    if type(s) = \[\ldots\]_M\_N then  // s is at least two-dimensional
        flatStm ← JoinStm(s)
        result ← MultiDimStreamOperation(flatStm)
        return result >> SplitStm<length(s)>  // length of the outermost dimension
    else
        return OneDimStreamOperation(s)
endfunction

Figure 6.3: A recursive function to make one-dimensional stream operations available for multidimensional streams.

The condition in this rule prevents infinite rewriting loops. Moreover, it is only triggered when streamA and streamB are streams, in which case their repetition is non-trivial and a buffer is actually beneficial.

In order to buffer multidimensional streams, the recursive function in Figure 6.3 is employed. The input stream is first flattened using as many JoinStm primitives needed to create a one-dimensional stream. This stream is then buffered as in Rule (6.7). After reading, SplitStm restores its original, multidimensional shape.

In matrix multiplication of two N×N matrices A and B, both matrices are read N times (2N^2 rows to read in total). Here, the buffer rewriting rule improves the performance. If the rows of A are buffered, the number of rows to read from host RAM decreases to N^2+N. The same happens, when the entire matrix B is buffered. However, if both the rows of matrix A and the entire matrix B are buffered, the overall number of rows to read from host RAM is drastically reduced to 2N.

Apart from that, the rewrite rules for buffer insertion are able to scale the input and output width of the memory. That way, more data can be read or written in parallel. This optimisation is crucial to enable highly parallel computation, which requires a faster supply with input data. If the computation was more parallel than the provided input data, the compute elements would not perform efficiently, due to waiting times. The sweet spot for matrix multiplication, for example, is reached, when both the output vector of the memory and the input vector to the computation (e.g., dot product) have the same width.

All in all, the memory-aware IR in Shir allows to conveniently perform memory related optimisations. The above presented rewrite rule inserts buffers for repeated data and thus increases the overall performance of the design.
Chapter 6. Optimisation Using Rewrite Rules

6.3 Stream and Vector Conversions

In the previous two sections, the initial design is first parallelised, in Section 6.1, and then fast intermediate buffers are inserted, in Section 6.2. This process is visualised in Figures 6.4a to 6.4c.

**Problem Statement**

However, as Figure 6.4c shows, these rewrite steps introduce conversions between streams and vectors, which pose a problem for the communication between memory and computation. In the case of StmToVec, only one ingoing element is consumed at a time, and in the case of VecToStm, it only produces element by element. The overall performance is still impaired due to this bottleneck. Furthermore, this communication may pass through some ‘other operations’, consisting of further computations or even data type conversions, because the memory’s hardware interface may differ from the required input data type of the computation.

**Optimisation**

In order to speed up the communication between memory and computation, a design as shown in Figure 6.4c must be transformed into a design as in Figure 6.4d. The parallel wires near memory and computation must spread out over the entire FPGA, effectively replacing slow stream-based transmissions with faster, vector-based ones.

The compiler automatically achieves this optimisation by moving the VecToStm primitive away from its data source, through the ‘other operations’, towards a potential StmToVec primitive. Once these two conversions meet in the IR, the communication bottleneck is fixed, by removing both conversions with the following rewrite rule:

\[ \text{input} \xrightarrow{\text{VecToStm}} \text{StmToVec} \]

\[ \implies \text{input} \] (6.8)

The VecToStm primitive is moved through the IR by applying a set of 15 rewrite rules, some of which are explained below by means of an example.

If, for instance, a VecToStm in combination with another function \( f \) is found in a MapStm, it is isolated with the following map-fission rule:

\[ \text{MapStm}(f \xrightarrow{\text{VecToStm}}) \]

\[ \implies \text{MapStm}(f) \xrightarrow{\text{MapStm( VecToStm )}} \] (6.9)
6.3. Stream and Vector Conversions

(a) Fully stream-based design with only one multiplier. The memory output width matches the input width of the computational part.

(b) StmToVec is inserted to connect the new $4 \times$ parallelised computation to the remaining part of the design. The memory port width is now the bottleneck.

(c) Memory width scaled by 4. VecToStm is inserted to connect it to the rest of the design. To remove this conversion, it is moved (red arrow) until it reaches a StmToVec.

(d) Final design after rewriting: The high memory bit-width is exploited by the parallelised computation. The performance is no longer impaired by the conversions.

Figure 6.4: Rewriting process from initial design (a) through intermediate steps (b) and (c) to fully parallelised design (d). All wires (black lines) have the same bit-width.
The rewriting system continues to move the combination of \texttt{MapStm} and \texttt{VecToStm} further away from the data source and towards a potential \texttt{StmToVec}. A map-fusion rule combines two \texttt{MapStm} into a single one, enabling further rewrites to then move the \texttt{VecToStm} through function $f$:

\[
\text{MapStm}(\text{VecToStm}) \rightarrow \text{MapStm}(f) \\
\implies \text{MapStm}(\text{VecToStm} \circ f)
\] (6.10)

If a \texttt{VecToStm} meets a \texttt{JoinStm} primitive during its relocation in the IR, it is stuck and cannot skip this \texttt{JoinStm}. It is impossible to replace a \texttt{VecToStm} \rightarrow \texttt{JoinStm} combination with another expression, where the \texttt{VecToStm} occurs on the right-hand side, while leaving the overall semantics untouched. Therefore, the \texttt{JoinStm} also needs to be pushed through the IR, to make room for a more parallelised design. One of the rewrite rules responsible for this works as follows, where \textit{any} is a placeholder for any expression:

\[
\text{JoinStm} \rightarrow \text{any} \\
\implies \text{MapStm(any)} \rightarrow \text{JoinStm}
\] (6.11)

In summary, memory and computation related spatial parallelisation optimisations easily cause a demand for additional conversions to be inserted into the IR to preserve a feasible design. These conversions, namely \texttt{VecToStm} and \texttt{StmToVec}, are a performance bottleneck for the hardware design. The rules presented exemplify how \texttt{VecToStm} is moved in the IR to eventually find a matching \texttt{StmToVec} and then annihilate both conversions. Once the fixed-point iteration terminates, as explained in Section 2.3, the communication wires between a wide memory and a parallelised computation are also parallelised, as in Figure 6.4d. This process is carried out in a similar way for the conversions at the computation’s output data, which is written back to host memory.

The strength of this multi-step approach for parallelisation lies within the separation of concerns: Each individual step is kept simple, even if this means they are sub-optimal at first. One part inserts naive spatial parallelism, the other part deals with removing inefficient conversions. However, in combination, they achieve the desired, parallelised design.
6.4 Efficient Data Reshaping

Despite all the advantages of a high-level functional approach to hardware design, there are some cases where these concepts are too far away from the hardware world. In particular, data reshaping operations such as transposition or sliding of data, as used in 2D convolution for instance, might require substantial amounts of hardware if implemented naively.

Nested Maps also cause problems when accessing different input data as in the case of matrix multiplication. In such cases, the input data must be repeated multiple times. A trivial task in pure software, since memory can be read multiple times. However, in the hardware world, data arrives in the form of streams from memories or data generators. Repeating such a data stream is a non-trivial business, since its state needs to be reset and some control mechanism needs to determine how many times the stream needs to be repeated. This might not be supported by the streaming protocol, or at best ‘only’ introduces long latency if the computation is deeply pipelined.

These issues are not specific to just the compiler presented in this thesis, but to all functional-based IRs used for high-level synthesis. Direct compilation from such a representation to hardware sometimes results in slow performance or even non-synthesisable hardware designs, as we will see in the evaluation in Section 7.4.

In this section, rewrite rules are used to generate high-performance designs in the presence of reshaping data. They make data reshaping operations more efficient on the underlying hardware, while preserving the program semantics. Instead of reshaping the data, as implied by the programmed algorithm, it is more efficient to reorganise the read addresses of that data in memory instead.

The following Section 6.4.1 describes how efficient hardware designs are achieved when processing repeated data. Then, Sections 6.4.2 and 6.4.3 optimise transposition and sliding operations to improve the performance of workloads that involve tiling and convolution.

6.4.1 Repetition of Data

Generally, the application’s input data for the accelerator is read from memory and then streamed through the compute logic of the device. However, some applications require the input data to be repeated.
One simple example for this is the creation of all possible pairs, the *cartesian product*, of the rows of two matrices, as in this functional code:

\[ matrixA \mapsto \text{MapStm}(\lambda rowA\. ) \]
\[ matrixB \mapsto \text{MapStm}(\lambda rowB\. \langle rowA, rowB \rangle \mapsto \text{ZipStm}) \]

Here, the outer MapStm operates on input matrix A, while the inner MapStm operates on input matrix B. Each input matrix comes in the form of a steam of data, generated from reading memory (not shown in the example).

**Problem Statement**

For functional programs executed in software, this example works without any issue. But if hardware is generated in a naive way, by creating a hardware module for each primitive in the program, as depicted in Figure 6.5a, the result will be incorrect.

A naive hardware implementation for MapStm would just extract the rows from the input matrix. Then, the ZipStm returns the pairs of the n-th row of matrix A and the n-th row of matrix B. However, due to the simultaneous use of parameters from two different, nested MapStm expressions, the original program specifies a different behaviour: The rows of matrix A must be repeatedly read from memory. The compiler *must* introduce some extra logic in the generated hardware design to achieve the correct repetition of data.

In general, this is required, whenever a lambda is using an *unbound parameter* in its body, as explained in Section 2.2.1. In the above example, the inner lambda is accessing the unbound parameter rowA, which must therefore be repeated.

A more familiar example for such a memory access pattern is matrix multiplication, with the only difference being the calculation of the dot products from these pairs of rows. This is under the assumption that the second matrix has been transposed in memory already. For two $N \times N$ matrices $A$ and $B$, where $A \times B = C$, each row of matrix $A$ and the entire matrix $B$ are read $N$ times. Thus, the input data must be repeated to provide the correct data for the computation.

This data repetition problem is solved by breaking it down into two simple, automatic steps. First, the compiler detects unbound parameter accesses and inserts explicit Repeat primitives at each access. While this might not be the most optimal location, as we will see shortly, this produces functionally correct hardware. Second, rewrite rules move these new primitives in the IR to more optimal places to improve performance, while preserving the functional semantics of the given application.
(a) The result of the naively generated hardware design is wrong, because the rows of matrix \( A \) are not repeated correctly.

(b) A Repeat block is inserted for the required repetition of one of the input rows. This repeat will back-propagate a signal to the producer to request that the stream must be repeated.

(c) The Repeat has moved into the address generation, to remove the need to back-propagate repeat signals and to improve the performance. The Read block for input \( A \) transforms a 3D stream of addresses into a 3D stream of data, containing \( N \) times each row.

Figure 6.5: Hardware block diagrams to create all pairs of rows for two input matrices. Counters emit 2D streams of addresses, that, after reading, result in 2D streams of values. Newly inserted modules are orange and have a dashed border. A dotted signal entering a MapStm means that this signal is not an input to this MapStm block.
Chapter 6. Optimisation Using Rewrite Rules

Insertion of Explicit Repeat

To enable the compiler to explicitly encode repetitions in the high-level IR, the `Repeat` primitive, introduced in Figure 4.9, is employed. It takes a natural number \( N \) and repeats the given input to create a stream of \( N \) times this input. This primitive can operate on basic data but also streams.

Coming back to the above example, the parameter \( rowA \) is fed into such a `Repeat`. Furthermore, the unbound parameter from the inner lambda is removed entirely using a rewrite rule. This produces an expression where the free parameter is hoisted out of the map by explicitly zipping it with the input. The repeated row of \( A \) and the matrix \( B \) are zipped together and fed into the map expression:

\[
\text{matrixA} \triangleright \text{MapStm}(\lambda rowA. \text{rowA} \triangleright \text{Repeat}<N>\text{matrixB} \triangleright \text{ZipStm} \triangleright \text{MapStm}((\text{ZipStm})))
\]

Thus, each row of matrix \( A \) is repeated \( N \) times before proceeding to the next row. The generated hardware design will contain a new module, as shown in the block diagram in Figure 6.5b, and produce the correct result.

Nevertheless, this design is far from perfect as the repetition of the rows is dynamic and therefore determined during runtime of the hardware. When reading the input data from memory, a counter is generating the addresses to be read. This counter is unable to predict when and which addresses must be repeated. Whenever it emits the last memory address of a row, the counter has to wait until it receives a potential repeat signal to decide whether to repeat the current row or to advance to the next one. Figure 6.6a visualises a pipeline with such a repeat signal. This pipeline between counter and repeat logic must stall to allow this communication. As the corresponding timing in Figure 6.6c shows, this stall causes a delay which depends on the number of registers in the dataflow.

Optimisation

To avoid the delay of the repeat signal, the compiler applies rewrite rules to move the `Repeat` inserted in the previous step. The `Repeat` primitives in the IR are moved step by step towards the IR’s leaves, which are address counters for reading the program’s input data. Effectively, the repetition of data is transformed into a more efficient repetition of addresses, as depicted in Figure 6.5c. The counter and repeat logic communicate within the same pipeline stage, as Figure 6.6b shows.
6.4. Efficient Data Reshaping

![Diagram] (a) Repetition of data: The address generation part must stall to allow the repeat logic to communicate with the counter over multiple pipeline stages. Grey boxes are registers.

![Diagram] (b) Repetition of addresses: The repeat logic is next to the counter. They communicate directly, no pipeline stall required.

![Waveforms] (c) The counter must wait three cycles (orange) for a potential repeat signal before sending the next address row. Here, a repeat row signal is sent, causing the address to jump back to a0.

![Waveforms] (d) The repeat signal is sent instantly when the row’s last address arrives. No cycles are wasted due to pipeline stalls.

Figure 6.6: Block diagrams and waveforms before (a), (c), and after repeat optimisation (b), (d). Here, the 2D counters emit an 8×8 matrix of addresses and memory access takes one cycle. The repeat logic, MapStm(Repeat), sends a signal (red arrow) to the address counter to reset its inner dimension. Superscript denotes nesting of MapStm.
Thus, the generated hardware is able to generate valid memory addresses each cycle, as indicated in Figure 6.6d. This may at first sound like an increase in the number of memory accesses, but the data locality optimisation from Section 6.2 will have placed a fast buffer for the repeated data to compensate that.

This optimisation is implemented as a set of rewrite rules, which are applied as a fixed-point iteration over the IR. To illustrate this process, the previous example with the explicit repetition is brought up again, but this time the part for reading input matrix $A$ is expanded:

$$\text{Counter}<0,1,N,N> \triangleright \text{MapStm}(\text{MapStm}(\text{Read}(\text{mem}))) \quad // \text{read matrix A} \quad (6.14)$$

$$\triangleright \text{MapStm}(\lambda \text{rowA}. \text{rowA} \triangleright \text{Repeat}<N>,\text{matrixB}) \triangleright \text{ZipStm} \triangleright \text{MapStm}(\text{ZipStm})$$

The first rewrite rule needed in this scenario splits the $\text{MapStm}$ ($\text{map-fission}$) to isolate the $\text{Repeat}$ primitive:

$$\text{MapStm}(\text{Repeat}<N> \triangleright f) = = \Rightarrow \text{MapStm}(\text{Repeat}<N>) \triangleright \text{MapStm}(f) \quad (6.15)$$

This rewriting step returns the following expression with one $\text{MapStm}$ for the repetition and another one for the creation of the tuple:

$$\text{Counter}<0,1,N,N> \triangleright \text{MapStm}(\text{MapStm}(\text{Read}(\text{mem})))) \quad (6.16)$$

$$\triangleright \text{MapStm}(\text{Repeat}<N>) \triangleright \text{MapStm}(\lambda \text{rowA}. \text{rowA} \triangleright \text{Repeat}<N>,\text{matrixB}) \triangleright \text{ZipStm} \triangleright \text{MapStm}(\text{ZipStm})$$

Next, the outer $\text{MapStm}$ for reading matrix $A$ and the one for repeating the rows are fused, by applying this rewrite rule:

$$\text{MapStm}(f) \triangleright \text{MapStm}(\text{Repeat}<N>) \Rightarrow \text{MapStm}(f \triangleright \text{Repeat}<N>) \quad (6.17)$$

The following expression is generated, which already gives an idea that the repetition moves towards the counter:

$$\text{Counter}<0,1,N,N> \triangleright \text{MapStm}(\text{MapStm}(\text{Read}(\text{mem}))) \triangleright \text{Repeat}<N>) \quad (6.18)$$

$$\triangleright \text{MapStm}(\lambda \text{rowA}. \text{rowA} \triangleright \text{Repeat}<N>,\text{matrixB}) \triangleright \text{ZipStm} \triangleright \text{MapStm}(\text{ZipStm})$$
Now, to cause a repetition of memory addresses, the repetition must be moved across the `MapStm(Read(mem))`. This is achieved with a more generic rewrite rule that allows the `Repeat` to jump over *any* expression. It also introduces another nesting of `MapStm` to compensate for the extra dimension created by the repetition:

\[
\text{any} \triangleright \text{Repeat}<N> \\
\quad \quad \quad \implies \text{Repeat}<N> \triangleright \text{MapStm(any)} \quad (6.19)
\]

The automatic rewriting process finally returns the following optimised expression:

\[
\text{Counter}<0,1,N,N> \triangleright \text{MapStm(Repeat}<N> \triangleright \text{MapStm(MapStm(Read(mem)))))} \quad (6.20) \\
\quad \triangleright \text{MapStm(λ rowA. (rowA,matrixB) \triangleright \text{ZipStm} \triangleright \text{MapStm(ZipStm))}
\]

This example has shown how rewriting gradually moves a `Repeat` primitive in the IR until it eventually reaches a counter. The compiler is able to generate efficient hardware for such a combination of counters and repeat modules. The address generation no longer simply increments its value, but jumps back to the beginning of the current row, whenever a repetition is desired. The corresponding VHDL template is shown in Appendix A.2.4 and is able to produce a valid address value each clock cycle. This way, the repetition problem is solved statically, during compile-time, and run-time pipeline stalls, as depicted in Figure 6.6c, disappear. This leads to improved overall performance, as will be evaluated in Section 7.4. While only three rewrite rules were applied here, the SHIR compiler provides more than 20 rules to generically optimise repetition for other scenarios as well.

### 6.4.2 Transposition

This section introduces and optimises transposition, which is used to implement *tiling*. If the input data for matrix multiplication or convolutional layers is too large for the FPGA’s on-chip buffers, tiling is applied. The large input matrix is partitioned into several small tiles, which are then processed one by one.

Using the `Split` primitive, horizontal slices are easily extracted from matrices. Full tiling, however, also requires these slices to be split into vertical chunks. For this, the `Split` primitive is applied on the *transposed* matrix. With this simple trick, tiling with \(M\) by \(N\) tiles is expressed as:

\[
\text{Map(Split}<N>\triangleright \text{Transpose} \triangleright \text{Map(Split}<M>\triangleright \text{Transpose) \quad (6.21)\n\]}
Chapter 6. Optimisation Using Rewrite Rules

To implement transposition, which is a special case of permutation, the PermuteVec primitive is first introduced:

\[
\text{PermuteVec}<\left(\mathbb{N} \rightarrow \mathbb{N}\right) > : \mathbb{B}[^TL] \rightarrow \mathbb{B}[^TL]
\]

(6.22)

It operates on a vector of \(N\) elements by applying a statically known permutation function \(\mathbb{N} \rightarrow \mathbb{N}\) to manipulate the order of elements based on their indices.

Now, with this new primitive, transposition of 2D vectors is expressed by first flattening the \(M \times N\) input matrix using JoinVec, then permuting it with a more complex function and finally splitting it again:

\[
\text{TransposeVec} := \text{JoinVec} \triangleright \text{PermuteVec}<\lambda i. i/N + (i \mod N) \cdot M> \triangleright \text{SplitVec}<M>
\]

Stream-based transposition builds on this but requires the 2D input data to be converted to vectors first:

\[
\text{TransposeStm} := \text{MapStm}(\text{StmToVec}) \triangleright \text{StmToVec} \triangleright \text{TransposeVec} \triangleright \text{VecToStm} \triangleright \text{MapStm}(\text{VecToStm})
\]

(6.24)

Problem Statement

As demonstrated, the required functionality for transposition is expressible based on PermuteVec. The generated hardware for this primitive results in several wire assignments, as depicted in Figure 6.7a. For large data structures, the wiring becomes too complex and requires more area than feasible for the FPGA.

Optimisation

In order to address this issue, a fixed-point iteration over the IR applies rewrite rules\(^1\) to express the transposition more efficiently. Similar to the repeat optimisation in Section 6.4.1, the transposition of data is first rewritten as a transposition of memory read addresses.

\(^1\)Rewrite rules for transposition optimisation designed and implemented by Tzung-Han Juang, co-author of [SJD22a, SJD22b].
6.4. Efficient Data Reshaping

(a) Naive transposition of 2D vectors based on the PermuteVec primitive and the permutation function $\lambda i. i/N + (i \mod N) \cdot M$.

(b) Optimised hardware to generate a transposed stream of 2D addresses. The complex wire mesh, as in Figure 6.7a, is no longer required.

Figure 6.7: Block diagrams for transposition of 2D data.
All the application’s input data eventually comes from some kind of memory. Therefore, each program contains a Counter and Read primitive somewhere in the IR to read this data. In the context of transposition, the initial program given to the compiler may contain the following expression:

\[ \text{Counter}\langle 0, 1, M, N \rangle \triangleright \text{MapStm}(\text{MapStm(Read(mem)))) \triangleright \text{TransposeStm} \]  

(6.25)

Note, that to simplify this example the placeholder TransposeStm is used, which actually consists of several primitives, as shown in expression (6.24). Given the above expression, which transposes the data, the following generic rewrite rule moves the TransposeStm further towards the Read’s input — the counter that generates the addresses:

\[
\text{MapStm}(\text{any}) \triangleright \text{TransposeStm} \Rightarrow \text{TransposeStm} \triangleright \text{MapStm}(\text{any})
\]  

(6.26)

As a result of this rewriting, the complex data reshaping operations are placed next to the address counter:

\[ \text{Counter}\langle 0, 1, M, N \rangle \triangleright \text{TransposeStm} \triangleright \text{MapStm}(\text{MapStm(Read(mem)))) \]  

(6.27)

Nevertheless, this expression still generates a complex wiring on the FPGA. The following rewrite rule replaces the transposition of addresses by a more efficient expression, based on two simple counters:

\[
\text{Counter}\langle C_0, C_S, M, N \rangle \triangleright \text{TransposeStm} \Rightarrow \text{Counter}\langle C_0, C_S, N \rangle \triangleright \text{MapStm}(\lambda p1. \text{Counter}\langle 0, N, M \rangle \triangleright \text{MapStm}(\lambda p2. (p1, p2) \triangleright \text{Add}))
\]  

(6.28)

The corresponding hardware design is shown in Figure 6.7b and has no more complex wiring left. After rewriting, this final, optimised expression is returned:

\[
\text{Counter}\langle 0, 1, N \rangle \triangleright \text{MapStm}(\lambda p1. \text{Counter}\langle 0, N, M \rangle \triangleright \text{MapStm}(\lambda p2. (p1, p2) \triangleright \text{Add})) \triangleright \text{MapStm}(\text{MapStm(Read(mem))))
\]  

(6.29)

### 6.4.3 Slide

Following the ideas of [HSS+18, SHS+19] around stencil computations, the Slide primitive is able to implement convolution. This primitive, first introduced in Figure 4.6, is generic over the window size $S_W$ and the step size $S_S$: Slide$\langle S_W, S_S \rangle$. 
For 2D convolution, the following combination of primitives expresses a 2D sliding window operation, where $S'_W$ is the height and $S''_W$ the width of the window:

$$\text{Slide}<S'_W,S''_S> \triangleright \text{Map} (\text{Transpose} \triangleright \text{Slide}<S''_W,S''_S> \triangleright \text{Map} (\text{Transpose})) \quad (6.30)$$

On the hardware-specific IR level, the slide primitive is refined to operate on streams (see Figure 4.9) and vectors (see Figure 4.11). With these primitives an efficient parallel convolution is expressed, as depicted in Figure 6.8. However, the implementation on FPGAs is anything but trivial.

**Problem Statement**

The hardware implementation of $\text{SlideStm}$ is based on a shift register, as depicted in Figure 4.10. This register emits its entire contents, the window vector, whenever a new input is received. The parallel variant $\text{SlideVec}$ creates a wiring mesh in hardware, similar to Figure 6.7a, but even more complex, because each input signal is connected to $S_W/S_S$ outputs in average. For large window sizes, either an enormous shift register or complex wiring is inferred, which may not fit onto the target FPGA. An optimisation is required to implement the elaborate slide operation on real hardware.
Optimisation

The main idea is similar to the optimisations for repetition and transposition, in Sections 6.4.1 and 6.4.2. Instead of reshaping data, the addresses of that data in memory are reorganised, which is more efficient. This is achieved by another rewrite-based fixed-point iteration\(^2\) over the IR. To describe this process on an example, the following initial expression with the sliding window operator on the Algorithmic Level is considered:

\[
\text{Input<} \text{IntT}(32), D_I > \triangleright \text{Slide<} S_W, S_S >
\]

(6.31)

After lowering to the Architecture Level, abiding by the rules in Figure 5.1, this expression is obtained:

\[
\text{Counter<} C_0, C_S, D_I > \triangleright \text{MapStm(Read(mem))} \triangleright \text{SlideStm<} S_W, S_S > \triangleright \text{MapStm(VecToStm)}
\]

(6.32)

The optimisation starts with moving the SlideStm and VecToStm primitives across the MapStm with the read operation. The generic rewrite rule for this does not only work for Read but also any other function inside MapStm:

\[
\text{MapStm(any)} \triangleright \text{SlideStm<} S_W, S_S > \triangleright \text{MapStm(VecToStm)} \implies \text{SlideStm<} S_W, S_S > \triangleright \text{MapStm(VecToStm)} \triangleright \text{MapStm(MapStm(any))}
\]

(6.33)

This rewriting returns an expression, where SlideStm operates on addresses instead of data:

\[
\text{Counter<} C_0, C_S, D_I > \triangleright \text{SlideStm<} S_W, S_S > \triangleright \text{MapStm(VecToStm)} \triangleright \text{MapStm(MapStm(Read(mem)))}
\]

(6.34)

Nevertheless, the generated circuit for this is still too complex in hardware. That is why another rewrite rule replaces the slide over addresses by a more efficient combination of counters that produces the same values:

\[
\text{Counter<} C_0, C_S, D_I > \triangleright \text{SlideStm<} S_W, S_S > \triangleright \text{MapStm(VecToStm)} \implies \text{Counter<} C_0, C_S, (D_I - S_W + C_S)/S_S + 1 \triangleright \text{MapStm(λp1.} \triangleright \text{MapStm(λp2.} \triangleright \text{Add})
\]

(6.35)

\(^2\)Rewrite rules for sliding window optimisation designed and implemented by Tzung-Han Juang, co-author of [SJD22a, SJD22b].
6.5. Maximising DMA Throughput

After these rewrite optimisations, the slide expression is removed entirely, and the following expression is obtained:

\[
\text{Counter}<C_0, S, (D_1 - S_W + C_S)/S_S + 1> \triangleright \text{MapStm}(\lambda p1. (6.36) \\
\text{Counter}<0, C_S, S_W> \triangleright \text{MapStm}(\lambda p2. (p1, p2) \triangleright \text{Add})) \\
\triangleright \text{MapStm}(\text{MapStm}(\text{Read}(\text{mem})))
\]

The generated hardware for the counters looks similar to the optimised transposition in Figure 6.7b, only the counters’ configurations are different. The large shift registers or wire meshes are in effect replaced by counters, reducing area while preserving the original functionality.

### 6.4.4 Summary

As seen in this section, data reshaping operations must be optimised to generate efficient hardware. The key idea is to move these operations across other primitives in the IR. This process brings the reshaping operations close to a counter, where they become more efficient or are optimised away.

### 6.5 Maximising DMA Throughput

Accelerators require both fast computation and fast communication to deliver high performance. Slow memory access can limit the performance of carefully parallelised hardware designs, as a chain is only as strong as its weakest link. This section presents two rewrite optimisations that increase throughput when directly accessing host memory.

#### 6.5.1 Concurrent Requests

The throughput of host RAM access via DMA depends heavily on the number of concurrently pending read and write requests. The more requests in the queue, the higher the throughput. As mentioned in Section 4.6, the length of `ReadAsync`’s input stream determines the maximum number of possible parallel requests. The `SHIR` compiler reshares the input stream with splits and joins and thus maximises the number of concurrent requests. This leads to a throughput close to the theoretical maximum, as the evaluation in Section 7.2 will show.
Chapter 6. Optimisation Using Rewrite Rules

Figure 6.9: The Alternate primitive with two provided functions. The input demux logic decides which function receives the input data, while the output mux logic alternately forwards the output data of one and then the other function. The signal flow of the two modes (red dashed lines for mode A, orange for mode B) is constantly switched back and forth in such a way that one function consumes the input, while the other one produces the output.

6.5.2 Input Double Buffering

A common strategy to improve throughput is double buffering, where filling and consuming operations alternate between two buffers. At any time, one of the buffers is busy receiving new data, while the other one is sending out the available data.

To exploit pipeline parallelism and simultaneously fetch new input data while running the computation, the Alternate primitive is proposed:

\[
\text{Alternate} : T^{\text{NonRamArray}} \rightarrow U^{\text{NonRamArray}} \rightarrow (T \xrightarrow{s} U) \xrightarrow{s} (T \xrightarrow{s} U) \xrightarrow{s} T \xrightarrow{s} U
\]

This primitive alternately feeds its input to one of its functions, while emitting the output of the other function, as visualised in Figure 6.9.

The rewrite optimisation for double buffering finds a single ReadAsync primitive in the IR and replaces it with this double-buffered implementation:

\[
\lambda \text{offset} . \text{offset} \triangleright \text{ReadAsync}(\text{rdCtrl}, \text{baseAddr}) \implies \lambda \text{offset} . \text{offset} \triangleright \text{Alternate}(\lambda \text{p1} . \text{p1} \triangleright \text{ReadAsync}(\text{rdCtrl}, \text{baseAddr}), \lambda \text{p2} . \text{p2} \triangleright \text{ReadAsync}(\text{rdCtrl}, \text{baseAddr}))
\]
Exploiting Device-Specific Resources

FPGAs often have specialised hardened functions that excel at efficiency and outperform similar logic-based implementations. In Section 4.7, the Mul2Add primitive is
introduced to perform two multiplications and one addition in a single DSP block on the Intel Arria 10.

However, Intel’s synthesis tool Quartus is only able to automatically exploit this feature under certain circumstances — although this optimisation may seem obvious to the user when looking at the IR. To enforce this efficient DSP block configuration, the compiler detects suitable multiply-add operations and makes the use of the Mul2Add primitive explicit. This optimisation is once again encoded as a rewrite rule:

\[
\text{input} \triangleright \text{MapVec}(\text{Mul}) \triangleright \text{SplitVec}_{2} \triangleright \text{MapVec}(\text{VecToTuple} \triangleright \text{Add})
\]

\[
\text{if } \text{bit width of elements in input} \leq 18 \quad \Rightarrow (6.40)
\]

\[
\text{input} \triangleright \text{SplitVec}_{2} \triangleright \text{MapVec}(\text{VecToTuple} \triangleright \text{Mul2Add})
\]

Due to hardware limitations, it is only possible to split up the DSP block into two separate multiplications when the input data width is less than or equal to 18 bits. For higher precision data, the entire DSP block is occupied. The condition of the rewrite rule captures this and restricts the application to cases where the input data width is sufficiently low.

Figure 6.10 sums up the advantages of this rewrite optimisation. Initially, the generated hardware allocates two separate DSP blocks for two multiplications and the addition is performed in programmable logic. The second multiplier and the hardened adder in each block remain unused. Due to the internal wirings of a DSP block, these wasted capabilities cannot be employed for other purposes, e.g., another multiplication.

After the rewriting optimisation, the IR explicitly states that two multiplications and one addition are performed in a single DSP block. This information is included in the generated hardware description so that the Quartus synthesis tool is able to allocate the hardware functions accordingly. Thus, an entire DSP block and a small amount of logic is freed up to be used for other operations.

In summary, the presented rewrite optimisation for the Mul2Add primitive demonstrates how device-specific features are easily exploited in the IR. As seen in the above example, the rewrite rule remains simple and short, yet its impact is great, as twice as many of the FPGA’s DSPs are made available.

These kinds of low-level optimisations assist the later synthesis process by taking hardware design decisions off the synthesis tool’s hands. The desired implementation choices are made explicit and inefficient designs are prevented from the very beginning.
6.7 Timing Correction

When a SHIR program is lowered, it gradually becomes more detailed and complex. A sequence of high-level primitives can easily generate long combinational paths with a large propagation delay in the synthesised FPGA design. Reduction trees, created by ReduceVec, are predestined to be affected by this issue. These large propagation delays either force a slowdown of the FPGA’s clock frequency, which is bad for performance, or worse, they result in faulty hardware operation. As explained in Section 2.1.5, pipelining can prevent this from happening. For this, the Registered primitive is introduced to the IR:

$$\text{Registered} : T^{\text{BasicData}} \rightarrow T \rightarrow T$$  \hspace{1cm} (6.41)

The type definition is similar to the identity function. However, during hardware generation, this primitive inserts a register that divides long signal paths into several shorter sections, allowing the synthesiser to meet the desired target clock frequency without violating timing constraints.

SHIR contains a heuristic-based rewrite rule that automatically detects reduction trees and inserts Registered expressions into the IR to form a pipeline after synthesis:

$$\text{SplitVec}<S> \triangleright \text{MapVec}(\text{Add})$$

$$\Rightarrow \text{SplitVec}<S> \triangleright \text{MapVec}(\text{Add} \triangleright \text{Registered})$$  \hspace{1cm} (6.42)
Figure 6.11 shows how the dataflow in a reduction tree is broken down into smaller parts after this rewrite optimisation. The heuristic is currently limited to detecting and optimising certain constructs only. Other combinations of primitives could still cause large propagation delays, unnoticed by this particular rewrite rule. In this case, additional rewrite rules are required, which are nevertheless easy to implement due to the practical \texttt{Registered} primitive.

Pipelining is an important technique in hardware design and, with the help of the \texttt{Registered} primitive, it is lifted to a high-level IR. This enables future extensions of \textsc{Shir}'s rewrite rules to insert registers, without the need to code in low-level HDL.

### 6.8 Informal Proof of Convergence

In order to keep the rewriting process simple and to prevent generating non-optimal programs, the rewrite system in \textsc{Shir} is designed to be convergent, see Section 2.3. This allows to take a set of rewrite rules and apply them to an expression as much as possible in arbitrary order, while the rewriting process is guaranteed to terminate and to yield the optimised design. The presented optimisations in this chapter consist of either a single rewrite rule, in which case the proof of convergence is trivial, or a set of rules which must guarantee termination and confluence.

The general idea in these rewriting optimisations is to move certain expressions either towards their data source (counters) or away from them. In the tree representation of an expression, this corresponds to downward or upward movements. Most of the rules presented aim in the direction of the counters, with the exception of the rules for \texttt{VecToStm} primitives in Section 6.3. It is obvious that such a process will always terminate, because the IR is a finite tree and therefore free from loops. For convergence, it is sufficient to show that each fixed-point iteration is free from conflicting rewrite rules which can both be applied to a given piece of IR. This is achieved by carefully designing the rewrite rules so that there is \textit{only one} way to move an expression downwards (or upwards) in the IR tree.

For a more formal proof, a monotone measure function $\varphi : \text{expr} \mapsto \mathbb{N}$ is defined to map an expression to a natural number. Each individual rewrite rule $t_0 \Rightarrow t_1$ must now guarantee to (monotonically) decrease this value, so that $\varphi(t_0) > \varphi(t_1)$. If this is given, an infinite rewrite sequence is impossible because this would induce an infinitely decreasing natural number.
More precisely, for the presented optimisations, the measure function $\varphi$ represents the distance of the currently considered expression to its closest leaf node in the IR tree, in case the expression is to be pushed towards its data source. Each rewrite operation decreases the distance and when the leaf of the IR is reached, the process ends, since no rewrite is able to further decrease the value of $\varphi$. This proof works equally for moving an expression up in the IR.

### 6.9 Summary

This chapter has demonstrated how common techniques to improve performance, like parallelisation and buffering, are automatically applied in the SHIR framework. A convergent rewrite system performs these optimisations in a fixed-point iteration, which terminates when the rules have been applied as much as possible. The stream and vector types facilitate exploring different levels of parallelisation, while the memory-aware IR with its ramarray type leverages memory-related improvements in the design.

The rewritings can cause data reshaping issues, which have a negative impact on the performance. However, it has been shown that they can be optimised away by applying further rewrite rules. Other data reshaping challenges are also addressed to achieve high-performance implementations for use-cases that involve repetition, permutation and sliding windows, such as matrix multiplication and convolution.

Regarding device-specific optimisations, the rewrite rule for the $\text{Mul2Add}$ primitive exemplifies how specialised low-level functions on an FPGA are exploited in a high-level representation. By making certain design decisions explicit in the IR, the synthesis process is guided towards implementations that perform better.

All the presented rules do not claim to be exhaustive. They cover important improvements for the considered main use-cases in this thesis, i.e., matrix multiplication and convolution. However, the framework is designed to be extensible and this chapter has shown how little coding is required to define a rewrite rule with yet great impact on the design.
Chapter 7

Evaluation

This chapter evaluates the proposed novel techniques of the SHIR framework. In particular, the previously introduced optimisations are applied and their impact on the generated hardware is discussed. The rest of this chapter is organised as follows.

First, Section 7.1 describes the test procedure and the environment in which the benchmarks are executed. Section 7.2 deals with rewrite rules to optimise DMA throughput of host memory and onboard memory\(^1\), and measures it for a selection of memory-bound workloads. Then, Section 7.3 shows how rewrite rules for buffering and parallelisation improve the performance of matrix multiplication, a typical workload for accelerators. All the data reshaping optimisations from Section 6.4 are evaluated in Section 7.4. To demonstrate their effect, more complex benchmarks based on tiled matrix multiplication and tiled 2D convolution\(^2\) are run. In Section 7.5, the performance of SHIR and OpenCL HLS generated hardware designs are compared with each other. Finally, Section 7.6 concludes this chapter.

7.1 Experimental Setup

The experiments are run on an Intel Arria 10 GX FPGA at 200 MHz with 1518 DSP blocks. This FPGA is widely deployed in the cloud and found in industry use-cases due to its large number of resources. In the experimental setup, it is connected via PCIe Gen 3 x8 to a host machine with two 18 core Intel Xeon Gold 6154 CPUs at 3 GHz and 512 GB DDR4 RAM at 2666 MHz. The bitstream for the FPGA is synthesised with Quartus Prime Version 19.2 from the VHDL files generated by the SHIR compiler.

\(^1\)Implementation of the VHDL template for onboard memory contributed by Ayan Chakraborty.

\(^2\)Benchmarks for 2D convolution written and results collected by Tzung-Han Juang, co-author of [SJD22a, SJD22b].
The timing requirements for the produced designs are met, unless stated otherwise. For example, some experiments are expected to be ‘not synthesisable’ due to disabled optimisations in the compiler, as we will see. All the FPGA applications operate on randomly generated input data sets, while the results are verified against a reference CPU implementation.

The evaluation is centred around a few well-known classes of applications on a real FPGA system, rather than taking the easier route of simulating a variety of benchmarks. A simulator exhibits a deterministic behaviour hiding many possible timing issues with the generated hardware. For example, physical timing delays caused by long signal paths on the FPGA do not occur in simulation. In contrast, real systems receive data from off-chip RAM via DMA. Due to traffic on the bus, this data comes in random order and at random time, leading to ever new circumstances for the accelerator. Furthermore, the hardware implementation must comply with certain design guidelines to be feasible for an FPGA at all, while a simulator can assume an arbitrary platform.

The Shir compiler generates the VHDL code for each benchmark in under five minutes, which is insignificant compared to the synthesis time which can take multiple hours. As a preparation for the experiments, the software side loads the input data, e.g., matrices, from binary files on hard drive into host RAM and arranges it to fit the cache line width. A cycle counter on the FPGA measures the end-to-end runtime, which includes the initial input data transfer to the FPGA, as well as the transfer of the results back to host RAM.

**Design Space Exploration**

The optimisation process in Shir is based on simple heuristics. All the presented optimisations are enabled by default. The design is parallelised as much as possible, buffers are inserted whenever sufficient on-chip memory is available and data reshaping operations are optimised if present.

In order to show the performance impact of certain optimisations, various points in the design space are targeted, by manually disabling different rewrite rules in the compiler. In ‘normal’ compiler operation, where all rules are enabled, this manual interaction with the rules is not required. The high-level hardware-agnostic expression is the only input provided by the user. Shir is able to generate even more different designs for the benchmarks. However, they are omitted, because they are not in the Pareto frontier with desirable area vs. throughput trade-offs.
7.2 Communication via DMA

The experiments in this section evaluate the throughput when transferring data via DMA. First, a memcopy-like SHIR program is run to determine the best transfer speeds possible between host and FPGA. Then, two basic stencil computations that rely on high memory throughput are executed. Finally, the maximum access speeds of onboard memory are measured.

Host Memcopy

The memcopy program copies data from host RAM to the FPGA and back. It is expressed on the Algorithmic Level with an \texttt{Id} expression and the input specification, which determines the amount of data to copy:

\begin{equation}
\text{Input<} \texttt{IntT}(512), 8388608\texttt{>} \triangleright \texttt{Id} \tag{7.1}
\end{equation}

The 8,388,608 cache lines of 512 bits result in a total of 512 MB of input data. The SHIR compiler automatically lowers the algorithmic expression into the Hardware Memory Level. To show how the maximum number of concurrently pending memory read requests via DMA affects memory throughput, the rewrite rules in the compiler flow are manually modified to explore weaker design points with fewer concurrent read requests. The experiments cover the range of 1 to 64 concurrent requests, because the Intel Arria 10 FPGA has a hardware limit of 64. Furthermore, the impact of double buffering the input in block RAM as introduced in Section 6.5.2 is also considered.

The results of this benchmark are presented in Figure 7.1. As expected, the best performance is achieved when the maximum number of concurrent requests is 64 and on-chip double buffering is enabled. The area usage for this optimal configuration is shown in Table 7.1 as experiment number 1. Furthermore, two interesting details can be observed with the memcopy benchmarks.

First, the effectiveness of double buffering: Using \texttt{N} concurrent requests with double buffering performs significantly better than using \texttt{2N} concurrent requests without double buffering, although these two designs have the same overall limit of pending concurrent requests.

Second, a tendency towards memory bandwidth saturation: The throughput for memcopy with double buffering increases by $\sim 1.7 \times$ from 16 concurrent requests to 32 concurrent requests. For the next doubling of concurrent requests, the throughput is only increases by $\sim 1.3 \times$. With a throughput of 6.5 GB/s, the best memcopy experiment
is close to the PCIe interface’s theoretical maximum of 7.875 GB/s at the physical layer. The remaining performance gap is due to PCIe protocol overheads and the mixed read and write access in the memcopy experiment. Other DMA benchmarks [NAZ+18] show similar speeds of up to 6.25 GB/s for DMA with mixed read and write access.

All the following experiments in this chapter use 64 concurrent requests to maximise memory throughput and on-chip double buffering to exploit pipeline parallelism for data fetching and computation.

**Stencil Computation**

The capability of SHIR to generate applications that utilise high-speed DMA is further evaluated on two simple stencil computations (experiments 2 and 3). Both of them operate on input matrices of $1024 \times 128$ 8-bit integers. Since these operations are memory bound, the efficiency is determined by comparing their throughput to the best performing memcopy experiment from above. The first stencil computation (exp. no. 2) is a 2D convolution with a $3 \times 3$ kernel and constant weights. On the Algorithmic Level in SHIR, basic 2D convolution is expressed as follows:

\[
\text{Input<} \text{IntT}(8), 128, 1024> \triangleright \text{Slide<}3, 1> \triangleright \text{Map}(\lambda \text{rowGroup}. \\
\text{rowGroup} \triangleright \text{Transpose} \triangleright \text{Slide<}3, 1> \triangleright \text{Map}(\lambda \text{window}. \\
\langle \text{window} \triangleright \text{Transpose, weights} \rangle \triangleright \text{Zip} \triangleright \text{Map}(\text{Zip}) \triangleright \text{Join} \\
\triangleright \text{Map}(\text{Mul}) \triangleright \text{Reduce}(\lambda a. \lambda b. \langle a, b \rangle \triangleright \text{Add, ConstInt<0>})) \quad // \text{dot product}
\]
7.2. Communication via DMA

Table 7.1: Performance and logic (ALM), RAM and DSP usage for memcopy and stencil computations on input matrices of $1024 \times 128$ 8-bit integers.

<table>
<thead>
<tr>
<th>exp. no.</th>
<th>operation</th>
<th>Performance</th>
<th>Resource Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>throughput</td>
<td>ALM</td>
</tr>
<tr>
<td>1 Memcopy</td>
<td>6.5 GB/s</td>
<td>100%</td>
<td>8%</td>
</tr>
<tr>
<td>2 2D Convolution</td>
<td>6.4 GB/s</td>
<td>99%</td>
<td>18%</td>
</tr>
<tr>
<td>3 2D Jacobi</td>
<td>6.4 GB/s</td>
<td>99%</td>
<td>8%</td>
</tr>
</tbody>
</table>

* The ratio of the experiment’s read and write speeds compared to that of memcopy.

The SHIR compiler automatically buffers some rows of the input data, so that each row has only to be read once in the entire runtime. The computation in this example is simple enough to fully parallelise it for each output element. Hence, the transposition and slide optimisations presented in Sections 6.4.2 and 6.4.3 are not yet required and separately evaluated later in Section 7.4. Table 7.1 shows the results. With a throughput of 6.4 GB/s the design generated by SHIR is as fast as memcopy, saturating the memory bandwidth. Larger input sizes exhibit the same performance.

Experiment 3 performs a single iteration of 2D Jacobi with a four-point stencil. The algorithmic SHIR code looks similar to above but the core computation on each window is modified to compute the average value of the four adjacent points. A combination of Join, Drop and Split extracts the four relevant values from the $3 \times 3$ window:

\[
\text{Input}<\text{Int}(8), 128, 1024> \triangleright \text{Slide}<3, 1> \triangleright \text{Map}(\lambda \text{rowGroup} . (7.3) \\
\text{rowGroup} \triangleright \text{Transpose} \triangleright \text{Slide}<3, 1> \triangleright \text{Map}(\lambda \text{window} . \\
\langle \text{window} \triangleright \text{Join} \triangleright \text{Drop}<1, 0> \triangleright \text{Split}<2> \triangleright \text{Map}(\text{Drop}<0, 1>) \triangleright \text{Join} \\
\triangleright \text{Reduce}(\lambda a . \lambda b . \langle a, b \rangle \triangleright \text{Add}, \text{ConstInt}<0>), \\
\text{ConstInt}<4> \triangleright \text{Div}))
\]

Again, SHIR inserts input row buffers to maximise the performance. The generated hardware does not employ any DSPs, because the algorithm only divides by 4. As described in Section 5.1, this is automatically rewritten as a bit shift operation by the compiler. Nevertheless, the design generated by SHIR is very efficient, because it reaches a throughput similar to that of memcopy, as shown in Table 7.1.
Onboard Memcopy

The memcopy program is modified to read data from and then write it back to onboard RAM. Since each memory bank has only a single shared channel for reading and writing, these types of access are measured separately. Both single bank and interleaved dual bank memory configurations are evaluated. Various quantities of data from 128 B to 2 MB are transmitted. All other parameters, such as the maximum number of concurrent requests, are kept uniform across all tests. The results of this experiment are summarised in Figure 7.2. As expected, the dual bank mode reaches approximately double the speed of a single bank. Moreover, the throughput of dual bank onboard RAM is \(3\times\) faster than host RAM.

Nevertheless, onboard memory has some disadvantages, especially in comparison to on-chip RAM. First, maximum speed is only reached after a substantial amount of data (\(~32\) KB) is transferred. Second, despite the speedup over host RAM, onboard RAM is still significantly slower than block RAM. As we will see in Section 7.4, the on-chip memories in the tiled matrix multiplication experiments deliver 4096 8-bit elements per clock cycle. At 200 MHz, this corresponds to a throughput of \(~760\) GB/s and is therefore \(~30\times\) faster than the best onboard memory transfer speed.

This is why the onboard RAM is not further used in the following experiments.
7.3 Buffering and Parallelisation

However, future work can benefit from it with workloads that involve large amounts of intermediate data, such as DNNs. In these use-cases, using onboard RAM instead of host RAM as an intermediate buffer can bring a speed advantage. For example, onboard memory can hold the result of a DNN layer, while the FPGA is reconfigured for the next layer.

7.3 Buffering and Parallelisation

This section demonstrates the effectiveness of buffering techniques and increasingly parallel computations. Both of these optimisations are achieved automatically in the SHIR compilation by applying the rewrite rules introduced in Sections 6.1 and 6.2. The basis of this evaluation is matrix multiplication, a demanding workload, which leaves many design choices for interesting optimisations.

Matrix Multiplication

For the matrix multiplication benchmarks, the input matrices consist of $1024 \times 1024$ 8-bit integers. It is assumed here that the matrix $B$ is already transposed on the host to simplify the expression. The matrix multiplication experiments are listed in Table 7.2 and are based on the following Algorithmic Level expression, which is the only user input required by the SHIR compiler:

\[
\text{Input}\langle\text{IntT}(8), 1024, 1024\rangle \triangleright \text{Map}(\lambda \text{rowA}. \quad \text{// matrix } A) \\
\text{Input}\langle\text{IntT}(8), 1024, 1024\rangle \triangleright \text{Map}(\lambda \text{colB}. \quad \text{// matrix } B) \\
\langle \text{rowA, colB} \rangle \triangleright \text{Zip} \\
\quad \triangleright \text{Map}(\text{Mul}) \triangleright \text{Reduce}(\lambda a. \lambda b. (a, b) \triangleright \text{Add}, \text{ConstInt<0>})) \quad \text{// dot product}
\]

The experiments 4-7 in Table 7.2 show how different buffering strategies, based on the rewrite rules from Section 6.2, affect the performance and on-chip RAM usage. In order to avoid artificial slowdown in the computational part, one cache line is processed in parallel, so that no time-consuming vector to stream conversion occurs. That is why the computation is parallelised by 64, which corresponds to the number of 8-bit elements in a single cache line.

In experiment 4, no buffering is employed. If data is repeated in the computation, it has to be read from host RAM again. This leads to $2N^2$ matrix rows being transferred via DMA. Each row consists of 1024 elements and therefore $1024/64 = 16$ cache lines. The total number of cache lines read in this experiment is $2 \cdot 1024^2 \cdot 16 = 33,554,432$. 
Table 7.2: Experiments for matrix multiplication of two $1024 \times 1024$ matrices with 8-bit integers. Different buffering strategies and levels of parallel computation affect the performance: Host RAM read requests via DMA, Giga Operations Per Second (GOPS) ($10^9$ multiply-add per second), Operations Per Cycle (OPC) (parallel multiply-add per cycle) and DSP usage efficiency.

<table>
<thead>
<tr>
<th>exp. no.</th>
<th>buffers</th>
<th>par. comp.</th>
<th>DMA reads</th>
<th>GOPS</th>
<th>OPC</th>
<th>DSP* efficiency</th>
<th>ALM</th>
<th>on-chip DSP</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>A</td>
<td>64</td>
<td>33,554K</td>
<td>1.4</td>
<td>7</td>
<td>11.2%</td>
<td>9%</td>
<td>3%</td>
<td>2%</td>
</tr>
<tr>
<td>5</td>
<td>✓</td>
<td>64</td>
<td>16,793K</td>
<td>1.6</td>
<td>7</td>
<td>12.3%</td>
<td>9%</td>
<td>3%</td>
<td>2%</td>
</tr>
<tr>
<td>6</td>
<td>✓ ✓</td>
<td>64</td>
<td>16,793K</td>
<td>1.7</td>
<td>8</td>
<td>13.0%</td>
<td>9%</td>
<td>20%</td>
<td>2%</td>
</tr>
<tr>
<td>7</td>
<td>✓ ✓</td>
<td>64</td>
<td>32K</td>
<td>12.7</td>
<td>63</td>
<td>98.9%</td>
<td>10%</td>
<td>20%</td>
<td>2%</td>
</tr>
<tr>
<td>8</td>
<td>✓ ✓</td>
<td>128</td>
<td>32K</td>
<td>25.1</td>
<td>125</td>
<td>98.2%</td>
<td>11%</td>
<td>20%</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>✓ ✓</td>
<td>256</td>
<td>32K</td>
<td>49.4</td>
<td>247</td>
<td>96.5%</td>
<td>15%</td>
<td>20%</td>
<td>8%</td>
</tr>
<tr>
<td>10</td>
<td>✓ ✓</td>
<td>512</td>
<td>32K</td>
<td>95.7</td>
<td>478</td>
<td>93.5%</td>
<td>22%</td>
<td>20%</td>
<td>17%</td>
</tr>
<tr>
<td>11</td>
<td>✓ ✓</td>
<td>1024</td>
<td>32K</td>
<td>173.7</td>
<td>868</td>
<td>84.8%</td>
<td>36%</td>
<td>20%</td>
<td>34%</td>
</tr>
</tbody>
</table>

* DSP efficiency is the proportion of the overall runtime in which the DSPs are active, i.e., OPC per allocated DSPs. With fewer idle cycles, the DSPs work more efficiently. At 100% efficiency they produce a new value each cycle and never have to wait for input data.

Experiment 5 applies the rewrite rule explained in Section 6.2. The row of matrix $A$ is buffered, which reduces the number of cache lines read by ~50% to 16,793,600 for $N^2+N$ rows. There is no significant performance improvement in terms of GOPS, because the matrix $B$ still has to be read from slow host RAM.

For experiment 6, the entire matrix $B$ is buffered with a similar result to the previous experiment. However, the on-chip RAM usage increased to 20%, because the buffer for the matrix requires 1024 times more memory than a row buffer.

In experiment 7, the buffer for the rows of matrix $A$ and the buffer for matrix $B$ can finally leverage the performance (~9× better than experiment 1). The number of rows to read is decreased to 2N, which corresponds to 32,768 cache lines.

In these experiments, no tiling is necessary because the entire input data fits in the FPGA’s on-chip memory. Later in Section 7.4, we will look into use-cases that involve tiling.
7.4 Data Reshaping

This section presents more sophisticated workloads to demonstrate the effect of the data reshaping optimisations from Section 6.4. Both matrix multiplication and 2D convolution now apply tiling. This allows larger matrices to be handled. In addition, the 2D convolution has multiple input and output channels, as found in typical convolutional neural networks. Although the rewrite rules are evaluated only on these two applications, they are applicable to any application expressed as a combination of dense array operations with reshaping.
Table 7.3: Generated tiled matrix multiplication designs with different rewrite optimisations enabled. The input matrix consists of $4096 \times 4096$ 8-bit integers. The tile size is $512 \times 2048$. The generated hardware uses 2048 of 3036 available multipliers.

<table>
<thead>
<tr>
<th>exp. no.</th>
<th>Rewrites</th>
<th>Performance</th>
<th>Resource Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trans-</td>
<td>GOPS</td>
<td>OPC</td>
</tr>
<tr>
<td></td>
<td>Repeat</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stm-Ve</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Conv.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>✓</td>
<td>0.3</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>✓ ✓</td>
<td>81.9</td>
<td>409</td>
</tr>
<tr>
<td>15</td>
<td>✓ ✓ ✓</td>
<td>290.9</td>
<td>1455</td>
</tr>
</tbody>
</table>

**Tiled Matrix Multiplication**

In tiled matrix multiplication, the input tiles are processed separately and accumulated directly on the FPGA to form a tile of the resulting matrix. The optimisation with the Mul2Add primitive from Section 6.6 allows to perform 2048 parallel multiplications with only 1024 DSP blocks.

Table 7.3 lists the corresponding experiments, with different rewrite rules enabled. The optimisations are applied automatically, if not explicitly disabled. Experiment 12 shows, that with all the rewrite rules disabled, the generated design cannot be synthesised for the FPGA. The stream-based or vector-based transpositions still remaining in the IR either produce too large on-chip buffers, or too complex wiring. The Transpose rewrite rule must be enabled to generate feasible designs, as the following experiments 13–15 show. The performance of the next experiment (no. 13) suffers from the repeat problem, as described in Section 6.4.1. The efficiency of the DSPs is low. Less than 1% of the overall runtime is actually used to perform useful operations on a DSP. The remaining time is spent waiting for input data to arrive due to pipeline stalls, as seen in Figure 6.6c, due to the need to back propagate the repeat signals. In experiment 14, all the rewrite rules to optimise repetitions are enabled. This leads to a few hundred times increase in performance, with a DSP efficiency of 20%.

When all optimisations are enabled, the DSP efficiency of experiment 15 increases by $3.5 \times$ compared to experiment 14, so that the DSPs are used in 71% of the overall runtime. Future work will look at increasing efficiency further, by using double-buffering to overlap tile loading with computation.
Table 7.4: Generated tiled 2D convolution designs with different rewrite rules. The input image has 1024×1024 8-bit integers. The tile size is 128×128. The kernel weights are 3×3 with 3 input and 64 output channels.

<table>
<thead>
<tr>
<th>Exp. no.</th>
<th>Slide</th>
<th>Transpose</th>
<th>Repeat</th>
<th>Stm-Vect</th>
<th>Conv.</th>
<th>Performance</th>
<th>Resource Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>not synthesisable!</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>not synthesisable!</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>3%</td>
<td>3% 17% 6% 28%</td>
</tr>
<tr>
<td>19</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>3%</td>
<td>3% 17% 6% 28%</td>
</tr>
<tr>
<td>20</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>76%</td>
<td>80% 14% 6% 28%</td>
</tr>
</tbody>
</table>

Tiled 2D Convolution

In this section, the data reshaping rewrite rules are evaluated on tiled 2D convolution. The results are shown in Table 7.4. The performance is assessed based on the memory throughput, since this benchmark is memory bound. It is specified as a percentage in comparison to the maximum throughput achieved by memcopy (6.5 GB/s).

The first two convolution experiments (16 and 17) show that both the Slide and Transpose optimisation must be enabled to generate synthesisable designs. Again, without these rules the design would require too much on-chip memory or cause too complex wire routing than feasible for the FPGA.

Once the rules are applied, as in experiments 18–20, the generated hardware design is able to run on the FPGA. With more rewrite rule optimisations enabled, the throughput increases from 3% up to 76% of its maximum possible value.

The introduction of the repeat rewrite rules do not have a noticeable effect on the performance of a parallelised convolution, because only vectors are repeated here, which does not induce much overhead. However, enabling the conversion optimisation does increase performance by 25×.
7.5 Comparison With OpenCL HLS

In this section, the performance of the generated hardware implementations from SHIR and Intel’s OpenCL HLS are compared with each other. All these benchmarks run on the same Intel Arria 10 FPGA. The results are shown in Figure 7.4.

Two versions are implemented for Intel’s HLS tool: A naive parallel OpenCL code, written in a hardware-agnostic way and an optimised OpenCL code with hardware-specific directives (pragmas) as well as explicit local memory usage. Neither SHIR’s Algorithmic Level code nor the naive OpenCL code contain any explicit unrolling, buffers, or other hardware-related optimisations. Still, SHIR’s designs outperform the naive OpenCL versions by up to \( \sim 100 \times \), as shown in Figure 7.4. This is because SHIR automatically introduces hardware optimisations by applying rewrite rules.

Stencil Computation

The optimised OpenCL code for 2D convolution and 2D jacobi implements the techniques mentioned in [HCGL15, JZ16], such as unrolling and local buffering. To further improve the parallelism and the pipeline efficiency, optimal pragmas and optimisation flags are chosen. In addition, the memory access is improved by tuning the cache settings in the compiler. Some communication overhead is observed in OpenCL HLS for small input sizes of about 128 KB. To ensure fairness in the comparison, it is increased to 2 GB, which leads to better OpenCL HLS results.
7.6 Summary

After exploring all the above-mentioned optimisations, the best OpenCL HLS designs achieve 4.6 GB/s for 2D convolution and 5.2 GB/s for 2D jacobi. The Shir code is truly hardware-agnostic and much more compact in comparison to the ~20 Lines of Code (LOC) of the optimised OpenCL code. Moreover, the performance of the Shir generated hardware outperforms the optimised OpenCL version by up to ~1.4×.

Matrix Multiplication

The optimised OpenCL matrix multiplication implementation comes from Intel and is the same as shown in Section 2.1.6. This time, the generated hardware performs as good as the Shir version. However, with ~30 significant LOC this OpenCL code is more verbose compared to the truly hardware-agnostic Shir input expression.

7.6 Summary

This chapter has demonstrated the viability of Shir for high-performance accelerator generation. The presented experiments are prime examples indicating the potential of a multi-level functional approach with rewrite rules.

The generated hardware designs exhibit high access speeds for off-chip and on-chip memories. They explore different levels of parallelisation and buffering strategies. The performance of workloads that involve sliding windows and tiling is improved by applying the data reshaping rewrite rules.

The performance of relevant use-cases has been evaluated on the basis of their throughput or number of operations per second. The memory-bound tiled 2D convolution achieves 76% of the maximum possible throughput thanks to Shir’s rewrite optimisation. The compute-bound tiled matrix multiplication also manages to achieve high performance after applying rewrite rules. Its best configuration uses 51% of the on-chip memory and 67% of the available DSPs, which perform valid computations in 71% of the runtime. With the efficient utilisation of so many compute resources, the Shir approach shows its ability to make good use of what the targeted FPGA has to offer. The scope for improvement over this is small considering that the memory transfer of the input and output data is included in the measurement and that FPGA resource utilisation over ~80% becomes increasingly difficult due to routing congestion.

The competitiveness of Shir is furthermore shown in a direct comparison with OpenCL HLS. Shir achieves equal or better performance although its input specification is free from hardware-specific details and higher-level than OpenCL code.
Part III

Conclusions
Chapter 8

Conclusions

This chapter summarises the contributions of this thesis, then discusses its limitations and finally gives an outlook for future work based on the obtained results.

8.1 Summary of Contributions

This thesis has presented the Shir framework with its multi-level functional IR and rewrite rule-based optimisations for higher-level synthesis of accelerators. The goal of Shir is to offer the best of two worlds: high-level hardware-agnostic abstractions for developers and high-performance hardware accelerator implementations. In order to achieve this, the following contributions have been made.

A Multi-Level IR Structure for Designing Hardware Accelerators

Based on System $F_{\rightarrow}$, the IRs designed for Shir are similar to types lambda calculus with generics and subtyping. Subtyping allows to easily extend the type hierarchy of Shir. As seen in this thesis, the IRs span over multiple levels of abstraction. Each IR fulfils a specific purpose in the compiler flow and exposes different features. The various concerns in hardware design are separated to maintain a clean organisation of the compiler. For example, algorithm and optimisations are handled separately in different places in the compiler. The IR levels are detailed in the following.

First, Shir’s higher-level approach starts on the Algorithmic Level. It serves as a hardware-agnostic entry point for the developer or high-level framework, where no hardware expertise is required. The intermediate language on this level consists of common primitives, such as Map, Reduce and Add. Although they are few, they are very flexible to combine and can be nested to express the desired algorithm.
This functional approach avoids any detour via traditional software-like representations, which are not well suited for hardware design.

The Architecture Level is the next lower abstraction, which exposes architectural features explicitly. The core types are extended by hardware specific types. Basic data types have an attached bit width. Furthermore, streams and vectors allow to express sequential pipelining and spatial parallelism on the type level. The primitives are refined accordingly for streams, e.g., MapStm, and vectors, e.g., MapVec.

Uniquely and in contrast to other approaches, see Section 3.6, SHIR exposes memory operations explicitly in the IR. This allows to express and optimise memory-related hardware details conveniently on high levels of abstraction. There are two IR levels in SHIR that deal with memory. First, the Abstract Memory Level, where memory is handled in a simple manner and similarly to conventional programming languages. It is allocated, read and written, respectively with the corresponding primitive: MemAlloc, Read and Write. A novel ramarray type represents data in the three major memories present in FPGA systems: Host, onboard and block RAM. Second, the Hardware Memory Level, where memory is expressed as functions, reusing existing mechanisms in the framework, such as shared functions with Let and function calls. These functions are either synchronous for on-chip memories or asynchronous for off-chip memories, which is encoded in the function type as an effect. Asynchronous communication via DMA features concurrent read or write requests to maximise throughput. This is implemented in the ReadAsync and WriteAsync primitives.

The lowest functional IR level deals with device-specific functions, specific to certain FPGAs. For example, the DSP blocks of the Intel Arria 10 contains two multipliers and an adder, which is represented by the Mul2Add primitive and can hence be made explicit in the IR to exploit the full potential of these efficient hardwired functions.

**Mechanical Lowering Procedures From Algorithm to Hardware**

The next contribution consists of mechanical lowering procedures, necessary to automatically translate a piece of IR without user input through multiple levels of abstraction. These procedures are implemented in SHIR’s compiler. As demonstrated, they replace expressions with their more refined counterparts to approach a hardware design step by step. No performance optimisations are carried out yet. This concern is handled separately by rewrite rules. The multi-level IR structure simplifies the long lowering process by breaking it down into a few smaller passes.

Data that is transferred between FPGA and off-chip memories is automatically
converted, *i.e.*, packed and padded, into a format based on cache line granularity. Once the lowest functional representation is reached, the expression is turned into a hierarchical dataflow graph. Nodes represent the primitives and consume and produce data. The transmission of this data is synchronised via a handshaking protocol. In case of shared resources, such as memory, arbiters are automatically inserted to distribute the permission to access it among the clients. Finally, the graph is translated into a hardware design based on VHDL files. For each node in the graph, the corresponding VHDL template is instantiated. An essential feature of these templates is their flexible composability. Although they are fine-grain and their individual function is rather simple, their composition allows to also achieve complex behaviour. The generated HDL code establishes the connections to the off-chip memory interfaces of the FPGA.

**Rewrite Rules for Optimisation**

Separated from SHIR’s IRs and compiler, this thesis contributes low-level hardware optimisations encoded in the form of rewrite rules sets. Each set is applied as much as possible in an automatic fixed-point iteration, making the given expression converge to the improved variant, while maintaining the original program semantics. The rewrites are applied on the IR level that best suits the particular optimisation. For instance, the fine-tuning for hardware-specific memory access is performed on a low-level IR. The optimisation process is taken over by SHIR, so that no user input is required.

A rewrite rule for spatial parallelisation has been presented. It replaces stream-based operators by their vector-based counterparts to leverage available compute resources on the FPGA for higher performance. When data reusage occurs in the IR, for example due to unbound parameters, a rewrite rule can detect this and insert a fast buffer for that data. These optimisations introduce conversions between streams and vectors, which present performance bottlenecks. In turn, these are also fixed by further rewrites.

A functional approach for HLS is challenging in the presence of data reshaping operations, such as repetition, transposition and slide. If not handled carefully, such operations can lead to incorrect designs, non-synthesisable designs or mediocre performance, as we have seen. This thesis has shown how all these reshaping challenges are solved through the introduction of an explicit Repeat primitive and through the application of rewrite rules. These rules optimise the design by moving the concerned primitives in the IR closer to the counters, where they are more efficient or even annihilated altogether.
Communication with off-chip memories is rewritten to maximise DMA throughput by increasing the number of concurrent requests and leveraging double buffering. Another device-specific rewrite rule maps two multiplications into a single DSP block, if possible due to low precision data. This accommodates the synthesis tool, which is sometimes unable to detect this optimisation opportunity. Large propagation delays, caused by reduction trees for example, are prevented by a rewrite rule that inserts registers into the signal path.

Evaluation on a Real FPGA

The Shir approach has been evaluated on real hardware, using an Intel Arria 10 FPGA. This contrasts with some related work, which simulates their hardware designs only, as discussed in Section 3.6. The Shir framework with its multi-level functional IR has proven viable for generating efficient accelerator implementations.

First of all, memcpy experiments show that the framework delivers high-speed off-chip memory communication. Further central use-cases are stencil computations, matrix multiplication and 2D convolution. Although being easy to understand, matrix multiplication exhibits interesting design choices that exercise many optimisation features presented in this thesis. Designs with different buffering strategies and levels of parallelisation have been explored.

Moreover, the effect of the rewrite rules on the identified data reshaping challenges has been analysed in a series of experiments based on tiled matrix multiplication and tiled 2D convolution workloads. They show that the performance increases drastically when applying the presented optimisations. In addition, we have seen that without these optimisations, the designs require too many resources to be synthesised for the FPGA. As demonstrated, the generated hardware designs are able to effectively exploit the available hardware features (e.g., 67% of the DSPs) of the FPGA and are competitive with OpenCL HLS implementations, although Shir starts at a much higher-level programming abstraction.

8.2 Critical Analysis

The previous section has summarised this thesis and its contributions towards automatically generating optimised hardware accelerators from a multi-level functional IR. Nevertheless, there are also limitations to this approach, as discussed in this section.
8.2. Critical Analysis

**Limited Set of Use-Cases**

First of all, the experiments conducted in this thesis are limited in number. The main experiments for matrix multiplication and convolution present relevant use-cases for accelerators and provide interesting optimisation opportunities but they do not cover every type of today’s workloads and the challenges they entail.

The evaluation of SHiR could have gone in one of two directions: Supporting a wide range of use-cases, however, at the expense of performance, because more rewrite rules for further types of optimisations are required. Or, picking a few ones and addressing their individual optimisation challenges, showing that competitive performance is achievable with SHiR.

The author of this thesis opted for the latter direction, because delivering high performance for a specific task is the actual primary purpose of an accelerator. Nevertheless, the evaluated use-cases demonstrate the generality of the SHiR approach. Other workloads could be supported as well but require some engineering efforts, since SHiR is in a research prototype state and debugging hardware designs is tedious.

**Limitations Within the IRs**

The IRs in SHiR have a few limiting factors affecting their flexibility. For example, they lack general recursion, narrowing down the set of supported input programs. However, this does not pose a practical limitation, since recursive algorithms can be converted to iterative ones.

In addition, certain combinations of types are prohibited by the type checker, because they would cause difficulties in the compilation. In the course of SHiR’s development, it was found that there are essentially two different interpretations of tuples. First, basic tuples, similar to records in VHDL, which simply concatenate the individual bits of their elements. The elements of such a BasicTuple\(T\) must be basic data, see Figure 4.8, and therefore instantaneously readable. In hardware, the ingoing elements are synchronised and a tuple element with only one outgoing handshake port is produced. This way, modules handle basic tuples just like scalars.

Second, a high-level tuple, as found in common functional programming languages and more flexible, which allows combining arbitrary elements. Tuples of streams are supported in SHiR, however, creating a tuple of two functions, e.g., \(\langle \lambda x. x, \lambda y. y \rangle\), is not. While the functional representation of this construct is possible, it has major consequences on the hardware side. The module that calls one of these functions only
must be directly wired to it, without affecting the other function. SHIR’s implementation of tuples is not sufficient for this because some control signals are getting lost in the tuple’s synchronisation, after which only one handshake port remains.

This issue could be addressed by a sophisticated compiler pass that rewrites the problematic tuples in the IR and essentially removes them, or a pass that generates the correct wiring on a the dataflow level. Neither of these exist in the current version of SHIR, and this is why dual port block RAM with fully independent ports cannot be expressed in the most suitable representation, as a tuple of functions.

Apart from that, there is another restriction when nesting types, as mentioned in Section 4.4. SHIR’s notion of vectors is that they are instantaneously accessible and therefore they must contain basic data only. While it is technically possible to allow streams within vectors on the type level, the implications on the hardware level are vague. It is unclear, how the individual streams in a vector would be synchronised, if at all, and when exactly a vector of streams would be produced or consumed. To avoid this problem, SHIR’s type system is designed to prevent vectors of streams right from the start. However, this design decision requires a workaround to express a sliding window operator over a two-dimensional stream, which temporarily has a vector of streams in its type before the operation is optimised away.

Guarantees for Propagation Delays

The feasibility of a generated design depends on whether the timing constraints of the FPGA are met. Especially in hardware generation it is child’s play to generate hardware with complex combinational logic resulting in too long propagation delays.

Section 6.7 introduced a simple rewrite optimisation to shorten the delays in reduction trees. Nevertheless, there are still ways to cause timing issues with other primitives, which are not caught by this rewrite rule. Therefore, the current version of SHIR does not guarantee that the generated HDL code works for any clock frequency.

A radical solution for this issue could insert registers around each primitive to reduce the propagation delay to a minimum. However, this comes at the expense of performance and area utilisation. It remains desirable to use registers only where they are really necessary.
8.3 Future Work

Based on this thesis and the developed SHIR framework, many new directions open up for further research work, which is detailed in this section.

Expanding the Scope of Use-Cases

The two main use-cases in this thesis, matrix multiplication and convolution, are certainly important but future work can take a look at other workloads as well. As stated in [JBB+17], CNNs are a minority as far as neural networks in data centres are concerned. LSTMs [HS97] in particular are Recurrent Neural Networks (RNNs) which are used about six times as often.

The author of [Ola15] describes how RNNs can be expressed in a functional style based on an ‘accumulating map’, also known as the ‘prefix sum’ or ‘scan’. In order to support RNN-based workloads in SHIR, such a primitive must be added, including its hardware implementation.

More Diverse Hardware Targets

This thesis evaluates SHIR on an FPGA from Intel, however, in the future devices from Xilinx can also be targeted. The development efforts required for this is mainly engineering work, as some of the hardware templates have to be adjusted to the new interfaces of the FPGA.

Apart from single-FPGA solutions, infrastructures with multiple FPGAs as in Project Brainwave [CFO+18, FOP+18] offer exciting starting points for further work. FPGA boards often come with a dedicated high-speed network interface allowing the FPGA chip to connect directly to a massive data centre network, without going through the host machine. Larger machine learning workloads, e.g., DNNs, could then be partitioned across FPGAs residing in multiple servers. From the software point of view, this pool of FPGAs appears as callable, shared hardware microservice [CFO+18].

Supporting multiple FPGAs in SHIR brings interesting challenges. A new abstract representation is needed to reflect the devices, allowing to explore the optimisation space of mappings between tasks and FPGAs. Rewrite rules could modify this IR to improve the partitioning of the workload. Especially in a heterogeneous environment with different FPGA models, there is a lot of room for variation.
Following this extension of the IR, Shir could also introduce a representation of the host machine, similar to the one in Lift-hls [KBSD19]. That way, parts of the calculation that require more flexibility could be run on the CPU.

**Guiding Design Space Exploration**

As the number of Shir’s use-cases and hardware targets grows, so does the design space, yielding many different ways to apply the optimisation rewrite rules. Simple, hard-coded heuristics, as currently implemented in Shir, would no longer be the tool of choice to find optimal designs, while logic synthesis remains too time-consuming for a design space exploration.

A performance model could provide an early estimate whether the implementation of a given expression would perform well on hardware. This model prioritises throughput, latency or even energy-efficiency, depending on the workload and the requirements. Its outcome would guide the application of rewrite rules towards a high-performance design. Due to the high semantic content of functional parallel patterns, the model would remain lightweight and run fast. The primitives could be linked almost one-to-one with the resulting performance. The multi-level IR structure allows (multiple) performance models to operate on different levels of abstraction.

Furthermore, the search space could be pruned by detecting and removing invalid implementations at an early stage. The authors of [MLRD22] present a method for this based on extracting constraints from an IR and then solving a constraint satisfaction problem to identify valid rewrites. In the future, these ideas can be integrated into Shir to reduce the effort of design space exploration and to obtain high-performance implementations more quickly.

**Interfacing With Other Frameworks**

Shir’s representations are meant to be intermediate forms in a larger design process. Even its highest level of abstraction, the algorithmic level, can be targeted from other front ends. Future work could map the ONNX format [BLZ+19] to algorithmic parallel patterns, thus connecting Shir to high-level deep learning frameworks, such as TensorFlow [ABC+16], MXNet [CLL+15] and PyTorch [PGM+19].

Recently, PyTorch 2.0 has introduced the new PrimTorch component to facilitate adding a custom backend. This component lowers complex PyTorch programs to a reduced set of elementary primitives. Shir could implement this operator set in a
8.3. Future Work

high-level IR, enabling PyTorch as a frontend.

To interact even more with other frameworks, Shir could be integrated into the unified MLIR infrastructure. The multiple IR levels of Shir would then be implemented as dialects, somewhat similar to [LSS21], and the lowering passes would become transformations in MLIR. Since the above mentioned high-level deep learning frameworks already have their specific dialect in MLIR, they could serve Shir as front ends. On the other side, the novel concepts of this thesis would become reusable and contribute to an expandable infrastructure.

Exploiting Reduced Precision

Recent research has already demonstrated that neural networks with low-precision data are highly efficient with only a slight loss in accuracy [BPF+18]. The memory footprint of the network’s weights is lower, reducing the slow, energy-intensive communication to off-chip memories. Additionally, the implementation of arithmetic operations becomes easier.

XNOR-Net [RORF16] is rigorous by reducing the precision of the network’s data to one bit. In such a Binary Neural Network (BNN), the weights are either +1 or -1, occupying significantly fewer resources on the FPGA than networks with high-precision data. Convolutions are approximated by XNOR and bitcounting operations. With an accuracy drop of ImageNet from 57% to 44%, the implementation of [RORF16] is $58\times$ faster and has $32\times$ memory savings.

The IRs in Shir expose the precision of data and support arbitrary bit-widths. However, exploiting extremely low precision in neural networks, assessing the accuracy loss and performance gains remains future work.
Appendix A

VHDL Templates

The main contributions in this thesis focus on SHIR’s high-level IRs, its compiler and its rewrite rules for optimisation. However, an important part of the framework are the VHDL templates that implement the high-level expressions for FPGAs. All the available VHDL templates in SHIR are listed in Appendix A.1. After that, some example template files are shown in more detail.

A.1 Overview

The following table presents the names of all the VHDL template files developed for SHIR and their associated Architecture Level primitives.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Corresponding Primitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>add.vhd</td>
<td>Add</td>
</tr>
<tr>
<td>alternate.vhd</td>
<td>Alternate</td>
</tr>
<tr>
<td>arbiter_async_function.vhd</td>
<td>(arbiter for shared async functions)</td>
</tr>
<tr>
<td>arbiter_distributor.vhd</td>
<td>(distributor for shared data)</td>
</tr>
<tr>
<td>arbiter_sync_function.vhd</td>
<td>(arbiter for shared sync functions)</td>
</tr>
<tr>
<td>block_ram.vhd</td>
<td>BlockRam</td>
</tr>
<tr>
<td>concat_stm.vhd</td>
<td>ConcatStm</td>
</tr>
<tr>
<td>concat_vec.vhd</td>
<td>ConcatVec</td>
</tr>
<tr>
<td>const_bit_vector.vhd</td>
<td>ConstInt</td>
</tr>
<tr>
<td>counter.vhd</td>
<td>Counter</td>
</tr>
<tr>
<td>drop_stm.vhd</td>
<td>DropStm</td>
</tr>
<tr>
<td>drop_vec.vhd</td>
<td>DropVec</td>
</tr>
<tr>
<td>File Name</td>
<td>Corresponding Primitive</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>id.vhd</td>
<td>Id</td>
</tr>
<tr>
<td>join_stm.vhd</td>
<td>JoinStm</td>
</tr>
<tr>
<td>join_vec.vhd</td>
<td>JoinVec</td>
</tr>
<tr>
<td>map_stm.vhd</td>
<td>MapStm</td>
</tr>
<tr>
<td>map_vec.vhd</td>
<td>MapVec</td>
</tr>
<tr>
<td>mul2add.vhd</td>
<td>Mul2Add</td>
</tr>
<tr>
<td>mul.vhd</td>
<td>Mul</td>
</tr>
<tr>
<td>onboard_ram.vhd</td>
<td>OnboardRam</td>
</tr>
<tr>
<td>permute_vec.vhd</td>
<td>PermuteVec</td>
</tr>
<tr>
<td>read_async.vhd</td>
<td>ReadAsync</td>
</tr>
<tr>
<td>read_host_memory_controller.vhd</td>
<td>ReadHostMemCtrl</td>
</tr>
<tr>
<td>read_onboard_memory_controller.vhd</td>
<td>ReadOnboardMemCtrl</td>
</tr>
<tr>
<td>read_sync.vhd</td>
<td>ReadSync</td>
</tr>
<tr>
<td>read_sync_memory_controller.vhd</td>
<td>ReadSyncMemCtrl</td>
</tr>
<tr>
<td>reduce_stm.vhd</td>
<td>ReduceStm</td>
</tr>
<tr>
<td>registered.vhd</td>
<td>Registered</td>
</tr>
<tr>
<td>repeat.vhd</td>
<td>Repeat</td>
</tr>
<tr>
<td>slide_stm.vhd</td>
<td>SlideStm</td>
</tr>
<tr>
<td>slide_vec.vhd</td>
<td>SlideVec</td>
</tr>
<tr>
<td>stm_to_vec.vhd</td>
<td>StmToVec</td>
</tr>
<tr>
<td>sub.vhd</td>
<td>Sub</td>
</tr>
<tr>
<td>tuple.vhd</td>
<td>Tuple, known as (&lt;e_1, e_2&gt;)</td>
</tr>
<tr>
<td>vec_to_stm.vhd</td>
<td>VecToStm</td>
</tr>
<tr>
<td>vec_to_tuple.vhd</td>
<td>VecToTuple</td>
</tr>
<tr>
<td>write_async.vhd</td>
<td>WriteAsync</td>
</tr>
<tr>
<td>write_host_memory_controller.vhd</td>
<td>WriteHostMemCtrl</td>
</tr>
<tr>
<td>write_onboard_memory_controller.vhd</td>
<td>WriteOnboardMemCtrl</td>
</tr>
<tr>
<td>write_sync.vhd</td>
<td>WriteSync</td>
</tr>
<tr>
<td>write_sync_memory_controller.vhd</td>
<td>WriteSyncMemCtrl</td>
</tr>
<tr>
<td>zip_stm.vhd</td>
<td>ZipStm</td>
</tr>
<tr>
<td>zip_vec.vhd</td>
<td>ZipVec</td>
</tr>
</tbody>
</table>
A.2 Examples

This section now showcases a selection of SHiR’s VHDL templates.

A.2.1 Add

The template for the Add primitive only uses combinational logic to add the two integer values of the incoming tuple. The control signals of the input port are simply forwarded to the output. The last signals are unused because only a scalar is transmitted, not a stream. Nevertheless, they are kept in the handshake ports for interoperability.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity add is
port(
  clk: in std_logic;
  reset: in std_logic;
  port_in_data: in data_tuple_type; -- type is replaced by code generator
  port_in_last: in std_logic_vector(-1 downto 0);
  port_in_valid: in std_logic;
  port_in_ready: out std_logic;
  port_out_data: out data_type; -- type is replaced by code generator
  port_out_last: out std_logic_vector(-1 downto 0);
  port_out_valid: out std_logic;
  port_out_ready: in std_logic
);
end add;

architecture behavioral of add is
begin
  port_out_data <= std_logic_vector(unsigned(port_in_data.t0) + unsigned(port_in_data.t1));
  port_out_last <= port_in_last;
  port_out_valid <= port_in_valid;
  port_in_ready <= port_out_ready;
end behavioral;
```

A.2.2 Mul2Add

The VHDL implementation for the Mul2Add primitive must be aligned to the actual hardwired DSP circuit on the FPGA. If the hardware description does not match this circuit, the synthesis tool cannot infer the DSP and the operation must be implemented in basic logic blocks instead. This is why the template for Mul2Add contains a pipeline with four stages, separated by intermediate registers. These registers keep...
the propagation delay through the complex DSP circuit short and allow high clock frequencies for the FPGA. They make up the greater part of the signal definitions and assignments in the template.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mul2add is
  port(
    clk: in std_logic;
    reset: in std_logic;
    port_in_data: in data_tuple_tuple_type; -- type replaced by code generator
    port_in_last: in std_logic_vector(-1 downto 0);
    port_in_valid: in std_logic;
    port_in_ready: out std_logic;
    port_out_data: out data_output_type; -- type replaced by code generator
    port_out_last: out std_logic_vector(-1 downto 0);
    port_out_valid: out std_logic;
    port_out_ready: in std_logic
  );
end mul2add;

architecture behavioral of mul2add is
begin
  in_ready <= not stall_regs;
end mul2add;
```
A.2. Examples

```vhdl
port_in_ready <= in_ready;
port_out.data <= acc_out_reg;
port_out.last <= (others => '0');
port_out.valid <= acc_out_valid;
stall_regs <= '1' when acc_out_valid = '1' and port_out.ready = '0' else '0';
mul_logic: process(clk, reset)
begin
if reset = '1' then -- reset must be async here in order to exploit DSP output registers!
  mul_in1_reg1 <= (others => '0'); mul_in1_reg2 <= (others => '0');
  mul_in2_reg1 <= (others => '0'); mul_in2_reg2 <= (others => '0');
  mul_out_reg1 <= (others => '0'); mul_out_reg2 <= (others => '0');
  acc_out_reg <= (others => '0');
elsif rising_edge(clk) then
  if stall_regs = '1' then
    mul_in1_reg1 <= mul_in1_reg1; mul_in1_reg2 <= mul_in1_reg2;
    mul_in2_reg1 <= mul_in2_reg1; mul_in2_reg2 <= mul_in2_reg2;
    mul_out_reg1 <= mul_out_reg1; mul_out_reg2 <= mul_out_reg2;
    acc_out_reg <= acc_out_reg;
  else
    mul_in1_reg1 <= ('0' & port_in_data.t0.t0);
    mul_in2_reg1 <= port_in_data.t0.t1;
    mul_in2_reg1 <= port_in_data.t1.t0;
    mul_in2_reg2 <= port_in_data.t1.t1;
    mul_in1_reg2 <= mul_in1_reg1; mul_in1_reg2 <= mul_in1_reg2;
    mul_in2_reg2 <= mul_in2_reg1; mul_in2_reg2 <= mul_in2_reg2;
    -- two parallel multiplications
    mul_out_reg1 <= std_logic_vector(unsigned(mul_in1_reg2) * unsigned(mul_in2_reg2));
    mul_out_reg2 <= std_logic_vector(unsigned(mul_in2_reg1) * unsigned(mul_in2_reg2));
    -- adding the two products
    acc_out_reg <= std_logic_vector(unsigned(mul_out_reg1) + unsigned(mul_out_reg2));
  end if;
end if;
end if;
end if;
end process;
```

```vhdl
pipeline_logic: process(clk) -- propagate the valid signal through the pipeline stages
begin
if rising_edge(clk) then
  if reset = '1' then
    mul_in_valid1 <= '0'; mul_in_valid2 <= '0'; mul_out_valid <= '0'; acc_out_valid <= '0';
  else
    if stall_regs = '1' then
      mul_in_valid1 <= mul_in_valid1; mul_in_valid2 <= mul_in_valid2;
      mul_out_valid <= mul_out_valid; acc_out_valid <= acc_out_valid;
    else
      mul_in_valid1 <= '0';
      if port_in_valid = '1' and in_ready = '1' then
```
A.2.3 SlideStm

The SlideStm primitive is used in convolution workloads. As depicted in Figure 4.10 it produces a stream of vectors, where each vector is a window of the slide operation. This is due to its hardware implementation based on a shift register, visible in the VHDL code below.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity slide_stm is
  generic( -- generics are set by code generator
    window_width: natural := 4;
    step_size: natural := 2
  );
  port(
    clk: in std_logic;
    reset: in std_logic;
    port_in_data: in data_type; -- type replaced by code generator
    port_in_last: in std_logic_vector(0 downto 0);
    port_in_valid: in std_logic;
    port_in_ready: out std_logic;
    port_out_data: out data_window_vector_type; -- type replaced by code generator
    port_out_last: out std_logic_vector(0 downto 0);
    port_out_valid: out std_logic;
    port_out_ready: in std_logic
  );
end slide_stm;

architecture behavioral of slide_stm is
  type shift_reg_type is array(window_width - 2 downto 0) of data_type; -- type replaced by code generator
  signal elements: natural range 0 to window_width - 1 := 0;
  signal reg: shift_reg_type;
  signal out_valid: std_logic := '0';
  signal in_ready: std_logic := '0';
begin
```

A.2. Examples

```vhdl
combine_output_data_signal: process(port_in_data, reg)
begin
  for i in reg'low to reg'high loop
    port_out_data(i) <= reg(i); -- connect shift register to output
  end loop;
  port_out_data(port_out_data'high) <= port_in_data; -- forward current ingoing element
end process;

port_out_last <= port_in_last;
out_valid <= '1' when port_in_valid = '1' and elements = window_width - 1 else '0';
port_out_valid <= out_valid;
in_ready <= '1' when elements < window_width - 1 else port_out_ready; -- always ready to receive data,
  when shift register is not yet filled
port_in_ready <= in_ready;

shift_reg_logic: process(clk)
begin
  if rising_edge(clk) then
    if reset = '1' then
      else
        if port_in_valid = '1' and in_ready = '1' then
          for i in reg'low to reg'high - 1 loop
            reg(i) <= reg(i + 1); -- shift
          end loop;
          reg(reg'high) <= port_in_data; -- feed in next input
        end if;
      end if;
      end if;
    end if;
  end if;
end process;

element_counter_logic: process(clk)
variable elements_v: natural range 0 to window_width := 0;
begin
  if rising_edge(clk) then
    if reset = '1' then
      else
        elements_v := elements;
        if port_in_valid = '1' and in_ready = '1' then
          elements_v := elements_v + 1;
        end if;
        if out_valid = '1' and port_out_ready = '1' then
          if port_in_last = "1" then
            elements_v := 0;
          else
            elements_v := elements_v - step_size;
          end if;
        end if;
        elements <= elements_v;
      end if;
      end if;
    end if;
  end if;
end process;
```
A.2.4 Counter

The Counter primitive is used in particular to generate addresses for memory access. It is able to generate values in the form of a multi-dimensional stream, as defined by dimensions in the VHDL template. The configuration of the generics in the template below, for example, specifies a \( \mathbb{Z}[\mathbb{Z}[\mathbb{Z}[T]_3]_5]_2 \) stream type for the counter. Furthermore, the bit vector repetitions defines whether the corresponding dimension is a repetition only. This changes the output counter values, while the dimensions remain the same. Without repetitions, and all bits of repetitions set to zero, the counter_value in the VHDL code below counts up to \( 3 \times 5 \times 2 \times \text{increment} + \text{start} \). In case the outermost dimension is marked as a repetition, the value is reset, so that the module counts twice from \( \text{start} \) up to \( 3 \times 5 \times \text{increment} + \text{start} \). The flexibility of the template allows to efficiently implement all the counters needed for the data reshaping optimisations from Section 6.4.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity counter is
    generic( -- generics are set by code generator
        start: natural := 0;
        increment: natural := 2;
        dimensions: natural_vector_type := (2, 5, 3); -- read from right to left (innermost dimension to outermost)
        repetitions: std_logic_vector := "001" -- read from left to right (innermost to outermost repeat)
    );
    port(
        clk: in std_logic;
        reset: in std_logic;
        port_out_data: out data_type; -- type replaced by code generator
        port_out_last: out std_logic_vector(1 downto 0);
        port_out_valid: out std_logic;
        port_out_ready: in std_logic
    );
end counter;

architecture behavioral of counter is
    -- function to precompute constants (during compilation)
    function precomp_increments return natural_vector_type is
        variable increment_per_dimension: natural_vector_type := (others => increment);
    begin
        if repetitions(0) = '1' then
            if repetitions(0) = '1' then
```
A.2. Examples

```vhdl
increment_per_dimension(0) := increment;
else
    increment_per_dimension(0) := dimensions(0) * increment;
end if;
-- multiplication accumulate
for i in dimensions'low + 1 to dimensions'high loop
    if repetitions(i) = '1' then
        increment_per_dimension(i) := increment_per_dimension(i - 1);
    else
        increment_per_dimension(i) := increment_per_dimension(i - 1) * dimensions(i);
    end if;
end loop;
return increment_per_dimension;
end function;

signal counter_value: natural := 0;
signal counter_dimensions: natural_vector_type := (others => 0);
constant increment_per_dimension: natural_vector_type := precomp_increments;
signal out_last: std_logic_vector(port_out_last'range) := (others => '0');
signal out_valid: std_logic := '1';

begin

port_out_data <= std_logic_vector(to_unsigned(counter_value + start, port_out_data'length));
port_out_last <= out_last;
port_out_valid <= out_valid;

last_signals: process(counter_dimensions)
begin
    out_last <= (others => '0');
    for i in dimensions'low to dimensions'high loop
        if counter_dimensions(i) = dimensions(i) - 1 then
            out_last(i) <= '1';
        end if;
    end loop;
end process;

counter_dimensions_logic: process(clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            counter_dimensions <= (others => 0);
        else
            if out_valid = '1' then
                for i in dimensions'low to dimensions'high loop
                    if port_out_ready = '1' then
                        if counter_dimensions(i) < dimensions(i) - 1 then
                            counter_dimensions(i) <= counter_dimensions(i) + 1;
                        else
                            counter_dimensions(i) <= 0;
                        end if;
                    end if;
                end loop;
            end if;
        end if;
    end if;
end process;
```

counter_value_logic: process(clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            counter_value <= 0;
            out_valid <= '1';
        else
            if out_valid = '1' and port_out_ready = '1' then
                if repetitions(repetitions'low) = '0' or out_last(out_last'low) = '1' then
                    counter_value <= counter_value + increment; -- increase value
                else -- finished counting
                    counter_value <= 0;
                    if port_out_ready = '1' then
                        out_valid <= '0';
                    end if;
                end if;
            end if;
            -- repeat if corresponding bit in repetitions is set
            for i in dimensions'low to dimensions'high - 1 loop
                if out_last(i) = '1' and port_out_ready = '1' and repetitions(i + 1) = '1' and out_last(i + 1) = '0' then
                    counter_value <= counter_value + increment - increment_per_dimension(i);
                    exit;
                end if;
            end loop;
        end if;
    end if;
end process;


[BNZ+16] Bruno Bodin, Luigi Nardi, M Zeeshan Zia, Harry Wagstaff, Govind Sreekar Shenoy, Murali Emani, John Mawer, Christos Kotselidis, Andy Nisbet, Mikel Lujan, Björn Franke, Paul HJ Kelly, and Michael O’Boyle. Integrating algorithmic parameters into benchmarking and design space


Bibliography


[LGN+17] Yun Liu, Krishna Gadepalli, Mohammad Norouzi, George E Dahl, Timo Kohlberger, Aleksy Boyko, Subhashini Venugopalan, Aleksei Timofeev, Philip Q Nelson, Gregory S Corrado, Jason D Hipp, Lily Peng, and


[NV15] Syed Waqar Nabi and Wim Vanderbauwhede. Using type transformations to generate program variants for fpga design space exploration. In 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pages 1–6, 2015.


[Xil21] Xilinx. Vivado design suite user guide — high-level synthesis. https://www.xilinx.com/content/dam/xilinx/support/documents/sw...
Bibliography

