FPGAs for the Masses: Affordable Hardware Synthesis from Domain-Specific Languages

THÈSE N^{O} 7004 (2016)

PRÉSENTÉE LE 3 JUIN 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ARCHITECTURE DE PROCESSEURS
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Nithin GEORGE

acceptée sur proposition du jury:

Prof. M. Odersky, président du jury
Prof. P. Ienne, directeur de thèse
Prof. J. Anderson, rapporteur
Prof. K. Olukotun, rapporteur
Prof. J. Larus, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

To my parents

Abstract

Field Programmable Gate Arrays (FPGAs) have the ability to be configured into application-specific architectures that are well suited to specific computing problems. This enables them to achieve performances and energy efficiencies that outclass other processor-based architectures, such as *Chip Multiprocessors* (CMPs), *Graphic Processing Units* (GPUs) and *Digital Signal Processors* (DSPs). Despite this, FPGAs are yet to gain widespread adoption, especially among application and software developers, because of their laborious application development process that requires hardware design expertise. In some application areas, domain-specific hardware synthesis tools alleviate this problem by using a *Domain-Specific Language* (DSL) to hide the low-level hardware details and also improve productivity of the developer. Additionally, these tools leverage domain knowledge to perform optimizations and produce high-quality hardware designs. While this approach holds great promise, the significant effort and cost of developing such domain-specific tools make it unaffordable in many application areas. In this thesis, we develop techniques to reduce the effort and cost of developing domain-specific hardware synthesis tools. To demonstrate our approach, we develop a toolchain to generate complete hardware systems from high-level functional specifications written in a DSL.

Firstly, our approach uses language embedding and type-directed staging to develop a DSL and compiler in a cost-effective manner. To further reduce effort, we develop this compiler by composing reusable optimization modules, and integrate it with existing hardware synthesis tools. However, most synthesis tools require users to have hardware design knowledge to produce high-quality results. Therefore, secondly, to facilitate people without hardware design skills to develop domain-specific tools, we develop a methodology to generate high-quality hardware designs from well known computational patterns, such as `map`, `zipWith`, `reduce` and `foreach`; computational patterns are algorithmic methods that capture the nature of computation and communication and can be easily understood and used without expert knowledge. In our approach, we decompose the DSL specifications into constituent computational patterns and exploit the properties of these patterns, such as degree of parallelism, interdependence between operations and data-access characteristics, to generate high-quality hardware modules to implement them, and compose them into a complete system design. Lastly, we extended our methodology to automatically parallelize computations across multiple hardware modules to benefit from the spatial parallelism of the FPGA as well as overcome performance problems caused by non-sequential data access patterns and long access latency to external memory. To achieve this, we utilize the data-access properties of the computa-

tional patterns to automatically identify synchronization requirements and generate such multi-module designs from the same high-level functional specifications.
Driven by power and performance constraints, today the world is turning to reconfigurable technology (i.e., FPGAs) to meet the computational needs of tomorrow. In this light, this work addresses the cardinal problem of making tomorrow's computing infrastructure programmable to application developers.
Keywords— High-level synthesis, domain-specific languages, computational patterns, FPGA, reconfigurable computing

Résumé

Les *Circuits Logiques Programmables* (FPGA) sont uniques dans leur capacité d'être configurable en architectures dédiées à une application. Ces architectures sont donc plus adaptées à ces problèmes computationnels spécifiques, leur permettant ainsi d'atteindre des performances et des efficacités energétiques surclassant d'autres architectures basées sur des processeurs, tels que les *Microprocesseurs Multi-coeurs* (CMPs), les *Processeurs Graphiques* (GPUs), et les *Processeurs de traitement du signal* (DSPs). Malgré cela, les FPGA n'ont toujours pas bénéficié d'une grande adoption, particulièrement parmi les developpeurs logiciels. Ceci s'explique principalement par le cycle de developpement d'applications pour FPGAs laborieux, nécessitant souvent des connaissances élevées en développement matériel. Dans certains domaines applicatifs, l'usage d'outils de synthèse matériel dédiés au domaine allègent ce problème en utilisant des langages dédiés (DSL) afin de cacher les détails les plus bas-niveau du matériel, et permettent ainsi d'améliorer la productivité des développeurs. De plus, ces outils tirent profit des connaissances du domaine pour effectuer des optimizations et ainsi produire des designs matériels de haute-qualité. Bien que cette approche soit très prometteur, le coût, ainsi que l'effort significatif requis pour développer de tels outils, la rend inabordable dans beaucoup de domaines applicatifs. Dans cette thèse, nous développons des techniques qui aident à réduire le coût et l'effort requis pour développer des outils de synthèse matériel dédiés à un domaine. Pour démontrer notre approche, nous developpons une suite d'outils pour générer des systèmes matériels entiers à partir de spécifications de haut-niveau décrit dans un langage dédié.

Dans un premier temps, notre approche utilise l'intégration des langages et du "staging" dirigé par types pour développer un langage dédié et un compilateur de manière rentable. Afin de réduire d'avantage l'effort requis, nous développons ce compilateur en composant des modules d'optimization réutilisables, et en l'intégrant avec des outils de synthèse matériel existants. Cependant, la plupart des outils de synthèse matériel nécessitent que les utilisateurs aient des connaissances en design matériel pour produire des systèmes de haute-qualité. Donc, dans un deuxième temps, afin de faciliter le développement d'outils de synthèse matériel dédiés à un domaine pour les personnes ayant peu de connaissances en design matériel, nous développons une méthodologie permettant de générer des designs matériels de haute-qualité à partir de motifs computationnels bien connus tels que `map`, `zipWith`, `reduce` et `foreach`. Ces motifs computationnels sont des méthodes algorithmiques décrivant la nature des calculs et des communications, et peuvent façilement être compris, ne nécessitent aucune connaissance en design matériel. Dans notre approche, nous décomposons d'abord les spécifications décrits

par le langage dédié en ses motifs computationnels constituants. Ensuite, nous exploitons leurs propriétés telles que le degré de parallelisme, l'interdépendance entre les opérations, et les charactéristiques des accès aux données, pour ainsi générer des modules matériels de haute-qualité, et les composons en un système matériel complet. Dans un dernier temps, nous étendons notre méthodologie pour parallelizer, de manière automatique, les calculs sur plusieurs modules matériels afin de profiter du parallelisme spatiale disponible dans un FPGA, et additionellement, afin de surmonter les problèmes de performances liés à l'accès non-contigu aux données et à la latence élevée des mémoires externes. Pour y parvenir, nous utilisons les propriétés d'accès aux données propre aux motifs computationnels pour identifier automatiquement les conditions de synchronization nécessaires, et ainsi générer ces designs à modules multiples à partir des mêmes spécifications fonctionelles de haut-niveau décrits par le langage dédié.
Porté par des contraintes énergétiques et de performances, le monde d'aujourd'hui se tourne de plus en plus vers les technologies reconfigurables (comme les FPGA) pour satisfaire les besoins computationnels de demain. Dans cette perspective, cette thèse adresse un problème essentiel qui est de rendre l'infrastructure computationnelle de demain programmable aux developpeurs logiciels.
Mot-clés— Synthèse matériel à haut-niveau, langages dédiés, motifs computationnels, FPGA, calcul reconfigurable

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Paolo Ienne, for his guidance and support. He has been a great mentor. His invaluable feedback helped to improve my research methodology and enabled me to grow as a person.

I would like to thank the members of the thesis committee, Jason Anderson, James Larus, Kunle Olukotun and Martin Odersky, for both finding time to review my work, and for their insightful comments and suggestions that significantly improved the quality of this thesis.

I had the good fortune to collaborate with many people during my Ph.D. studies. I would like to especially thank HyoukJoong Lee whose comments and suggestions had a remarkable impact on my work, and Mikhail Asiatici for his help with some of the engineering work. I would like to express my heartfelt gratitude to Joao Andrade, David Novo, Tiark Rompf, Kimon Karras, Philip Brisk, David Andrews, Gabriel Falcao, Kunle Olukotun and Martin Odersky for their help on various publications, some of which are used in this thesis. My sincere thanks also goes to Vipin Kizheppatt, Kermin Flemming, Hsin-Jung Yang, Nehal Bhandari and Mohammad Shahrad for their help on various projects.

I would like to express my deep appreciation to Ross Kory, Ali Galip Bayrak, Ana Petkovska and Grace Zgheib for their comments on this manuscript, and to Sahand Kashani-Akhavan for translating the abstract of the thesis into French.

It has been a great pleasure for me to be a part of LAP (Processor Architecture Laboratory). I had a great time in the company of my fellow LAP member—Ajay, Ali, Ana, Andrew, Chantal, David, Grace, Hadi, Lana, Madhura, Mikhail, Mirjana, Nikola, Paolo, Philip, Rene, Robert, Theo, Xavier, many interns and visiting collaborators. I would like to especially thank Ali, Robert and Chantal for their friendship, guidance, encouragement and support throughout my journey at LAP.

My stay in Switzerland was extremely enjoyable, thanks to the excellent company of friends—Amir, Carlos, Christian, Denys, Francois, Felix, Manohar, Mark, Nada, Olga, Panagiotis, Sandro, Sebastian, Sepand, Thomas, Vagia, Vlad, Vojin and many others. I thank Manohar and Sandro for always being available for discussions, some of which had a direct impact on my work.

It would have been impossible to go through this journey without the support of my family. I am eternally grateful for their unconditional love and limitless support throughout my life.

Lausanne, April 18, 2016 N. G.

Contents

List of Figures

List of Tables

1 Introduction

FPGAs can efficiently tackle the data deluge

Today, we are living in an age where the way we generate wealth, network socially, search for information, conduct research and advance science have all become increasingly data-driven. Naturally, we have come to rely heavily on our computing facilities [Bell et al., 2009, Anderson, 2008] to process this data. In 2007, humankind stored about 2.9×10^{20} bytes of optimally compressed digital data (growing at a rate of 23% annually) and were scaling up the sizable computing infrastructure at a rate of 58% annually [Hilbert and López, 2011]. Now, the increasing cost of operating this massive and growing computing infrastructure has become a major concern [Koomey et al., 2009]. Microprocessors are at the heart of almost all of our computing infrastructure today. Yet, microprocessors have been shown to be quite lacking in energy efficiency compared to *Application-Specific Integrated Circuits* (ASICs). Additionally, researchers have shown that application-specific modifications to the architecture can considerably improve both the energy-efficiency and performance of microprocessors [Hameed et al., 2010]. However, architecture of the microprocessor was designed to be flexible and perform well for a wide variety of computing tasks for different applications. Hence, it is infeasible to make application-specific alterations to the processor architecture without affecting its suitability to handle this large range of workloads. *Field Programmable Gate Arrays* (FPGAs) are devices that were designed to be programmed into customized computing architectures. Furthermore, researchers have shown that FPGAs can implement application-specific architectures that achieve both high energy-efficiency and computing performance [Kestur et al., 2010, Betkaoui et al., 2010, Fowers et al., 2012]. The broad objective of this thesis is to make it easier to develop FPGA applications and make reconfigurable technology more accessible to users.

FPGA users need to have hardware design expertise

FPGAs are used in a number of application areas, including communication, aerospace and defense, automotive, consumer electronics and medicine, but often as ASIC replacements for low volume products or as prototyping platforms. Despite their potential as a high performance and energy efficient computing unit, they are seldom used within datacenters or other general purpose computing infrastructure [Putnam et al., 2014]. To a large extent, this is due to the complicated workflow for developing and implementing applications on an FPGA which is markedly different from *any* software development flow.

The application implementation workflow for FPGAs is illustrated in Figure 1.1. The first step, as shown in the figure, is to develop a hardware design at a *Register Transfer Level* (RTL) from high-level functional specifications. This is often a tedious manual process that can easily take a few months. During this step, in addition to the hardware design, one needs to write a test bench to verify correctness of the design and create constraint files that hold the timing and physical placement constraints needed to implement the design on the FPGA. Once the design is verified through simulation, the *synthesis* step compiles the RTL description into a gate-level netlist and maps these gates to the physical components on the FPGA, such as LUTs, FFs, BRAMs and DSPs. The subsequent *place and route* step finds physical placements for the components on the device and determines how the wires between these components can be routed on the FPGA's configurable routing fabric. Static timing analysis is performed on the placed and routed design to check if the design meets the timing constraints set by the user. If the timing constraints are satisfied, the *generate bitstream* step produces the bitstream file that is used to program the FPGA. This file contains the configuration for the logic and routing components to implement the hardware design on the device. This FPGA design implementation workflow bears a lot of similarity to the standard ASIC development flow and warrants a significant amount of hardware design expertise. Therefore, application developers without any hardware design background find it difficult, if not impossible, to develop FPGA applications.

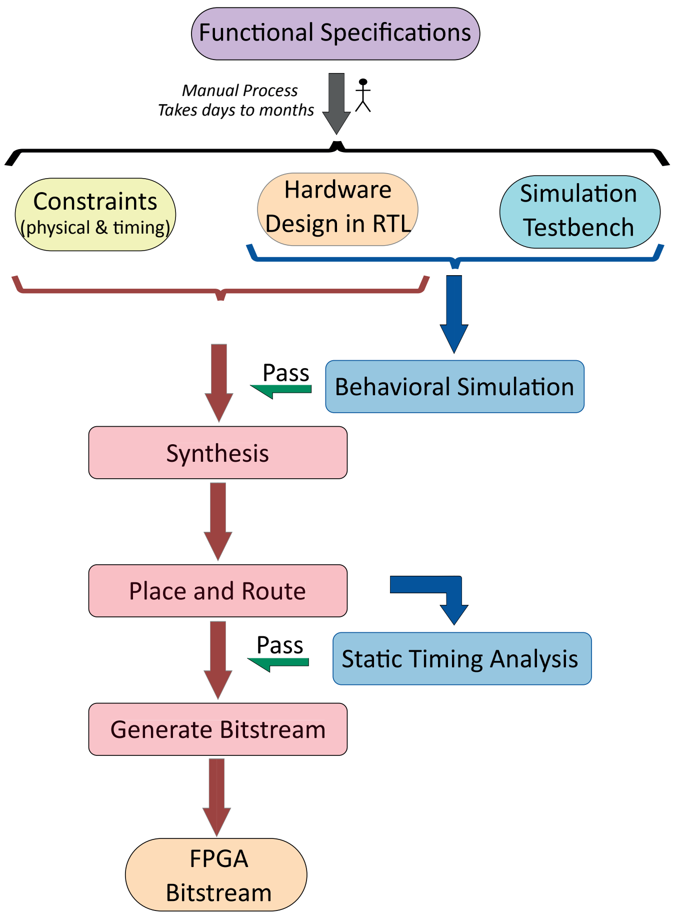


Figure 1.1: FPGA design development flow. In the typical FPGA design development flow, the user translates functional specifications into a hardware design, a tedious and error-prone process that can take months. Additionally, steps such as specifying the physical and timing constraints, verifying the design through simulation, and validating and correcting the implementation based on static analysis are all performed at a circuit-level. These aspects make this development flow extremely hard for users without hardware design expertise.

1.1 High-Level Synthesis

Developing the hardware design in RTL is perhaps the biggest obstacle that deters users without hardware design expertise from using FPGAs. Even for hardware design experts, developing the RTL specifications is a laborious task that limits their productivity and their ability to scale up to larger design sizes while coping with time-to-market pressures. In recent years, there has been a more concerted effort from the industry to tackle this problem and we have seen the emergence of new *High-*

HLS tool users also need hardware design knowledge

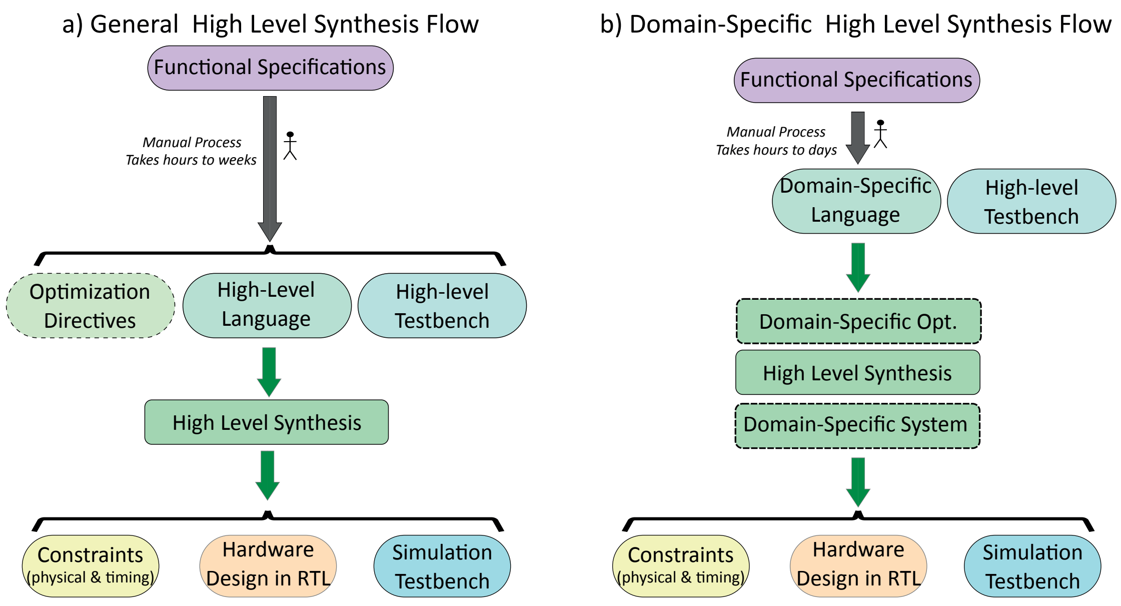


Figure 1.2: Design generation using HLS. a) *High-Level Synthesis* (HLS) tools enable users to generate hardware designs from functional specifications in high-level languages, such as C, C++, SystemC and OpenCL, and, therefore, requires less effort and time. b) Domain-specific synthesis tools focus on synthesizing hardware designs for specific application areas. They often use a *Domain-Specific Language* (DSL) for input specifications, perform optimizations based on the domain knowledge and create complete hardware systems for the specific application. This enables them to achieve improved productivity, better design quality and make designing hardware more accessible to users with no hardware design knowledge.

Level Synthesis (HLS) tools. These tools, as illustrated in Figure 1.2a, can generate hardware designs from high-level specifications written in languages such as in C, C++, SystemC and OpenCL, to significantly enhance the productivity of the user [Xilinx, 2013a, Czajkowski et al., 2012, Xilinx, 2015]. While most of these tools can produce hardware designs from high-level functional specifications, they still require the user to refactor the input specifications and provide additional optimization directives to produce good quality hardware designs. However, users require a considerable amount of hardware design expertise to specify these directives and, therefore, these tools are ineffective in enabling non-hardware-experts to develop FPGA applications.

Domain-specific synthesis tools can alleviate the need for hardware design expertise to use FPGAs

Some high-level synthesis tools, such as Spiral [Milder et al., 2012], HDL Coder [The MathWorks, 2015], Optimus [Hormati et al., 2008], and MMAlpha [Derrien et al., 2008] target specific application domains in which they make designing hardware more accessible to their users. As illustrated in Figure 1.2b, compared to other HLS tools domain-specific tools offer the following three important advantages:

1. They often use a *Domain-Specific Language* (DSL) for input specifications which provides a suitable syntax and abstraction to make it easier for domain experts to develop applications.
2. They leverage the detailed knowledge of the application domain properties to perform optimization and achieve improved quality of results.
3. Due to their domain specialization, these tools can automatically package the hardware designs they generate with necessary integration facilities (e.g., hardware interfaces, software *Application Programming Interface* (API) and drivers) to make it directly usable for the intended application.

Therefore, these tools hold great potential in making reconfigurable technology more accessible to users in different application domains. Despite these advantages, developing domain-specific tools incurs significant effort, as well as cost, and it makes this approach impractical in many application areas.

1.2 Objective

Our objective in this thesis is to develop a methodology to alleviate the cost and effort needed to develop domain-specific hardware synthesis tools. To demonstrate this methodology, we develop an infrastructure that can be used to generate complete hardware systems from high-level DSL applications.

1.2.1 Building Domain-Specific Hardware Synthesis Tools With Low Effort

Language embedding and type-directed staging can reduce domain-specific tool development cost

Developing a new domain-specific HLS tool is a significant effort because it often involves designing an entirely new DSL, compiler, development and debugging environments. However, if the new language is developed as an embedded DSL [Mernik et al., 2005] (i.e., the DSL is created by extending a host language by adding domain-specific language elements), it can share some of the infrastructure of the host language, such as type system, module system, development and debugging environments, and avoid repeating this effort. Additionally, we can use type-directed staging [Carette et al., 2009] to reduce the effort needed to develop a compiler for programs written in this new DSL. In our effort, we explore the benefits of this approach using Scala [Odersky et al., 2004] as the host language and *Lightweight Modular Staging* (LMS) [Rompf and Odersky, 2012] infrastructure for developing the compiler. These techniques have been successfully employed for software development [Sujeeth et al., 2014, Ofenbeck et al., 2013, Ackermann et al., 2012], and we investigate how they can be useful for developing domain-specific hardware synthesis tools [George et al., 2013]. Additionally, to reduce the repeated development effort for different toolchains, we propose to develop the tool in a modular fashion by creating reusable optimization modules; these modules, once developed, can be reused in a completely different toolchain with very little effort.

1.2.2 Generating Efficient Hardware Designs from Computational Patterns

Computational patterns can bridge the divide between high-level applications and high-performance designs

Developing new domain-specific tools will become appreciably easier if they can share a common infrastructure for generating hardware. Furthermore, to enable software developers and domain-experts, who might be developing domain-specific tools, to benefit from this infrastructure, it must not demand hardware design expertise from the user. After studying applications from different domains, researchers have determined that they often contain a small set of computational patterns that can be used to express these applications [Asanovic et al., 2006, McCool et al., 2012]. *Computational patterns* are simple algorithmic methods that capture a pattern of computation and communication which makes them easy to understand and use without any advanced knowledge. These computational patterns have well known properties, such as parallelism, dependency between operations or the nature in which they produce or consume data. There are compilation infrastructures that can leverage these properties to efficiently implement high-level applications on a variety of different platforms, including *Chip Multiprocessors* (CMPs), *Graphic Processing Units* (GPUs) and clusters [Sujeeth et al., 2014, Catanzaro et al., 2011]. Similar techniques can be used to map applications on FPGAs where the properties of the computational patterns can be exploited to produce efficient circuit structures to implement them [George et al., 2014]. For instance, computational patterns with a large amount of structured parallelism can leverage the spatial parallelism in the FPGA. Patterns with no parallelism can also benefit from architecture customization, but since these patterns often offer limited scope for acceleration, they can be implemented on a shared processor to conserve resources. When the application is written in a DSL, the domain knowledge can be utilized to perform optimizations as well as to select a suitable architecture template for the system-level integration. This can enable a tool to automatically generate complete hardware systems to implement the application.

1.2.3 Generating Multi-Module Hardware Designs from Computational Patterns

Computational patterns can enable automated tools to leverage the spatial parallelism of the FPGA

Parallel architectures such as CMPs, GPUs and FPGAs perform particularly well for applications that have sufficient parallelism to leverage all the processing resources on the device. To achieve good performance, the implementation must also ensure a high-bandwidth supply of data to keep the parallel processing resources busy. But, non-sequential data access patterns and long memory access latencies can often cause data starvation at the processing resources and significantly hamper the performance of applications. Parallelizing computation across multiple independent hardware modules is a way to tackle this problem and one that been successfully used in CMPs and GPUs. To achieve this on an FPGA, the application developer must correctly identify how shared data is accessed from different modules and use synchronization schemes to guard access to this data. This can be difficult for large applications and there can be issues such as false sharing [Bolosky and Scott, 1993] that make this process even harder. When an application is decomposed into computational patterns, these patterns capture the data access properties within the application. We leverage these properties to automatically identify when synchronization is needed among the modules and how these synchronization requirements can be relaxed [George et al., 2015]. Armed with this knowledge, a tool can automatically generate complete hardware designs where the computation is parallelized across multiple modules. Moreover, these designs employ a dynamic workload partitioning scheme to effectively leverage these modules and deliver performance improvements.

1.3 Outline

Outline The rest of the thesis is organized as follows.

- **Chapter 2** provides some background information on different HLS approaches and discusses related work.
- **Chapter 3** explores the idea of using language embedding and type-directed staging to reduce the effort needed to develop domain specific hardware synthesis flows. We will illustrate how these ideas can be applied to generate hardware circuits to implement linear algebra expressions.
- **Chapter 4** discusses how decomposing high-level applications into computational patterns can enable automatic generation of high performance hardware systems targeting FPGAs. To demonstrate the approach, we extend the Delite compiler infrastructure [Sujeeth et al., 2014, Lee et al., 2011] to develop a tool flow that generates complete hardware systems from high-level functional specifications in a DSL-program.
- **Chapter 5** extends the approach developed in Chapter 4 to automatically generate multi-module hardware systems from the same high-level DSL-programs. These hardware systems parallelize the computation across multiple hardware modules to deliver improved performance.
- **Chapter 6** concludes the thesis and presents some ideas for future work.

2 Background and Related Work

2.1 High-Level Synthesis Tools

Benefits of HLS tools

Developing hardware designs for FPGAs is an effort intensive process that can stretch into months. Therefore, there has been widespread interest in developing HLS tools to alleviate this development effort and make reconfigurable technology more accessible to application developers. HLS tools enable users to synthesize hardware designs from high-level specifications in languages such as C [Kernighan et al., 1988], C++ [Stroustrup, 1986], OpenCL [Stone et al., 2010], CUDA [NVIDIA Corporation, 2015] and Java [Arnold et al., 1996]. Compared to designing in RTL, the benefits of using HLS tools include the following:

1. *Productivity and scalability*: HLS tools improve productivity since they accept input specifications at a higher level of abstraction and generate RTL specifications for a hardware design, typically in *Hardware Description Languages* (HDL) such as VHDL or Verilog. This enables users to develop large scale designs at a high level of abstraction.
2. *Design space exploration*: Since HLS tools can reduce the design development time, users can leverage these tools to quickly evaluate many design options and, thereby, explore a much larger design space.
3. *Accessibility to non-hardware-experts*: Many HLS tools use high-level languages (i.e., C, C++, OpenCL, CUDA and Java) which are widely used for software development. Therefore, these tools are more comfortable to use for people with software development

backgrounds. Although these tools often require hardware design expertise to develop good quality designs, the higher level of abstraction offered by these tools make them much easier to learn for new users.

4. *Debugging and verification efforts*: Many HLS tools also help users to easily generate testbenches[1] to verify the correctness of the generated hardware designs. Since the HLS tools often use popular software development languages, users can also leverage widely used software debugging facilities to easily verify the high-level specifications. More recently, there are also efforts at integrating debugging infrastructure into HLS tools to make it easier to debug the generated designs without delving down to the circuit-level details [Calagar et al., 2014, Goeders and Wilton, 2015].
5. *Portability*: Starting from the same input specifications, HLS tools can automatically generate the RTL specifications that are tuned to different implementation targets, e.g., FPGA devices from different product families, vendors or with different resource constraints. Therefore, these tools offer a design portability that is seldom possible when developing the design directly in HDL.

History of HLS tools

Due to these benefits, there has been many efforts, both in academia and industry, to develop such HLS tools. Martin and Smith [Martin and Smith, 2009] provide a good overview of the history of HLS tools. According to the authors, HLS tools have evolved over three generation. During the first generation, tool development occurred mostly in academia and much of the core algorithmic research occurred during this period. In the second generation, the industry took active interest in HLS tools and many commercial tools were developed, but they did not have much economic success, largely because of the bad quality of results and wrong choice of specification languages. In the following third generation, which started in early 2000s, there was a change in focus and many new HLS tools were developed that take specifications in languages such as C, C++ and SystemC to produce good quality hardware designs; This has improved adoption of HLS tools among system-level designers who already have a significant amount of hardware design expertise. However, to further improve the adoption, we need to develop tools that will permit software developers and application domain experts

[1]Testbenches contain a set of input stimuli as well as expected responses and are used to verify the correctness of a hardware circuit

to create good quality hardware designs despite their lack of hardware design knowledge.

Survey and classification of HLS tools

As a result of the rich history, today there exists a large number of HLS tools, each focusing on different aspects, like target application domain, implementation architectures and input languages. Nane et al. [Nane et al., 2016] and Meeus et al. [Meeus et al., 2012] have provided good overviews of the rich selection of hardware synthesis tools that exist today, and Bacon et al. [Bacon et al., 2013] classified some of these tools based on the programming languages they use. We are interested in tools that use domain-specific languages to make programming reconfigurable hardware more accessible to application domain experts and software programmers. Therefore, in this section, we try to classify these efforts into two main categories: general-purpose hardware synthesis tools and domain-specific hardware synthesis tools.

2.1.1 General-Purpose Hardware Synthesis Tools

The most popular approach today is to offer HLS tools that take input specifications in a C-like language, such as C, C++, OpenCL and CUDA. Among them, the traditional approach was to develop tools to use sequential variants of C, such as C and C++. Later, with the increasing prominence of parallel programming methodology, there have been efforts to use parallel programming models, such as PThreads [Nichols et al., 1996] and OpenMP [Dagum and Menon, 1998], and later, explicitly parallel variants to C, such as OpenCL and CUDA. In addition to these, there have been efforts to use other general-purpose programming languages, such as Python [Van Rossum and Drake, 2003], Java [Arnold et al., 1996], Haskell [Thompson, 1999] and custom developed languages for hardware synthesis.

Synthesis from Sequential C-Like Languages

Using sequential C-like languages for hardware synthesis

Languages such as C and C++ offer direct and low-level control while developing software for processors. Consequently, these languages are often used to develop performance critical parts of programs. In areas such as embedded systems, these languages are almost defacto standards for application development. Since ASICs and FPGAs can offer improved performance and energy efficiency compared to processors, there is a lot of interest in developing tools to help port the critical parts of such applications into hardware designs that can be implemented as an ASIC or on an FPGA. Therefore, there are many HLS tools that generate hardware designs from specifications in C and C++. Many of these tools also support SystemC [Grötker et al., 2010] which was developed as set of C++ classes to make it easier to model hardware circuits in C.

Tools that accept C, C++ or SystemC for input specifications include Vivado High-Level Synthesis [Xilinx, 2013a], Handel-C [Mentor Graphics, 2015], Catapult [Calypto Design Systems, 2014], Synphony [Synopsys Inc., 2014], CyberWorkBench [Wakabayashi, 2005], ROCCC [Villarreal et al., 2010], LegUp [Canis et al., 2011] and Trident[Tripp et al., 2007] to name just a few. Among them, Vivado High-Level Synthesis, Handel-C, Catapult, Synphony and CyberWorkBench are commercial tools. Trident focuses on mapping computations rich in floating-point operations on FPGA. ROCCC and LegUp are open-source tools that are being developed in the academia; among them LegUp can generate standalone hardware implementations as well as processor-accelerator architectures targeting different FPGAs.

Drawbacks

FPGA implementations often leverage the spatial parallelism of the device to improve the performance of application. However, extracting parallelism from sequential C-programs is hard [Cong et al., 2011]. Therefore, many of these tools require the user to refactor the code and provide additional information (e.g., compiler directives and configuration parameters) to generate parallel hardware. This makes these tools harder to use for the developer.

Synthesis using Parallel Programming Models

Using parallel programming models for hardware synthesis

FPGAs excel in parallel execution by performing operations in a spatially parallel manner. Therefore, researchers have tried to use parallel programming models, such as OpenMP [Dagum and Menon, 1998] and PThreads [Nichols et al., 1996], to develop applications for FPGAs. These models extend the sequential C-like languages to enable users to programmatically express the parallelism in the computation. Additionally, this also makes the specifications portable between the different tools since the users no longer have to use ad hoc optimization directives or configuration parameters that differ from one tool to another. Efforts to synthesize hardware using parallel programming models include Leow et al. [Leow et al., 2006] who automate the generation of hardware systems by producing Handel-C and VHDL code from OpenMP programs; Cilardo et al. [Cilardo et al., 2013] who generate C code from OpenMP program and use CoDeveloper [Antola et al., 2007] (another commercial C-to-HDL tool) to generate hardware; and Choi et al. [Choi et al., 2013] who achieve the same from programs using both OpenMP and PThreads by extending LegUp to generate hardware systems.

Using parallel programming models to target multi-processor systems on FPGAs

Other efforts have implemented multi-processor systems on FPGAs and utilized the parallel programming models to program these systems. Among them, efforts such as Hthreads [Andrews et al., 2008], Fuse [Ismail and Shannon, 2011] and ReconOS [Agne et al., 2014] use PThreads and provide generalized operating system services to systems that support hardware and software threads. SPREAD [Wang et al., 2013] utilizes PThreads and provides an integrated solution for streaming applications.

Drawbacks

These efforts, especially those that target multi-processor systems, have been successful in enabling software developers to develop and run applications on FPGAs. But, the common problem with these efforts is that they place the tedious and error-prone task of identifying synchronization requirements and correctly parallelizing the application on the programmer. In the case of HLS tools, the users may need to understand details of the generated hardware design to perform optimizations. The processor-based approaches do not fully exploit the application specific customizability offered by FPGAs. Since they use processors as computing units, they also suffer from the energy-inefficiencies of this general-purpose, instruction driven architecture [Hameed et al., 2010]; additionally, these inefficiencies are now worsened because the implementation target is an FPGA and not an ASIC [Kuon and Rose, 2007].

Synthesis from Parallel C-Like Languages

Using parallel C-like languages for hardware synthesis

The growing popularity of using GPUs for general-purpose computation saw the emergence of new parallel C-like languages such as CUDA [NVIDIA Corporation, 2015] and OpenCL [Stone et al., 2010] that were developed to program these devices. Compared to the sequential variants, these languages make the parallelism explicit and therefore much easier for compilers to generate implementations targeting parallel architectures, including FPGAs. Papakonstantinou et al. [Papakonstantinou et al., 2009] demonstrate that CUDA can be used for programing FPGAs. Commercial tools to generate hardware designs from OpenCL programs include Xilinx's SDAccel [Xilinx, 2015] and Altera SDK for OpenCL [Czajkowski et al., 2012]. SOpenCL [Owaida et al., 2011] is an academic tool that generates hardware designs from OpenCL program.

Drawbacks

Languages such as OpenCL and CUDA are good to program applications that have regular, fine-grained data parallelism. Therefore, programs written in these languages can be efficiently mapped to regular architectures such as GPUs and benefit from the vector processor units on modern CPUs. In addition to regular, fine-grained parallelism, FPGAs excel when applications have irregular parallelism and can benefit from architecture customization, such as application-specific memory or compute structures. More recently, extensions, such as OpenCL Pipes, have been proposed and they can potentially improve the suitability of these languages for developing FPGA applications [Altera Corporation, 2015]. However, such extensions also complicate the application development process since there are now many ways to write the same application, each with a different trade-off between cost and performance.

Common problem with C-like languages

HLS tools using the different variant of C-like languages can produce good results [Andrade et al., 2015, Rupnow et al., 2011, Cong et al., 2011, Chen and Singh, 2012]. However, a common problem in these approaches is that their input languages are too low-level; therefore, the user require to have a detailed knowledge of the optimization potential of the application and manually perform some optimizations, such as refactoring the program or adding compiler directives [Rupnow et al., 2011]. There are also sound arguments against using C-like languages for HLS [Edwards, 2006]. Therefore, researchers have investigated using other languages for hardware synthesis.

Synthesis from Other General-Purpose Programming Languages

Using other general-purpose languages for hardware synthesis

Researchers have synthesized hardware designs from other programming languages, besides the C-like languages noted above. These efforts include Kiwi [Greaves and Singh, 2008] which uses a parallel programming library to generate hardware circuits from a parallel program in C# [Hejlsberg et al., 2003]; MyHDL [Decaluwe, 2004] that models concurrency of hardware circuits with generator functions in Python; JHDL [Bellows and Hutchings, 1998] that uses Java to model hardware circuits; and Lava [Bjesse et al., 1998] which leverages functional programming language features, such as monads and type classes, to generate hardware circuits. However, among them, MyHDL and JHDL start from a circuit-level description of the hardware design and offer only a limited productivity advantage.

Synthesis from Custom Programming Languages

Using custom languages for hardware synthesis

Other researchers have tried to create new languages to generate hardware. Among them, BlueSpec [Nikhil, 2004] uses guarded atomic actions to express concurrent FSM and they excel in generating control dominated circuits. Chisel [Bachrach et al., 2012] was developed as a language embedded in Scala [Odersky et al., 2004] and it provides a more sophisticated language for hardware development. Both these approaches improve the designer productivity compared to using Verilog or VHDL; however, they require the designer to step down from the abstraction level of algorithms to think more in terms of the hardware design and its functioning. Lime [Auerbach et al., 2010] is a Java-based language that uses task-based data-flow programming model to target heterogeneous system that include CPUs, GPUs and FPGAs. When targeting FPGAs, the tasks in Lime becomes separate hardware circuits that are interconnected according to a task-graph.

Drawbacks

These tools improve the productivity of the designer and, in many cases, produce reasonably good results. However, the common problem with all general-purpose tools is that they lack domain-knowledge and therefore cannot perform domain-specific optimizations making it difficult to tune the generated hardware design for specific application areas. These tools, therefore, depend on the users to appropriately refine the input specifications, which requires hardware design knowledge, and guide the tool to produce better quality designs. This makes the tool less useful to application domain experts or users with a software development background.

2.1.2 Domain-Specific High-Level Synthesis Tools

Benefits of domain-specific languages for hardware synthesis

In contrast to general-purpose tools, domain-specific tools focus on specific application domains where they make hardware synthesis easier and more accessible for their users. Compared to general-purpose tools, these tools offer the following advantages:

1. They often use a custom DSL for input specifications. These DSLs have a syntax that makes it easier to express applications in the domain and, thereby, improves the productivity of the user.
2. They can leverage advanced domain-knowledge to perform optimizations and/or have custom implementations of common domain operations. This enables them to produce better quality results compared to general-purpose tools.

Tools using domain-specific languages for hardware synthesis

These include tools such as PARO [Hannig et al., 2008] and MMAlpha [Derrien et al., 2008] which focus on loop transformations to implement highly parallel systems; Spiral [Milder et al., 2012] which synthesizes hardware for linear transforms in signal processing applications; Optimus [Hormati et al., 2008] which focuses on streaming applications; and HIPAcc [Reiche et al., 2014] that targets image processing applications. All these tools focus on specific application areas and use custom DSLs to elicit input specifications. Some commercial tools in this category include HDL Coder [The MathWorks, 2015] which takes Matlab program or Simulink models and generates hardware designs for them; LabView [Bishop, 2014] that provides a graphical user interface to develop designs for select application areas, such as signal processing, instrument control, embedded system monitoring and control; and SDNet [Brebner and Jiang, 2014, Xilinx, 2014] from Xilinx which focuses on networking applications.

Some efforts have also tried to use existing general-purpose programming languages to target specific application domain. These include efforts such as Gaut [Coussy et al., 2008] that takes specifications in C but performs optimizations to target signal processing, and Streams-C [Gokhale et al., 2000] which also uses C and focuses on stream processing.

Reducing the effort to develop domain-specific tools

While domain-specific tools have huge potential in making FPGA more accessible to users without hardware design knowledge, the high effort required to build these tools makes this approach impractical in many areas. In the software domain, research efforts demonstrate that language embedding and type-directed staging [Rompf and Odersky, 2012] based approaches can be used to reduce domain-specific tool development effort. These efforts include the Delite project [Lee et al., 2011, Sujeeth et al., 2014] which targets portability and high performance for applications running on heterogeneous platforms; a software-only subset of Spiral [Ofenbeck et al., 2013]; and Jet [Ackermann et al., 2012] which targets BigData computation. Inspired by these results, we investigate how similar ideas can be applied to reduce the effort needed to develop hardware synthesis tools [George et al., 2013]. Additionally, we show that developing these tools using reusable optimization modules and integrating with existing general-purpose hardware generation tools can make it significantly easier to build such domain-specific tools.

Need for developing hardware synthesis tools that do not require hardware design expertise

As noted above, integrating with existing general-purpose HLS tools can significantly lower the effort needed to develop new domain-specific hardware synthesis tools. But, the general-purpose HLS tools require the user to tune the input specification (e.g., by refactoring the code or supplying additional optimization directives) based on hardware design expertise to produce good quality results [Rupnow et al., 2011]. People developing domain-specific tools, however, might be domain-experts or software developers who lack hardware design skills. Therefore, to make it easier for them to create new domain-specific tools, we need to develop a hardware generation tool that does not require hardware design expertise to use and can yet produce good quality designs.

2.2 Generating Hardware Designs from Computational Patterns

2.2.1 Overview of Computational Patterns

Understanding computational patterns

On analyzing applications from different domains, researchers have observed that they contain a few unique patterns of computation and communication. *Computational pattern* [Asanovic et al., 2006, McCool et al., 2012] are algorithmic methods that capture patterns of computation and communication. These patterns have well defined properties, such as parallelism in the operations, interdependence between elemental operations, data access patterns and data-sharing characteristics.

More importantly, these properties can be leveraged to efficiently map these patterns on different computational architectures, such as CMPs, GPUs, clusters and even FPGAs [Newburn et al., 2011, Sujeeth et al., 2014, Chambers et al., 2010, George et al., 2014].

2.2.2 Using Computational Patterns for Processor-Based Architectures

Programming processor-based architectures from computational patterns

As noted above, the well defined properties of the computational patterns make it easy to optimize them and map them efficiently on a variety of different architectures. For instance, Intel's Array Building Blocks [Newburn et al., 2011] uses data-parallel patterns to generate parallel code for CMPs; Copperhead [Catanzaro et al., 2011] uses patterns to generate CUDA code from a subset of Python to target GPUs; DryadLINQ [Yu et al., 2008] takes programs in LINQ [Meijer et al., 2006] and executes them over clusters using Dryad [Isard et al., 2007]; Delite [Lee et al., 2011, Sujeeth et al., 2014] decomposes high-level DSL programs to patterns and executes them on heterogeneous machines containing CMPs and GPUs as well as on clusters; and FlumeJava [Chambers et al., 2010] from Google provides a Java library to develop applications which it decomposes into pipelines of MapReduce operations that can be executed on their MapReduce framework [Dean and Ghemawat, 2008].

These efforts demonstrate that the well understood properties offered by computational patterns can be used to generate implementations for different targets. More importantly for our purpose, as seen in the case of Delite, high-level compilers can decompose a DSL program into these computational patterns. Therefore, if we develop a toolchain to generate hardware designs from computational patterns, it can be used along with a high-level compiler to compile DSL programs into hardware designs.

2.2.3 Using Computational Patterns for Hardware Synthesis

Hardware generation from computational patterns

Computational patterns have been used before in the context of hardware designs. Patterns were used to analyze the amenability of accelerating algorithms on FPGAs before implementing them manually [Nagarajan et al., 2011]. Some parallel programming models (e.g., OpenMP) and languages (e.g., Lime) have constructs to express some patterns, such as `map` and `reduce`, to reveal the parallelism in the computation. Developing a tool to generate hardware from computational patterns has the following advantages:

1. The tool can leverage the well understood properties of the computational patterns to produce high-quality hardware implementations for them.
2. Since the computational patterns are algorithmic methods, they are easily understood by domain-experts or software developers who can use the tool to generate hardware designs. Additionally, they can also develop high-level compilers that will decompose DSL-programs into computational patterns and generate hardware designs.

In our work, we decompose high-level DSL applications into comptuational patterns and develop a toolchain to generate hardware designs from these patterns [George et al., 2014]. To generate high performance designs from this toolchain, we leverage the properties of the patterns and perform additional compiler analysis to infer optimizations and generate a well structured HLS code with all the necessary optimization directives. Our implementation uses Vivado High Level Synthesis [Xilinx, 2013a] to generate hardware modules and Vivado Design Suite [Xilinx, 2013b] to generate bitstreams for the FPGA. Since we automatically generate the input to the HLS tool from the high-level patterns, we can also reduce the syntactic variance in the generated code and, consequently, its impact on the performance of the generated design [Chaiyakul et al., 1992]. Furthermore, we integrated this toolchain with Delite [Lee et al., 2011, Sujeeth et al., 2014], an extensible compiler infrastructure that can be used to easily develop DSLs for new application domains, to compile high-level DSL programs in Delite to complete hardware systems. To generate such complete systems, our approach leverages the domain-awareness provided by the DSL to select a suitable system-architecture template for the implementation.

Multi-module parallelization from computational patterns

Later, we extended this methodology to automatically generate designs that parallelize computations across multiple independent hardware modules [George et al., 2015]. This enables us to utilize the parallelism revealed by the patterns to overcome performance bottlenecks of designs that only use a single HLS-generated hardware module for each parallel operation. These include the underutilization of the available system bandwidth either due to the nature of computation in the module [Zhang et al., 2015] or due to the data access patterns when coupled with the long external memory access latency. Our approach exploits the properties of the pattern to automatically parallelize computation and uses a dynamic load-balancing strategy to leverage multiple modules to improve the performance of the application. An orthogonal approach proposed by Winterstein et al. [Winterstein et al., 2015] is to apply program analysis on the HLS code to identify non-overlapping memory regions and parallelize applications.

The multi-module parallelization can benefit applications with irregular data access patterns. Therefore, there have been efforts to develop specialized HLS tools that use techinques such as deep pipelining [Halstead and Najjar, 2013] and context switching [Tan et al., 2014] to improve the performance of such applications. These techniques are also orthogonal to our approach and can be used in conjunction with our work.

Other hardware generation efforts from computational patterns

Subsequently, Prabakar et al. [Prabhakar et al., 2015] proposed using tiling and metapipelining optimizations to improve the quality of the designs generated from computational patterns; they used Delite as the front-end and Maxeler's MaxCompiler [Maxeler Technologies, 2011] to generate hardware designs. Ma et al. [Ma et al., 2015] have shown that by decomposing high-level applications into a library of patterns, they can utilize the dynamic reconfiguration ability of FPGAs to enable runtime interpretation of application programs. These efforts reiterate the benefit of generating hardware designs from computational patterns.

3 Making Domain-Specific Hardware Synthesis Tools Cost-Efficient

Advantages of domain-specific tools

High-level synthesis tools can enhance productivity of FPGA application developers. Among them, tools such as Spiral [Milder et al., 2012], HDL Coder [The MathWorks, 2015], Optimus [Hormati et al., 2008], and MMAlpha [Derrien et al., 2008] target specific application domains in which they make designing hardware even more convenient and accessible to their users. To achieve this, they utilize a high-level *Domain-Specific Language* (DSL) for the input specifications, such as SPL for Spiral, Matlab/Simulink design for HDL Coder, StreamIt for Optimus and Alpha for MMAlpha, which is more natural to express the applications they target. Additionally, many of these tools also leverage domain knowledge to optimize the hardware designs and deliver better design quality compared to general-purpose tools. A simple tool for synthesizing matrix expressions into hardware circuits can help us to understand these advantages: Firstly, using a DSL that uses concepts like matrices and operations on matrices makes expressing the input design easy, especially for domain-experts. Secondly, since the tool knows that the input is a matrix expression, it can automatically choose ideal data storage formats and operator implementations to ensure higher quality results compared to general-purpose tools. Lastly, the tool can exploit its domain awareness, e.g., knowledge of the rules of matrix algebra, to optimize the design and produce even better results.

3.1 Motivation

Build domain-specific tools with low effort

Developing a new domain-specific HLS tool is a significant investment since it may involve designing an entirely new DSL, compiler and application development environment, such as IDE and debugging tools; developing just the compiler would comprise writing a parser, multiple analysis and optimization phases and output code generators. This large development effort and, consequently, high cost limits the viability of developing new domain-specific HLS tools for new application domains. However, techniques such as language embedding [Mernik et al., 2005] and type-directed staging [Carette et al., 2009] can considerably reduce this development effort. In this chapter, we explore the feasibility of using these techniques to develop domain-specific synthesis tools; we embed the new DSL in Scala [Odersky et al., 2004] and utilize the *Lightweight Modular Staging* (LMS) [Rompf and Odersky, 2012] infrastructure for developing the compiler.

Case study: hardware synthesize from matrix expressions

As an illustration of this approach, we create an HLS flow to synthesize matrix expressions into hardware designs. This flow is composed of two optimization modules, one performing optimizations at the matrix-level and the other at the level of matrix elements (scalar-level). Using this flow, we demonstrate the flexibility of the approach by showing how we can easily reuse the optimization modules and by integrating it with external tools like LegUp [Canis et al., 2011], a C-to-RTL compiler, and FloPoCo [De Dinechin and Pasca, 2011], an arithmetic core generator. While we present the concepts using a specific HLS flow, they are indeed quite general and can be used to create domain-specific tool-chains and/or to augment existing tool-chains with domain-specific optimizations with a very reasonable development effort.

Chapter outline

The remainder of this chapter is organized as follows. Section 3.2 describes our modular design approach and Section 3.3 provides a brief introduction to LMS and our hardware design generation work-flow. In Section 3.4, we detail the development of the optimization modules. We evaluate our approach in Section 3.5 using a case study of implementing multiple matrix expressions in hardware. The results show that we can leverage high-level optimizations to considerably reduce the implementation area without significant performance loss. We then summarize our findings in Section 3.6.

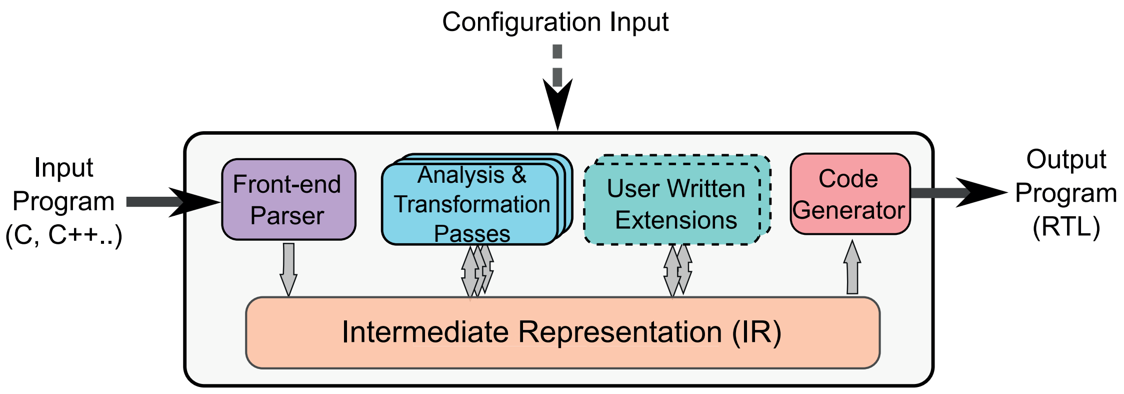


Figure 3.1: HLS Tool Design Using Traditional Compiler Frameworks. Typical HLS tools use traditional compiler frameworks like LLVM or GCC which have use one or more low-level IR-formats.

3.2 Compiler Frameworks for Domain-Specific Hardware Synthesis

In this section, we discuss the advantages of LMS over popular open-source compiler frameworks, such as LLVM [Lattner and Adve, 2004] and GCC [Stallman et al., 2015], as a common platform for implementing multiple domain-specific synthesis tools.

Overview of traditional compiler frameworks

Open-source compiler frameworks like LLVM and GCC have been used to build HLS tools like Vivado HLS [Xilinx, 2013a], OpenCL to FPGA [Czajkowski et al., 2012], LegUp [Canis et al., 2011] and Trident [Tripp et al., 2007]. In these HLS tools, as shown in Figure 3.1, the input program is translated into an *Intermediate Representation* (IR) format by the front-end parser. Various analysis and transformation passes then optimize this IR, maintaining its original format, until the code generator finally uses it to produce the output RTL code. While HLS tool designers often extend the framework by adding additional front-end parsers, analysis and transformation passes, and code generators, they usually use the same low-level IR-format[1] used by the framework since changing it is not easy. By retaining the IR-format, they can reuse some of the standard optimizations that are already available in the framework, but it also introduces limitations.

[1]GCC does have multiple IR-formats, but they are all at a low-level; therefore, these multiple IR-formats offer no benefit when performing high-level domain-specific optimizations.

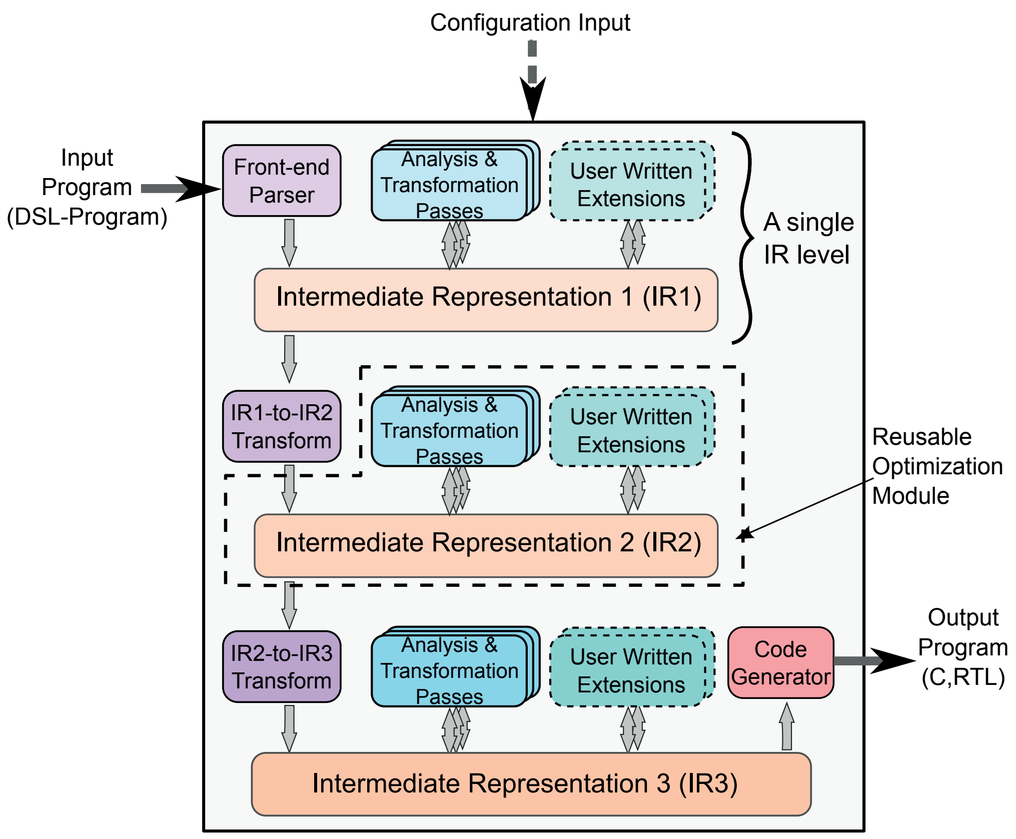

Configuration Input
Input Program (DSL-Program)
Front-end Parser
Analysis & Transformation Passes
User Written Extensions
A single IR level
Intermediate Representation 1 (IR1)
IR1-to-IR2 Transform
Analysis & Transformation Passes
User Written Extensions
Reusable Optimization Module
Intermediate Representation 2 (IR2)
IR2-to-IR3 Transform
Analysis & Transformation Passes
User Written Extensions
Code Generator
Output Program (C,RTL)
Intermediate Representation 3 (IR3)

Figure 3.2: HLS Tool Design Using Our Approach. In contrast to Figure 3.1, we propose using an architecture based on multiple IR-formats at different levels of abstraction, each used by a unique optimization module, to enable a more efficient hardware synthesis.

Traditional compiler frameworks are unfit for domain-specific tools

For instance, many domain-specific tools, like Spiral [Milder et al., 2012] and MMAlpha [Derrien et al., 2008], use multiple IR-formats to perform high-level, domain-specific optimizations at different levels of abstraction. One way to achieve this with a single IR is to perform the high-level, domain-specific optimizations directly at the front-end parser, before the low-level IR is generated. But, these optimizations are now tied to the specific IR used by the developer and, consequently, cannot be reused in a different HLS flow if the IR changes. An alternative is to perform the high-level optimizations using a sequence of optimizations on the low-level IR. But, this can be extremely difficult due to the low-level nature of the IR; for instance, performing simple optimizations using rules of matrix algebra is quite easy when the IR represents operations on matrices, but it becomes hard or just not feasible when the higher-level information is lost. Furthermore, with a single IR-format, there can also be complex interactions between the different optimization steps, making it harder to add new optimizations or reuse the existing ones selectively for an entirely new flow.

Proposed approach

In our approach, we use the extensible LMS compiler framework to define multiple IR-formats, as shown in Figure 3.2, that are often at different levels of abstraction. We create separate standalone *optimization modules* for each unique IR-format. The components of an optimization module include multiple analysis and transformation passes; an optional lightweight, front-end parser for the input DSL; and optional code-generators for the different output formats. Since all these components operate only on the unique IR-format used in the module, we can more easily avoid the possibility of optimizations in different modules affecting each other; this makes it easier to make changes or add/remove optimizations. Additionally, as shown in the figure, we can develop complex HLS flows by connecting multiple optimization modules to each other using a special variant of a transformation pass called *IR-to-IR transform* [Rompf et al., 2013]. Often, the IR-formats are common across different HLS flows, enabling a natural reuse of optimization modules.

Contrast with other efforts

In the software domain, the Delite project [Lee et al., 2011] employs a similar strategy to target code-generation for multi-core CPUs and GPUs starting from a high-level DSL. In this work, however, we demonstrate that a similar approach can dramatically lower the cost for developing domain-specific HLS flows to achieve efficient hardware generation. In particular, we show that this method of building domain-specific flows fits naturally into the current EDA ecosystem by enabling an easy integration of existing IP-cores and the extension of mature HLS flows by providing domain-specific optimizations.

Case study to demonstrate the benefits of the approach

Using the proposed approach, we can easily develop domain-specific hardware synthesis tools like Spiral or MMAlpha by creating different optimization modules for each level of abstraction at which we want to apply optimizations. To serve as an example and illustrate this point, we use a simple tool to synthesize matrix expressions into efficient hardware. We implement this flow using two optimization modules, one for performing optimizations at the matrix-level and the other for applying optimizations at the scalar-level. As done in many domain-specific tools, we also create a simple DSL to the make it easy for the end-user to express design specifications. Our matrix-level optimization module uses an IR that represents computation on matrices (i.e., a matrix-level IR) to reorder multiplications, apply the distributive property, and eliminate common subexpressions at the matrix-level. After that we use an IR-to-IR transform to decompose this matrix-level IR into a scalar-level one that represents operations on individual matrix elements. We then perform standard scalar-level optimizations, such as strength reduction and common subexpression elimination, as done in many HLS tools, before generating the output design. Although this is only a well understood example, it is general enough to show the potential of the proposed methodology.

3.3 Hardware Synthesis Using Scala and LMS

In this section, we first introduce the LMS framework and discuss how we use it to efficiently implement the optimization modules. We then describe our LMS-based workflow for hardware synthesis.

3.3.1 Overview of Scala and LMS

Creating a DSL and custom compiler

Scala is a novel programming language that brings together functional and object-oriented programming concepts. More importantly, one can leverage Scala's syntax, rich type system and module system to embed other languages in it, giving us a very cost-effective way to create a custom DSL for our HLS flow. LMS is a library and compiler framework written in Scala that lends the ability to optimize programs written in a DSL that is embedded in Scala. More specifically, LMS enables one to represent the DSL-program in an *Intermediate Representation* (IR) format of our choice, a process referred to as *staging*; additionally, by using multiple IR-formats that represent the DSL-program at different levels of abstraction, we can progressively optimize it to achieve high quality results. In our approach, we group the different optimizations based on the IR-format that is most suited for their application and leverage the modularity of the LMS framework to organize them into separate reusable optimization modules. The framework enables us to easily compose these optimization modules and to connect them as required to implement custom HLS flows similar to the one shown in Figure 3.2.

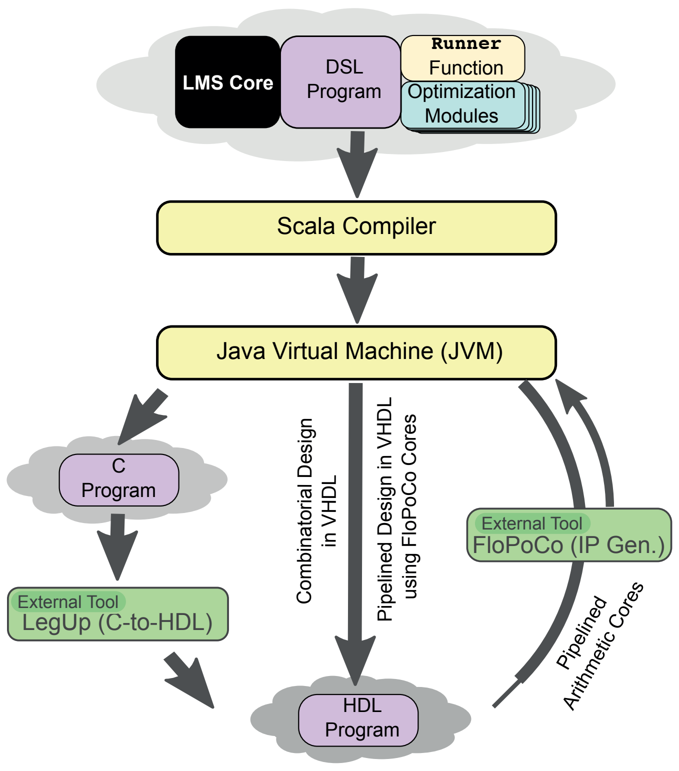


Figure 3.3: An LMS-based workflow for hardware synthesis. The input program to the flow consists of the LMS library, the DSL-program, the custom optimization modules and the `Runner` function. This input is compiled using the *Scala* compiler to generate a Java byte-code which is then executed by the *Java Virtual Machine* to produce the optimized, low-level program in the user specified output format (i.e., C and VHDL in our example). Other tools can be integrated into this flow by interacting with them during this byte-code execution. We demonstrate this by integrating both LegUp and FloPoCo into our workflow.

Building HLS flow using Scala and LMS

To build a new HLS flow, the tool designer develops custom optimization modules and defines a `Runner`; this `Runner` function controls when the optimizations in these modules are applied to generate the optimized hardware design. While creating the optimization modules, the tool designer can specify the following:

1. The DSL used to write the input specifications.
2. The IR-format for representing the DSL-program.
3. The transformation rules and how they are applied to optimize the IR for the DSL-program.
4. The format for the output produced by the tool.

To keep the development effort low, the tool designer can also reuse optimization modules from other flows and interface to external tools when it is both beneficial and possible.

3.3.2 Hardware Synthesis Workflow Using Scala and LMS

Overview of the workflow

Figure 3.3 shows the workflow we use for synthesizing hardware using LMS. The input to this flow is composed of the LMS library, the DSL program, a set of optimization modules that are needed for the specific HLS flow and the `Runner` function. This input is compiled using the Scala compiler into a byte-code format which is then executed on a standard *Java Virtual Machine* (JVM) to produce the output design.

Operation of the workflow

When the JVM executes this byte-code, the execution control passes to the `Runner` function which then orchestrates the process of translating the high-level DSL-program into optimized hardware. In the flow we present, this function first uses the facilities in LMS to *stage* the DSL-program (i.e., converts it into the IR); coordinates all the optimization steps, first at the matrix-level and then at the scalar-level; and finally generates the optimized hardware design as output. The output design can be generated in different formats as specified by the tool designer. To demonstrate this, we generate our outputs as a C program, as a combinatorial design in VHDL and as a pipelined design in VHDL that uses arithmetic cores from FloPoCo. In the last case, the tool directs FloPoCo so that the arithmetic cores are automatically pipelined depending on the desired operating frequency.

3.4 Hardware Designs from Matrix Expressions

In this section, we highlight the features of LMS and how we use them to design our optimization modules. To serve as an example, we use a matrix-level optimization module that tries to reduce the resources needed to implement a matrix expression in hardware. But, the features presented here can be used to develop optimization modules for other purposes, such as optimizing state-machines, DSP algorithms or streaming computation.

Example design

Figure 3.4 shows an example of the optimization performed by the matrix-level optimization module. The graph on the left computes

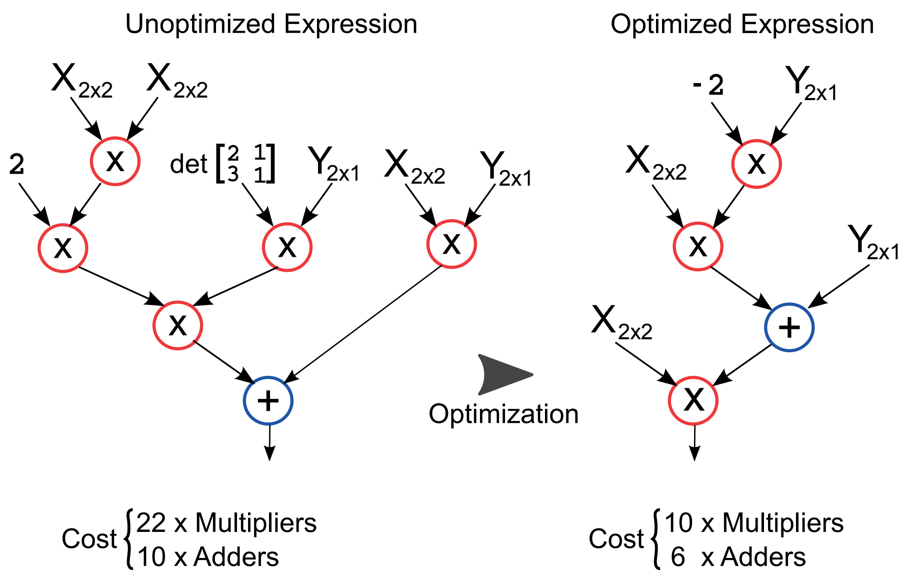


Figure 3.4: The graph on the left shows the matrix expression as specified by the user. To implement this directly in hardware, it would need 22 multipliers and 10 adders. The graph on the right is the same expression after folding away constants and applying some matrix-level optimizations. Implementing this design needs only 10 multipliers and 6 adders.

$2 \cdot det(\left[\begin{smallmatrix}2&1\\3&1\end{smallmatrix}\right]) \cdot X^2_{2\times2} \cdot Y_{2\times1} + X_{2\times2} \cdot Y_{2\times1}$, where $X_{2\times2}$ and $Y_{2\times1}$ are variable matrices, and it needs 22 multipliers and 10 adders, without considering the cost to compute $det(\left[\begin{smallmatrix}2&1\\3&1\end{smallmatrix}\right])$. Our module can transform this expression into the optimized form, shown on the right of the figure, implementing the same expression with only 10 multipliers and 6 adders.

3.4.1 Designing the Optimization Modules

As explained in Section 3.3.1, there are four aspects to developing an optimization module. We will now see these aspects in the context of the matrix-level optimization module.

Specifying the DSL

Creating the DSL operators and datatypes

To write matrix expressions efficiently, we can easily define a custom DSL, called *MatrixDSL*, by defining its data-types and operators, as shown in Figure 3.5; note that the syntax used here is the standard Scala syntax. In the figure, `Mat` and `T` are datatypes we defined for DSL to represent matrices and matrix elements, respectively. `newMat` and

```
//Datatype for matrices
type Mat
//Datatype for the matrix elements
type T = Int
//To create a Mat from Rep-type elements
def newMat (rows: Int, cols: Int, data: Rep[T]*) : Rep[Mat]
//To create a Mat from non-Rep elements
def newcMat(rows: Int, cols: Int, data: T*) : Mat
//Scalar-Matrix multiplication
def mult(x: Rep[T]  , y: Rep[Mat]) : Rep[Mat]
//Matrix-Matrix multiplication
def mult(x: Rep[Mat], y: Rep[Mat]) : Rep[Mat]
//Matrix-Matrix addition
def add(x: Rep[Mat] , y: Rep[Mat]) : Rep[Mat]
```

Figure 3.5: Defining the datatypes and operators for the custom DSL (MatrixDSL) that enables users to write computation on matrices efficiently.

newcMat are operators to compose matrices from scalar values of type T. Here, mult is an overloaded operator for multiplying matrices or a scalar with a matrix, and add specifies additions between matrices. The other data-types and operators of the DSL are defined in a similar manner.

Type-qualifier for *staging*

The Rep[] in the code is a special type-qualifier used by LMS that is used to mark variables that will be part of the constructed IR. Operators such as newMat, mult and add that produce or use Rep-type variables also become part of this IR. In LMS, this process of constructing an IR from a DSL-program is called *staging*.

The example design in MatrixDSL

Now, we can express the matrix expression illustrated in Figure 3.4 as the MatrixDSL-program shown in Figure 3.6. Here, lines 2–4 construct the matrices $X_{2\times2}$, $Y_{2\times1}$ and the constant matrix, $C_{2\times2}$. Lines 5–8 express the computation on these matrices. As mentioned earlier, only those operations that either use or produce Rep-type variables are made part of the IR. Others, like det(C), will be evaluated and replaced by its constant value in the IR. In this manner, LMS performs a partial-evaluation of the DSL-programs while staging it.

```
def func(a: Rep[T], b:Rep[T], c:Rep[T], d:Rep[T],
                              e:Rep[T], f:Rep[T])={
  val X = newMat (2,2, a,b,c,d)
  val Y = newMat (2,1, e,f)
  val C = newcMat(2,2, 2,1,3,1)
  val t1 = mult(X, X)
  val t2 = mult(2, t1)
  val t3 = mult(det(C), Y)
  add(mult(t2, t3), mult(X, Y)) // Return value
}
```

Figure 3.6: `MatrixDSL`-program for the matrix expression on Figure 3.4 (left).

```
1 case class MulMM (x:Rep[Mat], y:Rep[Mat])   extends Def[Mat]
2 case class MulSM (x:Rep[T]  , y:Rep[Mat])   extends Def[Mat]
3 case class AddMM (x:Rep[Mat], y:Exp[Mat])   extends Def[Mat]
4 override mult(x:Rep[Mat], y:Rep[Mat]) = MulMM(x,y)
5 override mult(x:Rep[T]   , y:Rep[Mat]) = MulSM(x,y)
6 override add(x:Rep[Mat] , y:Rep[Mat]) = AddMM(x,y)
```

Figure 3.7: IR-nodes for some operators defined by `MatrixDSL`. These nodes are used in the IR-graph constructed while staging a `MatrixDSL`-program.

Specifying the IR

Specifying the IR for DSL-programs

LMS provides the facility to represent DSL-programs in an IR-format that makes it easier to perform optimizations on it. The IR-format used by LMS is a directed *sea-of-nodes* graph that captures the dependencies between operations; this makes it easy to analyze and specify optimization rules on it. To represent a `MatrixDSL`-program as an IR-graph, we specify the IR-nodes for the operators defined in `MatrixDSL`, as shown for the `mult` and `add` operators in Figure 3.7 (lines `1`–`3`). We also redefine the original `mult` and `add` operators (lines `4`–`6`) so that each use of these operators will instantiate the respective IR-nodes in the IR-graph. In addition to the IR-nodes for operators used in the DSL-program, we need to also define IR-nodes that may be created during our optimization steps. Figure 3.8 shows the graph format IR resulting from *staging* the code in Figure 3.6.

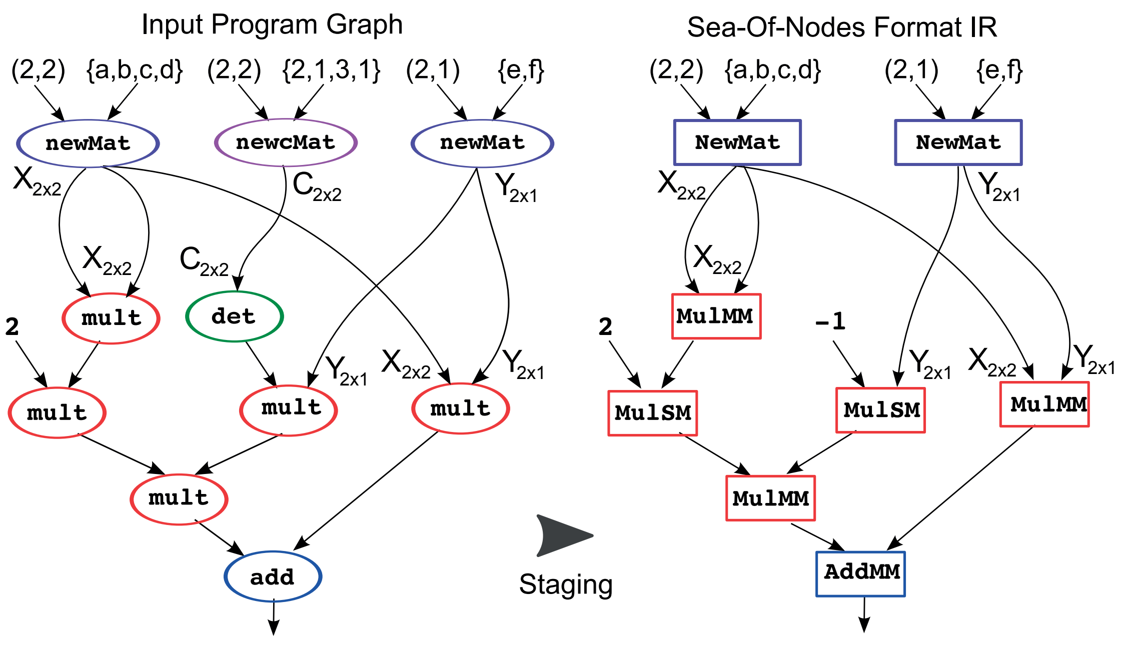


Figure 3.8: *Staging* the DSL-program. This figure illustrates the *staging* process. On the left we have the function `func()` (defined in Figure 3.6) represented in a graph-format and on the right is the *sea-of-nodes* IR-format constructed from *staging* this function.

Performing Analysis and Transformation

Once the DSL-program is expressed as an IR-graph, we can perform analyses and optimizations on it. LMS provides two mechanisms to perform optimizations: *staging-time macros* and *IR-transformers*.

Staging-time macros: Automatically triggered rewrites

Staging-time macros are rewriting rules that are applied automatically as the IR-graph for the DSL-program is being constructed; therefore, these are not intended for situations that need rewrite rules to be applied in a specific order. Here, the tool-flow designer specifies how certain IR-graph patterns must be rewritten. When any of these patterns appear in the IR-graph, the rewrite rule corresponding to it is automatically applied (i.e., triggered). LMS uses Scala's powerful pattern matching construct to make them easy to specify. Figure 3.9 shows some of the staging-time macros that are applied to the `mult` operators in the IR-graph on Figure 3.7. Figure 3.10 illustrates the steps through which these staging-time macros transform the initial IR-graph. Here, first the rules on lines `12`–`13` get applied to move the scalar multiplications $-1 \cdot Y_{2\times1}$ and $2 \cdot X^2_{2\times2}$ to occur after the matrix multiplication that

```
 1 // Mult operator
 2 def mult(x:Rep[T], y:Rep[T]) = (x,y) match {
 3   // Constant propagation: m * n --> (m*n)
 4   case(Const(m), Const(n)) => Const(m*n)
 5   ... }
 6 def mult(x:Rep[T], y:Rep[Mat]) = (x,y) match {
 7   // Constant propagation: x * (s*N) --> (x*s)*N
 8   case(x, Def(MulSM(s, n))) => mult(mult(x,s),n)
 9   ... }
10 def mult(x:Rep[Mat], y:Rep[Mat]) = (x,y) match {
11   // Forward scalar multiply:
12   // X * (s*N) --> s * (X*N)
13   case(x, Def(MulSM(s, n))) => mult(s,mult(x,n))
14   // (s*M) * Y --> s * (M*Y)
15   case(Def(MulSM(s, m)), y) => mult(s,mult(m,y))
16   ... }
```

Figure 3.9: An example of Scala's powerful pattern matching construct used to specify optimization rules in staging-time macros.

initially succeeded it. Now, the rule on line `8` rewrites the two successive scalar multiplications in $2 \cdot (-1 \cdot X^2_{2\times2} \cdot Y_{2\times1})$ into $(2 \cdot -1) \cdot X^2_{2\times2} \cdot Y_{2\times1}$. Finally, the rule on line `4` performs constant propagation, rewriting $2 \cdot -1$ as -2.

IR-Transformers: programmer triggered rewrites

Staging-time macros are very convenient, but they only perform localized rewrites without having a global view of the program and are automatically applied. To overcome these limitations, LMS provides the ability to perform analysis passes and *IR-transformers*. Unlike the staging-time macros, IR-transformers are explicitly called by the `Runner` function and can be preceded by analysis passes that collect additional information; this enables IR-transformers to have a global view of the program while performing rewrites. The IR-transformers are similar to the conventional compiler optimizations found in all compiler infrastructures. However, when using LMS, we can leverage Scala's pattern matching syntax which makes them easy to specify.

Using IR-transformers for optimization

Applying staging-time macros reduced the cost for implementing the initial matrix expression to 14 multipliers and 8 adders, as shown in Figure 3.10. We can now apply the distributive property of matrices

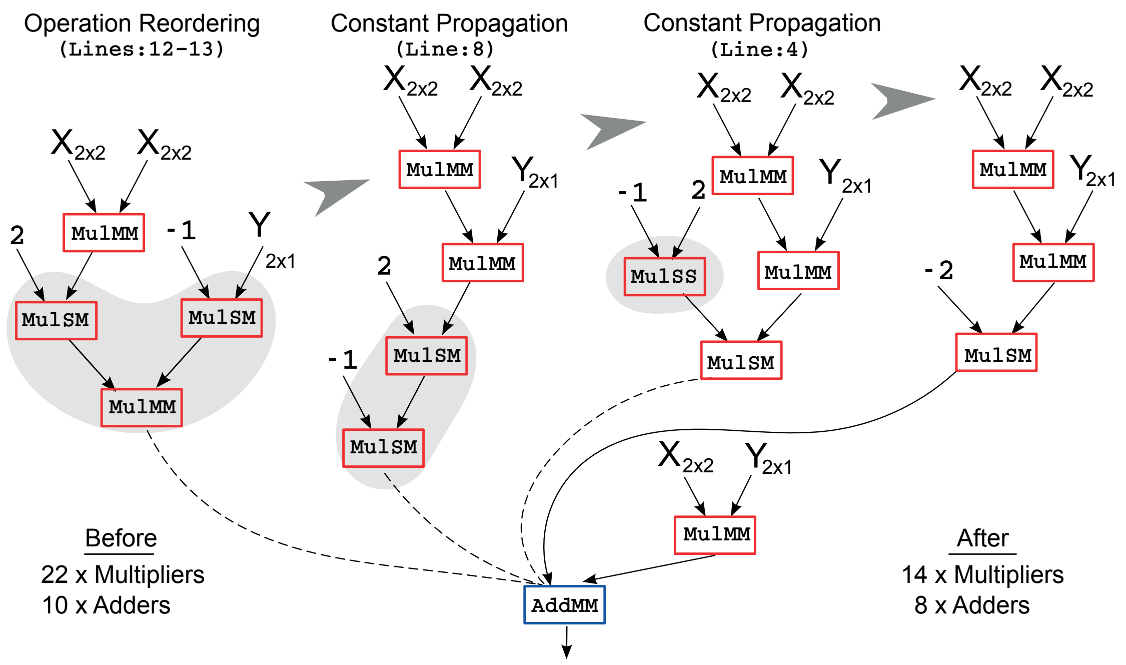


Figure 3.10: Staging-time macros get automatically applied as the IR-graph for the user design is being constructed (step shown in Figure 3.8). This figure shows how staging-time macros progressively optimize one of the operands to the AddMM-node. The figure also shows the line numbers of the rules in Figure 3.9 that are being applied in each step. (The NewMat nodes in this graph have been substituted with just $X_{2\times2}$ and $Y_{2\times1}$ to keep the figure simple to understand).

to further reduce the cost, but this presents us with three options as shown in Table 3.1. To select the best one, we first run analysis passes that estimate the cost after applying each of the possible optimizations and thus gather the necessary global information. With this knowledge, we now use an IR-transformer to move just the multiplication with $X_{2\times2}$ after the addition to obtain the result shown in Figure 3.4. If we had to use staging-time macros here, without the global view, we might easily end up choosing one of the other alternatives, achieving only sub-optimal results.

Using IR-transformers to connect optimization modules

In our HLS flow, we also use a variant of IR-transformers, which we call IR-to-IR transformer, to connect different optimization modules to each other. For instance, to connect the matrix-level optimization module to the one at the scalar-level, we use an IR-transformer that rewrites each matrix-level IR-node using multiple scalar-level IR-nodes. This effectively represents the matrix-level computation using operations at the scalar-level enabling us to use standard scalar-level optimizations to

Table 3.1: Comparison of alternatives after applying staging-time macros

Expression after applying staging-time macros	Multipliers	Adders
$-2 \cdot X_{2\times2}^2 \cdot Y_{2\times1} + X_{2\times2} \cdot Y_{2\times1}$	14	8

Implementation alternatives	Multipliers	Adders
$X_{2\times2}(-2 \cdot X_{2\times2} \cdot Y_{2\times1} + Y_{2\times1})$	10	6
$(-2 \cdot X_{2\times2}^2 + X_{2\times2}) Y_{2\times1}$	16	10
$X_{2\times2}(-2 \cdot X_{2\times2} + \left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)) Y_{2\times1}$	12	8

```
// Produces "sym = x * y;" in the C code
case MulSS(x, y) =>
 stream.println("%s=%s*%s;".format(quote(sym),quote(x),quote(y)))
```

Figure 3.11: This figure shows how the instances of `MulSS`-nodes are translated during code generation into C-code for scalar multiplications. The tool-flow designer must specify how each node will be translated during code generation.

further optimize the design. Using this technique, we can interconnect different optimization modules as needed while creating custom HLS flows.

Generating the Optimized Design as Output

To produce an output, LMS traverses the IR-nodes in the optimized IR-graph in the topological order, i.e., a node is visited only after all the nodes it depends on have been visited, and translates it into the output design. Using this facility, we only need to specify the translation rules for the IR-nodes that appear in the optimized IR-graph to generate the optimized output. During code generation, since LMS only visits IR-nodes which affect the final output, it also performs dead-code elimination.

Generating multiple output formats

Our matrix-level optimization module can generate an equivalent C program from its IR-format. Figure 3.11 shows how the `MulSS` IR-node that represents scalar multiplication is translated into the equivalent C-code where the result of `x*y` is assigned to `sym`. An optimization module can have multiple sets of translation rules to support different output formats, like different variants of C-code, VHDL-code, and have specific

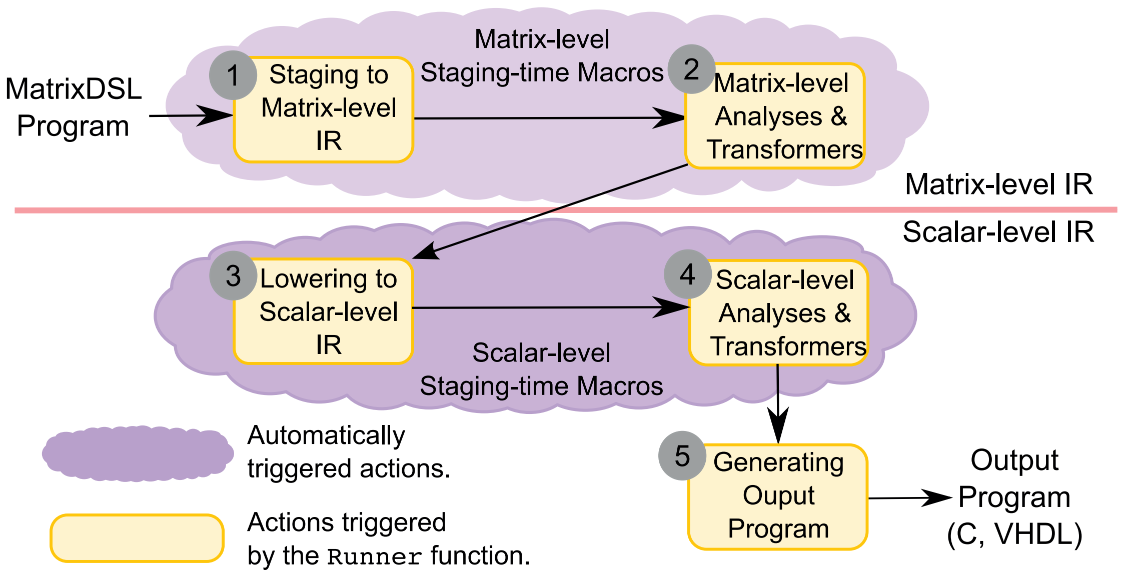

MatrixDSL Program
1 Staging to Matrix-level IR
Matrix-level Staging-time Macros
2 Matrix-level Analyses & Transformers
Matrix-level IR
Scalar-level IR
3 Lowering to Scalar-level IR
4 Scalar-level Analyses & Transformers
Scalar-level Staging-time Macros
5 Generating Ouput Program
Output Program (C, VHDL)
Automatically triggered actions.
Actions triggered by the Runner function.

Figure 3.12: This figure shows the steps involved in transforming a high-level `MatrixDSL`-program into optimized output program at the scalar-level.

interfaces to different external tools. For instance, our scalar-level optimization module produces both C and VHDL codes from its scalar-level IR-nodes. Additionally, during VHDL code-generation, it can generate a combinatorial design or interface with FloPoCo (an arithmetic core generator) to create the arithmetic components needed to generate a pipelined design. The module also uses the feedback from FloPoCo to know the pipeline stages in each generated component in order to produce a glue-logic containing registers to have a correctly pipelined datapath. We will not detail the development of the scalar-level optimization module separately since it uses the same LMS features that we have just described.

3.4.2 Managing the Compilation Process

Integrating the steps with the `Runner` program

Figure 3.12 shows the different phases in transforming a high-level `MatrixDSL`-program into an optimized hardware design. Referring back to Figure 3.3, as the JVM executes the byte-code, the execution control is handed over to the `Runner` function. This function then generates the IR-graph for the DSL-program at the matrix-level and calls the individual analysis and transformation steps in the matrix-level optimization

Table 3.2: Design IDs, benchmark expressions and corresponding matrix-level optimizations

ID	Matrix Expression	Matrix-Level Optimizations
M1	$4 \cdot A_{1\times5} \cdot B_{5\times5} \cdot C_{5\times3} \cdot D_{3\times1}$	Multiplication ordering
M2	$A_{4\times4} \cdot B_{4\times4} \cdot C_{4\times4} + B_{4\times4} \cdot C_{4\times4} \cdot D_{4\times4}$	Subexpression elimination
M3	$A_{3\times3} \cdot C_{3\times3} + B_{3\times3} \cdot C_{3\times3}$	Distributive property
M4	$2 \cdot det\begin{pmatrix}2 & 1\\ 3 & 1\end{pmatrix} \cdot X^2_{2\times2} \cdot Y_{2\times1} + X_{2\times2} \cdot Y_{2\times1}$	Distributive property & partial evaluation

Table 3.3: Results using LegUp to generate fixed-point datapaths

	Exp1: C with no opt.				**Exp2**: C with only matrix-level opt.				**Exp3**: C with two level opt.			
ID	# LEs	# Regs	Latency (μs)	Fmax (*MHz*)	# LEs	# Regs	Latency (μs)	Fmax (*MHz*)	# LEs	# Regs	Latency (μs)	Fmax (*MHz*)
M1	10,356	2,141	0.11	126	7,167	1,434	0.11	132	5,359	1,226	0.10	127
M2	39,541	5,688	0.15	117	35,322	4,825	0.15	117	13,158	3,072	0.17	106
M3	3,661	1,494	0.07	181	2,281	729	0.08	159	1,608	155	0.07	140
M4	1,748	463	0.07	131	1,547	309	0.06	148	1,547	309	0.06	154

module. The staging-time macros in this module are always automatically applied whenever one of its patterns appear in the IR-graph. Once the matrix-level optimizations are completed, the `Runner` function calls the IR-to-IR transform that converts the matrix-level IR into scalar-level IR. Then, the analysis and transform functions at the scalar-level are called to further optimize the DSL-program. Once again, the staging-time macros at the scalar-level get automatically applied whenever their control patterns appear in the graph. After the DSL-program has been optimized at this level, the `Runner` function calls the code-generator to translate this scalar-level IR into the final output design.

3.5 Evaluation Results

In this section, we will evaluate the quality of the results produced by our HLS flow and illustrate how performing optimizations at multiple abstraction levels can help improve results. However, our main objective is to demonstrate the reuse and integration flexibility of our approach and to show how it can be useful in a practical context.

3.5.1 Benchmark Designs

Designs used for evaluation

We use four designs listed in Table 3.3 for our evaluation; each of these focuses on a specific type of matrix-level optimization. Matrices in these designs were created by a random assignment of 10 independent scalar variables. We did this to limit the number of input arguments to the wrapper function. These designs are used to point out that existing general-purpose HLS tools do not perform higher order, domain-specific optimization and to illustrate how our design approach can avert this limitation.

Optimizations at the matrix-level

Table 3.2 lists the matrix expressions we used for the evaluation. In the expression **M1**, our matrix-level optimization module reorders multiplications to reduce its implementation cost. For expression **M2**, it performs common subexpression extraction to avoid the repeated computation of $B_{4\times4} \cdot C_{4\times4}$. In **M3**, the module uses the distribute property to extract the common operand, $C_{3\times3}$, from both the terms. **M4** is the running example used in Section 3.4; in this case, the optimization-module performs the transformation shown in Figure 3.4.

Optimizations at the scalar-level

For all designs, our scalar-level optimizations include common subexpressions elimination, such as removing the repeated instance of $a+b$ in $\binom{a}{b} + \binom{b}{a}$, and operator strength-reduction, like replacing multiplication by a power-of-two value with shift operation, that are found in many existing HLS tools.

3.5.2 Evaluating Benefit of Integrating to C-to-RTL Tool

Benefits of matrix-level optimizations

We first consider the case where the matrix elements are fixed-point integers and we integrate our HLS flow to LegUp 3.0 [Canis et al., 2011], an open-source C-to-RTL tool, to generate the hardware design. For each design, our flow generates three C-programs: **Exp1**, where our tool performs no optimizations; **Exp2**, with only matrix-level optimizations; and **Exp3**, with both matrix-level and scalar-level optimizations[2]. These C-programs are then compiled using LegUp with its best optimization setting and the generated RTL designs are synthesized using the Altera Quartus II toolset [Altera Corporation, 2013] to target a Stratix IV FPGA

[2] Since LegUp can only supports one output port in the hardware it synthesizes, we adapted our C-program generator by adding all the elements in the resultant matrix to produce a scalar value as output. To maintain complete fairness, we do this for every design we use in our study, even when we use other tools.

device; the designs were verified using circuit simulation. To make it possible to compare across designs, we restricted Quartus II from using any DSP blocks. Table 3.3 shows the result from synthesis in terms of number of *Logic Elements* (LEs), number of registers (Reg), circuit latency and maximum operation frequency (Fmax) for each case. Comparing the results from **Exp1** and **Exp2** on Table 3.3, it is easy to see that our matrix-level optimizations consistently reduce the resources needed for the implementation without impacting the frequency or latency significantly. These results reveal the potential of high-level optimization and how our methodology can complement existing tools to benefit from them.

Benefits of additional scalar-level optimizations

The scalar-level optimizations bring further improvement in terms of area (i.e., resources used by the design), as revealed by the results of **Exp3** on Table 3.3. The area saving mostly comes from the common subexpression elimination since LegUp already performs the strength reduction optimizations. This is easily evident in **M4** where we get no further area improvement through the scalar-level optimization as the program on Figure 3.6 offers no opportunity for scalar-level subexpression elimination. There is no consistent trend with Fmax and latency because there are two opposing effects: The smaller design size helps to improve Fmax and to lower the latency. But, since some of the optimizations increase the amount value sharing between different parts of the design, it places more pressure on the routing resources and has an adverse affect on both Fmax and latency. But, overall, the improvement in area is much more significant compared to the variations in timing. These experiments show that performing optimizations at different abstraction levels can progressively improve the design quality. Furthermore, implementing both of our optimization modules in LMS required less than 2,000 lines of code in total (including all the debugging code, comments and empty lines for formatting). This gives an indication of how little effort is needed to develop custom optimization modules for a new application.

Extending the tool to support floating-point matrices

We now consider the case of the matrices with floating-point elements to investigate the impact of such a change in our HLS flow. Since our matrix-level optimizations are agnostic to the element-type, we can reuse them by modifying only the type used for representing matrix elements (`T` in Figure 3.5). However, we do need to make some modifications to the scalar-level optimization module and its code-generator to

Table 3.4: Results using LegUp to generate single precision floating-point datapaths with two-level optimizations (**Exp4**)

ID	Matrix Expression	# LE	# Regs	Latency	Fmax	Throughput
			(μs)	(MHz)	(10^6· Results/s)	
M1	$4 \cdot A_{1\times5} \cdot B_{5\times5} \cdot C_{5\times3} \cdot D_{3\times1}$	10,178	11,806	0.89	235	1.1
M2	$A_{4\times4} \cdot B_{4\times4} \cdot C_{4\times4} + B_{4\times4} \cdot C_{4\times4} \cdot D_{4\times4}$	64,520	89,693	1.22	166	0.8
M3	$A_{3\times3} \cdot C_{3\times3} + B_{3\times3} \cdot C_{3\times3}$	11,281	14,097	0.57	231	1.8
M4	$2 \cdot det\left(\begin{smallmatrix}2 & 1\\ 3 & 1\end{smallmatrix}\right) \cdot X^2_{2\times2} \cdot Y_{2\times1} + X_{2\times2} \cdot Y_{2\times1}$	5,295	6,742	0.38	282	2.6

Table 3.5: Results using FloPoCo to generate pipelined datapaths with two-level optimization

ID	**Exp5**: FloPoCo-single precision equivalent					**Exp6**: FloPoCo-custom precision				
	# LE	# Regs	Latency (μs)	Fmax (MHz)	Throughput (10^6· Results/s)	# LE	# Regs	Latency (μs)	Fmax (MHz)	Throughput (10^6· Results/s)
M1	46,967	53,180	0.87	176	176	15,237	27,639	0.50	242	242
M2	174,193	181,619	1.36	116	116	59,787	88,504	0.65	188	188
M3	31,942	35,860	0.53	186	186	11,245	16,331	0.33	231	231
M4	9,657	12,452	0.32	212	212	2,919	3,830	0.17	312	312

support this feature. The total change involved adding/editing less than 50 lines of code, which illustrates the flexibility and the reuse possible in our approach. Table 3.4 shows the results obtained using LegUp after this modification and applying all our optimizations. The benefit of the different levels of optimizations follow similar trends as seen in Table 3.3, hence we have not reported them separately.

3.5.3 Evaluating Benefit of Integrating to IP-Core Generator

Generating pipelined designs with FloPoCo

If we now want to improve the throughput of our design, we need to pipeline it. Unfortunately, LegUp 3.0 does not support pipelining floating-point computation. While future versions of the tool may remove this limitation, it is still an excellent example where selective integration to another tool may be beneficial. To overcome this limitation, we added a VHDL code-generator to the scalar-level optimization module. Additionally, to generate pipelined floating-point operators, we integrated FloPoCo [De Dinechin and Pasca, 2011], an open-source arithmetic core generator, into our flow. During code-generation, our tool directs FloPoCo to create the necessary floating-point cores. Our code-generator uses the pipelining information returned by FloPoCo to generate the glue-logic that connects the individual cores through registers, implementing a correctly pipelined datapath; the generated designs were verified using circuit simulation. As shown under **Exp5** on Table 3.5, this design has a much higher throughput because of the pipelined architecture, but also needs significantly higher area. If the target application only needs floating-point operators of smaller precision, we can modify two configuration parameters in our HLS flow to generate custom cores from FloPoCo that trade-off precision for area. By using floating-point operators that have a 10-bit mantissa and 5-bit exponent, we get the results shown under **Exp6** on Table 3.5.

These results illustrate how our approach can be used to perform higher order and domain-specific optimizations that typically go untapped in general purpose HLS tools. Our examples are simple and well understood, but the whole approach is generally applicable. We showed that performing optimizations at different abstraction levels can progressively improve the result. We also demonstrated the flexibility and reuse potential of our approach by integrating different tools and reusing our optimization modules to create different output designs in a cost-effective manner.

3.6 Discussion

Key insights

Domain-specific hardware synthesis tools can make reconfigurable technology more accessible to domain experts who have little hardware design knowledge. In this chapter, we have presented an approach to significantly reduce the effort needed to develop such tools. To achieve this, we use Scala and LMS as a common platform to develop standalone optimization modules that can be easily reused across different HLS flows. We further reduced the design effort by integrating with external tools, when possible, and building the domain-specific HLS tool incrementally. To illustrate this approach, we developed a simple tool to synthesize matrix expressions into hardware. The evaluation results in Section 3.5 show that our methodology is flexible to accommodate changes, needs low-development effort, and is able to effectively leverage domain-specific optimizations to produce better designs. As we demonstrated in the case of LegUp, our approach can also be used to augment existing general-purpose tools with additional domain-specific optimizations, enabling them to produce better results.

What is missing?

The techniques described in this chapter are useful for developing DSLs and performing domain-specific optimizations. However, the incremental effort needed to develop such tools can be further reduced if they can share a common hardware generation infrastructure to implement the domain-specific operators. Additionally, since the people developing new domain-specific tools might be domain experts or people with software development background who do not have any hardware design expertise, to remain widely usable, this shared infrastructure must not demand much hardware design knowledge. We will explore the feasibility of developing such a shared infrastructure in chapter 4.

4 Hardware System Synthesis from Computational Patterns

The effort needed to develop new domain-specific tools can be considerably reduced if they can share a common hardware generation infrastructure. Since developers of new domain-specific tools might be domain-experts or software developers, for successful adoption, this shared infrastructure must not require hardware design knowledge to use. However, off-the-shelf HLS tools do not satisfy this requirement. Most of these HLS tools accept input specifications in high-level languages, such as C, C++ or SystemC and, thereby, provide a more convenient programming interface for designing hardware. But, to develop high-quality designs with these tools, the user still needs to manually perform optimizations that require a detailed knowledge of the tool and the generated hardware design as well as the implementation target. This requires a substantial amount of hardware design knowledge and, consequently, makes these tools unsuitable for non-hardware-experts.

4.1 Motivation

HLS tool users require hardware design expertise

In order to illustrate this problem, consider developing a simple hardware design that adds 512 integers held in an external memory and stores back the result. This design can be synthesized using an off-the-shelf HLS tool from the C++ program shown in Figure 4.1a. But, this program does not consider aspects of the underlying system architecture (e.g., the maximum data-width of the memory interface, the communication modes on this interface, i.e., burst mode vs. individual accesses, and the available parallelism in each data word from memory) and does not fully leverage the features in the HLS tool to exploit this

```
void sum(int* mem){
  mem[512] = 0;
  for(int i=0; i<512; i++)
     mem[512] += mem[i];
}
```

(a) Unoptimized HLS Program; Execution Time = 27,236 clock cycles

```
// Width of MPort = 16 * sizeof(int)
#define ChunkSize (sizeof(MPort)/sizeof(int))
#define LoopCount (512/ChunkSize)
// Maximize data width from memory
void sum(MPort* mem){
   // Use a local buffer and burst access
   MPort buff[LoopCount];
   memcpy(buff, mem, LoopCount*sizeof(MPort));
   // Use a local variable for accumulation
   int sum=0;
   for(int i=0; i<LoopCount; i++){
   // Use additional directives where useful
   // e.g. pipeline and unroll for parallel exec.
   #pragma PIPELINE
      for(int j=0; j<ChunkSize; j++){
      #pragma UNROLL
         sum+=(int)(buff[i]>>j*sizeof(int)*8);}}
    mem[LoopCount]=sum;
}
```

(b) Optimized HLS Program; Execution Time = 302 clock cycles

```
// Here, data_array is an array of 512 integers.
// sum adds its elements and the stores back the result
val result = data_array.sum()
```

(c) DSL Program; Execution Time = 368 clock cycles

Figure 4.1: Comparing optimized HLS, unoptimized HLS and DSL specifications. All three programs produce hardware to perform the same computation. The optimized specification leverages on a detailed knowledge of the HLS tool, the resulting hardware and the implementation platform while performing optimizations to improve the performance. The DSL code is much simpler to express and yet provides comparable performance.

parallelism; therefore, the generated hardware is extremely inefficient. The hardware design synthesized from the program in Figure 4.1a, using Vivado HLS (2013.4) [Xilinx, 2013a] with the highest level of optimization, needs 27,236 clock cycles on our test platform to complete the computation. To achieve good performance, the developer needs to write the more complex program shown in Figure 4.1b; this program considers the relevant system-level aspects and exploits the available data parallelism to generate a more efficient hardware design that performs the same task in 302 clock cycles. Moreover, many HLS tools will only synthesize these programs into standalone hardware modules and not a complete design that includes necessary external connections to board-level interfaces and peripherals, such as the instantiation of the memory controller and the connection to the external memory in our example. This forces application developers to make these connections manually and sometimes even generate the essential clock and control signals for the module to obtain a complete design. So, while HLS tools are capable of generating good quality designs and can provide convenient programming interface to enhance developer productivity, in practice, they are difficult to use for application developers who often lack the necessary hardware design skills.

Computational patterns: productivity and performance without hardware design skills

To overcome these limitations, we propose an automated methodology to generate complete hardware systems from programs written in a high-level *Domain-Specific Language* (DSL) using structured computational patterns. In this methodology, as illustrated in Chapter 3, we first leverage the application domain-knowledge and the domain-specific semantics of the DSL to perform optimization. After optimization, the domain operations are mapped to a set of structured computation patterns, such as `map`, `reduce`, `foreach` and `zipwith`. These computation patterns are algorithmic methods that capture the pattern of computation and communication and, therefore, can be easily used without any hardware design expertise. Additionally, they have well defined properties that enables us to create premeditated strategies to optimize them and generate high-quality hardware modules to implement them. For instance, the DSL code in Figure 4.1c is simple to express for the application developer and it will be mapped to a `reduce` pattern. This enables us to automatically generate a program that is similar to Figure 4.1b, and obtain a hardware module to perform the same computation in

368 clock cycles[1]. Furthermore, since each DSL targets a specific application domain, we can have a set of predefined system-architecture templates that are suitable for applications in that domain. By utilizing these templates, we can autonomously interconnect the different hardware modules in the application, generate necessary control signals, and obtain a complete design that is ready for FPGA bitstream generation. Consequently, this approach enables application developers to generate hardware designs from high-level functional specifications without having to meddle with any hardware level details.

Chapter outline

The rest of this chapter discusses the details of this proposed methodology, starting with a general overview in Section 4.2. We look at the compilation of high-level application programs into computational patterns in Section 4.3 and discuss how these patterns are generated into optimized high-performance hardware systems in Section 4.4. In Section 4.5, we first evaluate the quality of our generated computation patterns using a set of microbenchmarks and then assess the overall effectiveness of our flow using four applications written in OptiML [Sujeeth et al., 2011], a high-level DSL for machine learning. The results reveal that our optimizations significantly improve the performance of the generated hardware designs. In comparison with a laptop CPU, our automatically generated hardware achieves reasonable performance and a much better energy efficiency. Finally, we reiterate the benefits of the approach and summarize findings in Section 4.6.

4.2 Overview of the Methodology

Decomposing DSL application to computational patterns

Our automated methodology accepts an application program written in a high-level DSL and generates a complete hardware design that can be programmed on the target FPGA. Figure 4.2 illustrates the steps in this process. In this methodology, the application program is a purely functional specification and writing it in a DSL offers higher productivity and shields the application developer from the hardware-level details; at the same time, it reveals the specific application domain targeted by the developer to the compiler. Our *compiler infrastructure*, indicated as A in the figure, optimizes this DSL-program by performing both

[1]The performance of the automatically generated module is slightly lower compared to the one obtained from Figure 4.1b because it is more generic and designed for handling reductions of larger sizes.

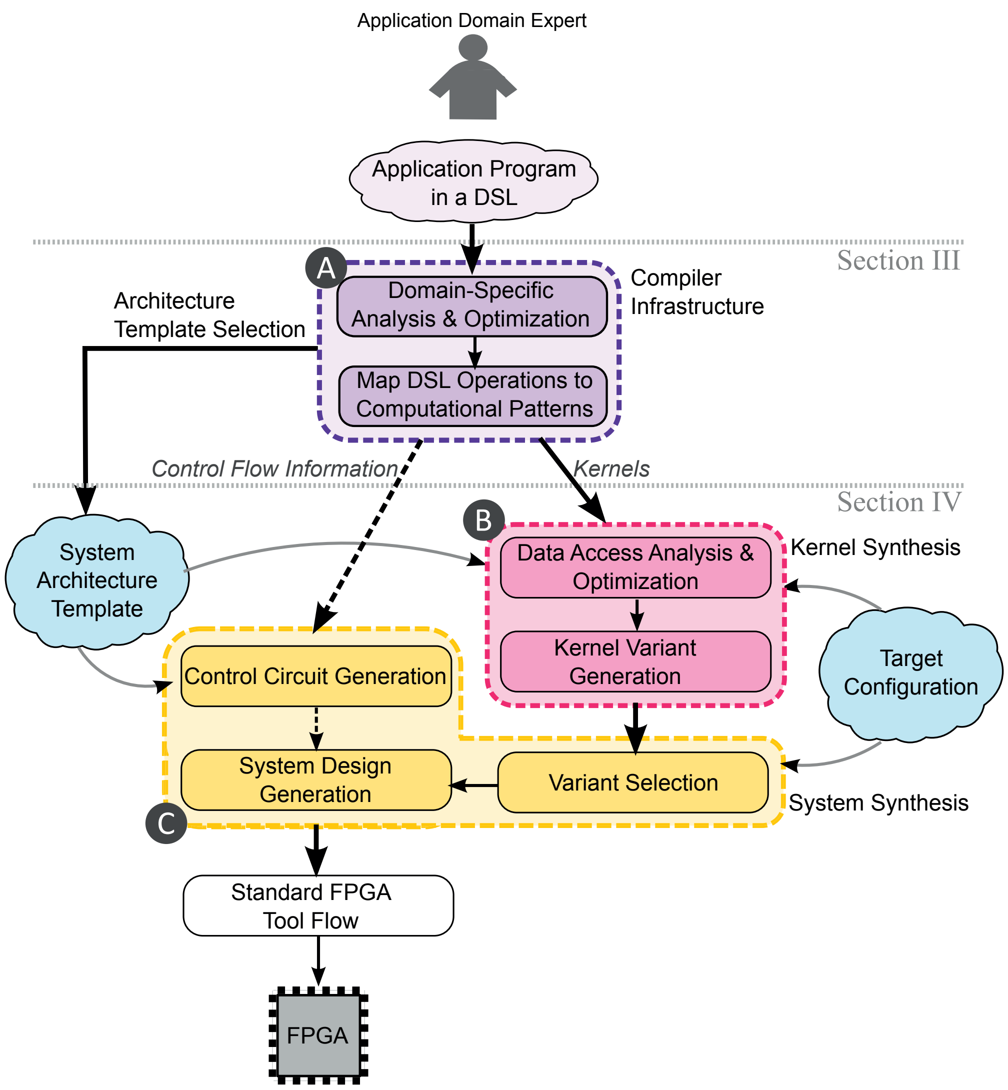

Application Domain Expert
Application Program in a DSL
Section III
A
Compiler Infrastructure
Domain-Specific Analysis & Optimization
Map DSL Operations to Computational Patterns
Architecture Template Selection
Control Flow Information
Kernels
Section IV
B
Data Access Analysis & Optimization
Kernel Synthesis
Kernel Variant Generation
System Architecture Template
Target Configuration
Control Circuit Generation
System Design Generation
Variant Selection
C
System Synthesis
Standard FPGA Tool Flow
FPGA

Figure 4.2: Overview of the methodology. This figure illustrates how the high-level specifications in a DSL is compiled and then automatically transformed into a hardware system that can be implemented on an FPGA.

domain-specific optimizations (e.g., applying linear algebra simplification rules) and general-purpose optimizations (e.g., common subexpression elimination and dead-code elimination). After optimization, the compiler maps the operations in the DSL program as a composition of computational patterns. The compiler then regroups these patterns into *kernels* after performing additional fusion optimization [Rompf et al., 2013]; therefore, each kernel can contain one or more computational patterns, either fused together or nested inside one another. Additionally, the compiler represents the complete application program as a dependency-graph between these separate kernels. The computational patterns, as we will see, have well defined properties, such as parallelism, data-access behavior and inter-operation dependences, which enables the tool to generate efficient and high-performance hardware implementations for them. Based on the domain of the DSL, the compiler selects a suitable *system-architecture template* for the final hardware implementation. This architecture template, which is inspired from approaches such as BORPH [So, 2007] and LEAP [Parashar et al., 2010], delineates how the various hardware components will be interconnected while composing the final system design.

Synthesizing computational patterns into hardware modules

The hardware generation process starts with the *kernel-synthesis* step, denoted as B in the Figure 4.2, which takes the kernels in the application program and generates concrete implementations (i.e., either hardware or software modules) for them. To facilitate this, as shown in the figure, system-architecture template provides information about the shared components in the system as well as the information about the interfaces to the generated hardware modules. During this step, the kernels containing some structured parallelism are generated into hardware modules and those without parallelism are generated into software modules to be executed on a microprocessor. The kernel-synthesis step also gets information about the specific FPGA platform used as the implementation target, such as clock frequency constraints, size of external memory and the available device resources (e.g., look-up tables, flip-flops, block RAMs and DSP units), from the *target-configuration*; this information is used to ensure that the generated kernel implementations are compatible with the selected target. For each parallel kernel, the kernel-synthesis step produces multiple hardware implementations, which we call *variants*, that achieve different trade-offs between area and performance. Generating multiple variants during the kernel-synthesis step is essential in order to enable the subsequent system-synthesis step to compose a system design that will achieve good performance and still fit within the limited resources available on the target FPGA.

Composing hardware modules into a system design

System-synthesis, indicated as C in the Figure 4.2, uses the information from the system-architecture template to compose the complete system design. This step uses the target-configuration to know the capabilities of the chosen FPGA and selects specific variants to implement each parallel kernel in the application. The control circuitry for this system design is also automatically generated based on the application program's dependency graph produced by the high-level compiler. This complete system design is provided to a standard FPGA tool flow, i.e., logic synthesis, place and route and bitstream generation, to produce the bitstream to program the target FPGA device.

4.3 Compiling DSL-Programs to Patterns

In this section, we will focus on the high-level compiler infrastructure we use and discuss how it compiles high-level DSL programs into computational patterns.

4.3.1 Compiler Infrastructure

Delite: the high-level compilation infrastructure

We implement our compiler infrastructure, indicated as A in Figure 4.2, by extending the Delite [Lee et al., 2011] compiler framework. Delite is an extensible compiler framework that makes it easy to develop DSLs targeting heterogeneous systems. The core idea of Delite is to provide DSL developers with a set of structured computation patterns and data structures that can be extended to implement domain operations and custom data structures needed in the DSL. Delite currently supports computation patterns such as `map`, `reduce`, `zipwith`, `foreach`, `filter`, `group-by`, `sort` and `serial`; among them, `serial` is used for non-structured computations that cannot be parallelized. For managing data, Delite provides scalar datatypes, multi-element datatypes such as `array`, `vector`, `matrix` and `hashmap` that are called *collections*, and user-defined compositions of these datatypes. The structured nature of Delite components facilitates parallelizing and optimizing DSL programs for different target architectures such as multi-core CPUs and GPUs. DSL developers using Delite can easily add domain-specific optimizations that leverage the detailed domain-knowledge to perform optimizations and they automatically get the generic optimizations, such as loop fusion and data structure transformations, that are already built into Delite to produce high-performance implementations. When a DSL program is compiled with Delite, it produces 1) a set of kernels that are each composed out of one or more computational patterns and 2) a dependency-graph between these kernels.

Generating hardware designs from Delite-patterns

In order to a generate hardware system for a DSL program, we extended Delite to generate concrete implementations for the kernels found in the program. As we will see, we added the ability to generate hardware modules to implement some kernels by producing an optimized input program for an HLS tool; other kernels were generated as software modules and executed on a processor on the FPGA. Finally, we use the program's dependency-graph to generate a controller that guarantees that these kernels are executed in a valid order. We will discuss these extensions in detail in Section 4.4. In order to demonstrate the approach, we generate hardware systems to implement applications written in OptiML [Sujeeth et al., 2011], a machine learning DSL implemented using Delite. Among the Delite components OptiML supports, we limit ourselves to the most widely used computation patterns, i.e., `serial`, `map`, `zipWith`, `reduce` and `foreach`, and data structures, i.e., scalar datatypes and collections, such as `array`, `vector`, `matrix` and user-defined compositions of these, which can be efficiently implemented on the FPGA. One thing to note, however, is that our toolchain is not limited to OptiML and can be directly used by other DSLs in Delite that use the computational patterns and data structures we currently support. Furthermore, our approach is quite general and can be used in other application development infrastructures that use the concept of computational patterns, such as Copperhead [Catanzaro et al., 2011], Intel's Array Building Blocks [Newburn et al., 2011] and FlumeJava [Chambers et al., 2010].

4.3.2 Computational Patterns

Properties of the computational patterns

Our tool flow generates hardware systems from application programs that are decomposed into structured computations patterns, such as `map`, `zipWith`, `reduce` and `foreach`, and non-structured computational patterns (i.e., `serial`). Within each computational pattern, we can have operations on primitive datatypes (e.g., `bool`, `int`, `float` and `double`), *collections* (e.g., `array`, `vector`, `matrix`) and other user-defined types. These computational patterns have a well defined properties such as the degree of parallelism in the operation, interdependences between elemental operations and data-access patterns.

Among them, the non-structured computational pattern (i.e., `serial`) offers no operation-level parallelism and operates only on primitive

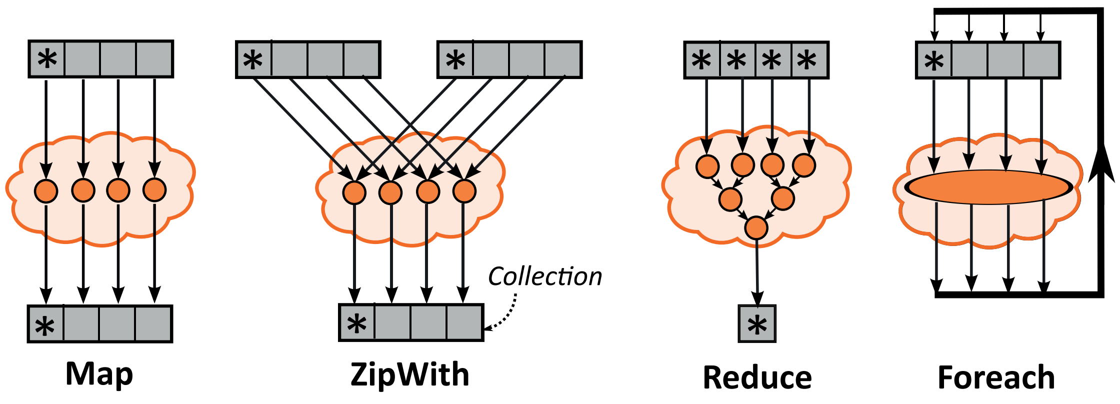


Figure 4.3: In the `map` and `zipWith` patterns, a pure function (i.e., side-effect free function) is used to create a new collection from one or more collections, respectively. The `reduce` uses a binary operation that is associative and commutative to compute a single value from a collection. And, the `foreach` uses an impure function (i.e., function not having side effects) to update the values of an existing collection.

datatypes; consequently, the kernels composed of this pattern have no parallelism and are called *serial kernels*. However, as illustrated in Figure 4.3, the structured computational patterns we support operate on collections, in addition to primitive datatypes, and have parallelism in their operation; the kernels composed of these patterns, naturally, have parallelism and are called *parallel kernels*. As seen in the figure, the `map` and `zipWith` patterns always use a pure (side-effect free) function to create a fresh collection. The difference between them is that `map` has a single input collection while the `zipWith` has multiple input collections. So, squaring each element in a `vector` uses a `map` while adding two `vectors` requires a `zipWith`. The `reduce` pattern computes a single element by applying a binary function that is both associative and commutative to all the elements in a collection; so, finding the minimum or maximum values in an array are great examples for this pattern. The `foreach` pattern is typically used to modify values in an existing collection by applying an impure function (with side-effects) to each element, such as to set all the negative numbers in an array to zero. However, the programming model we use in Delite restricts `foreach` to guarantee that this pattern can be executed in parallel without data races.

During the kernel-synthesis step, B in Figure 4.2, we leverage these well defined properties of these computational patterns to generate efficient implementations for them.

```
1  // User inputs for element count, normalized-min and normalized-
      max
2  val count = args(0).toInt
3  val minValue = args(1).toInt
4  val maxValue = args(2).toInt
5
6  // Some function used to initialize the data to be normalized
7  val data = (0::count){i => ((4*i+2*i+i) % 2048)}.mutable
8  // Compute the min and max from the initialized data
9  val min = data.min
10 val max = data.max
11 // Normalize the data using the user provided min and max value.
12 for(i <- (0::count)) { data(i) =
13         (data(i)-min)*(maxValue - minValue)/(max-min) + minValue
14   }
15 // Print the normalized data
16 data.pprint
```

Figure 4.4: Normalization application written in OptiML. This OptiML-application accepts three user inputs, `count`, `minValue` and `maxValue`, an uses this data to generate an array and normalize its values to lie within the user provided range (between `minValue` and `maxValue`). The functional specifications in the program is automatically decomposed into `serial`, `map`, `reduce` and `foreach` patterns that are then generated into a hardware system design.

4.3.3 OptiML Application Example: Normalization

Case study: compiling a DSL application with Delite

To illustrate how the patterns are extracted and optimized by the high-level compiler, we use the example of a simple OptiML-application to perform normalization that is shown in Figure 4.4. This application first reads some user provided parameters, such as the `count`, `minValue` and `maxValue`. Based on these parameters, it initializes a data-set, computes the minimum and maximum values in the data-set, normalizes the data-set to be within the interval [`minValue`,`maxValue`] and prints out the normalized values. The DSL operations in this application are mapped to appropriate computational patterns as defined by the DSL developer. Here, the operation to read user parameters, on lines 2 to 4 and the print operation on line 15 are instances of `serial` pattern that become serial kernels. The data-set initialization on line 7 is an instance of a `map` pattern, the minimum and maximum computation on lines 9 and 10 are translated as `reduce` patterns, and the normalization operation on lines 12 becomes a `foreach` pattern. Furthermore, since the minimum and

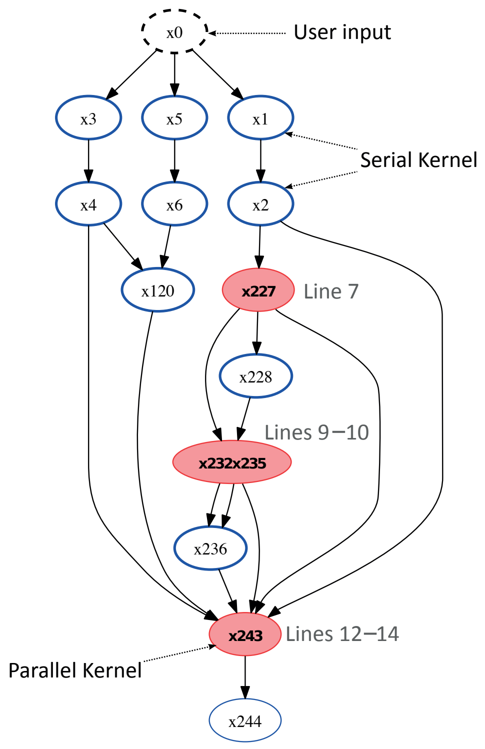


Figure 4.5: Dependency graph for the normalization application. This graph shows the kernels (graph-nodes) and kernel-dependency (graph-edges) information generated by our high-level compiler. Here, kernel `x0` stands for the user inputs. `x227`, `x232x235` and `x243` are parallel kernels that are generated from structured patterns, `map`, `reduce` and `foreach`, respectively; the line numbers in Figure 4.4 that generated these kernels are indicated next to the kernels. All other kernels are generated from `serial` patterns. Among the parallel kernels, `x232x235` is generated from applying loop-fusion on the `max` and `min` operations in the DSL program.

maximum computations have no interdependences and operates on the same iteration range, the compiler applies loop-fusion optimization to merge the two operations into a single `reduce` pattern and benefit from data-reuse.

The compiler also produces the dependency-graph shown in Figure 4.5 from the application program. The nodes in this graph represent kernels in the program and edges represent data or control dependencies between these kernels. As seen in the figure, the compiler produces three parallel kernels and set of serial kernels from the normalization application. These kernels are generated into concrete implementations in the kernel-synthesis step discussed in Section 4.4.

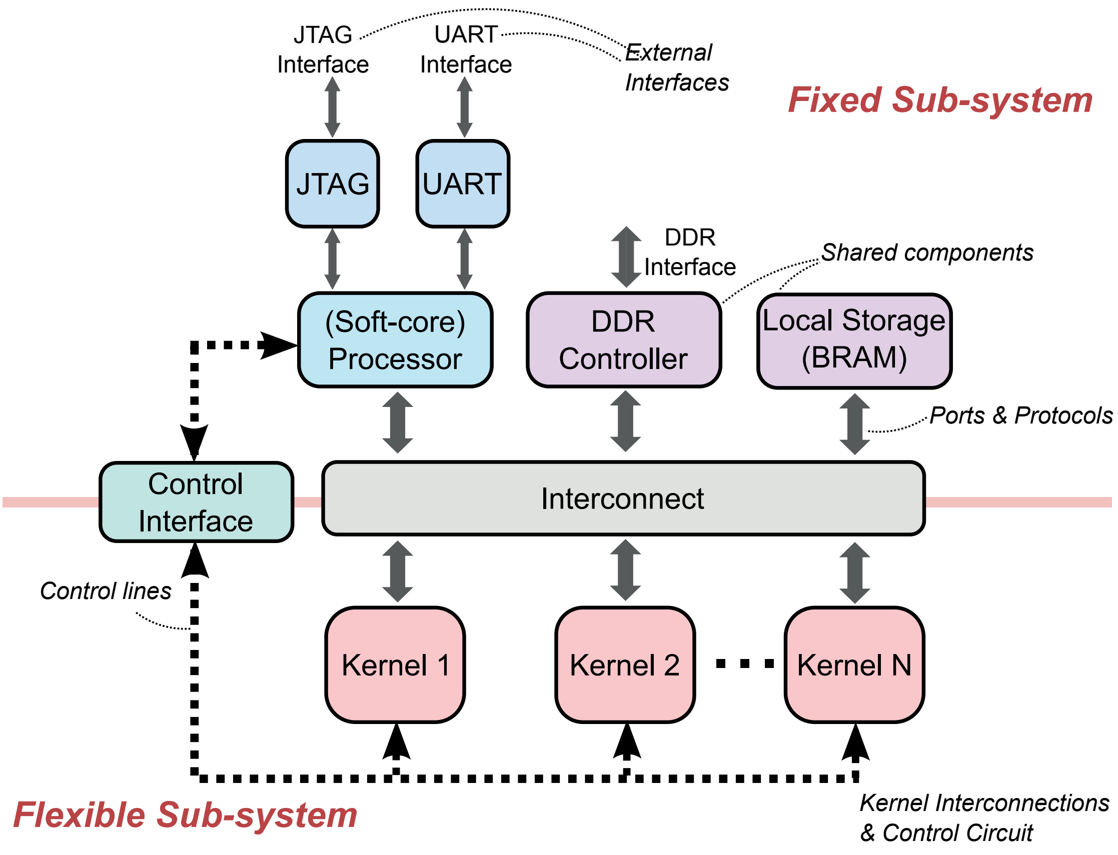


Figure 4.6: This figure shows the system-architecture template used for hardware systems generated from OptiML applications. The annotations in the figure are some of the specific details this template provides to the hardware generation process.

4.4 Hardware Generation from Patterns

In this section, we will discuss how the kernels extracted from the application and its associated dependency-graph are automatically transformed into a hardware design.

4.4.1 System Architecture Template

Composition of the system architecture template

The hardware design generation starts with the selection of a system-architecture template by the compiler. This template delineates the various components and interfaces available in the final system as well as their interconnections. Hence, it provides a general outline based on which the hardware system for the application will be generated.

The template we use is composed of two parts: the fixed subsystem, which remains constant for every application using that template, and the flexible subsystem, which changes from one application to another. Figure 4.6 shows the system-architecture template we use for all OptiML applications. Here, the fixed subsystem includes a processor, on-chip memory, external memory controller, and external interfaces to the DRAM, UART and JTAG. The flexible subsystem defines how the kernels that are synthesized into hardware modules will be connected to the rest of the system and how the control infrastructure will be generated for them. We currently use variations of this system-architecture template for generating designs for different FPGA devices, e.g., Virtex 7 device with only reconfigurable logic and Zynq device that contain a hardened processor core and reconfigurable logic. However, the methodology supports having multiple templates that are very different, in which case the selection of the appropriate template will be done according to the specifications of the DSL developer.

4.4.2 Kernel Synthesis

Inputs the kernel synthesis step

The kernel-synthesis step generates concrete implementations for the kernels in the application; it is marked as B in Figure 4.2. As shown in the figure, the inputs to this step are 1) the set of kernels from the compiler, 2) the information about the complete hardware system from the system-architecture template, and 3) the information about the implementation target from the target-configuration. The system-architecture template provides details, such as the number of data-ports to each kernel, communication protocols on these ports, shared memories in the design, that are necessary to ensure that the synthesized kernels will function correctly and can be easily integrated into the final design. The target-configuration specifies additional details, such as the sizes and address ranges of the available memories, the bitwidth of the ports and target-specific resource constraints which help to tailor the generated kernels to the specific FPGA used in the implementation.

Mapping `serial` kernels on the FPGA

As seen in Section 4.3.3, OptiML applications can contain multiple serial and parallel kernels. The serial kernels offer only limited opportunities for acceleration using custom hardware. Therefore, we map all the serial kernels in the application to the (soft-core) processor in the system-design template to share implementation resources among them.

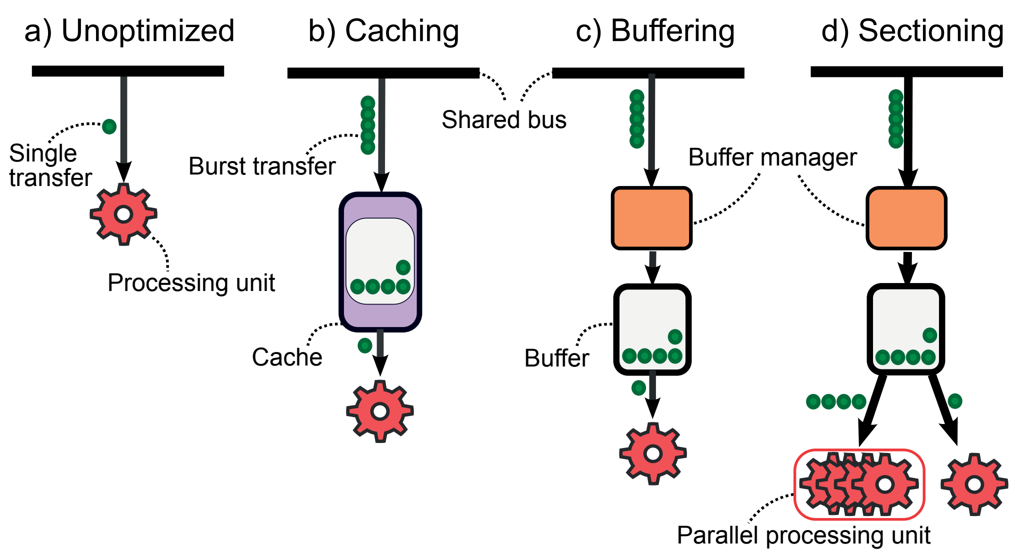


Figure 4.7: Optimizations applied to the kernels. The unoptimized kernel access each data element separately and, therefore, has a low effective bandwidth to the data (a). When accesses are sequential, we can improve this bandwidth using burst transfers by adding a local cache (b) or by using local buffer and a buffer manager (c). To benefit from this improved bandwidth, we need to generate the kernel differently to correctly leverage the available data parallelism (d).

Mapping parallel kernels (i.e., `map`, `zipWith`, `reduce` and `foreach`) on the FPGA

The parallel kernels, however, can benefit greatly from custom hardware that can aptly exploit the available parallelism. Moreover, in typical OptiML applications, these parallel kernels dominate the overall execution time. Since these parallel kernels are generated from a limited number of computational patterns that are supported by the compiler, we can have premeditated strategies to generate high-quality hardware implementation for them. Although the set of computational patterns are small, actual computation and data access properties in the kernels will still vary significantly from one application to another. Additionally, within each kernel, these patterns can be nested within one another or fused together, as we saw in the case of the normalization application. Due to this large variability, we cannot use fixed templates for each pattern. Instead, we leverage the properties of the computational patterns and perform additional compiler analysis to identify optimization opportunities and produce a well formatted code for an HLS tool. This code contains the necessary optimization directives that guides the HLS tool to generate a high-quality hardware implementation for the kernel.

4.4.3 Kernel Optimization

Generic HLS optimizations (e.g., loop-unrolling, loop pipelining) alone are insufficient

While synthesizing parallel kernels, we can use generic optimizations, such as loop-unrolling and loop-pipelining, to generate parallel hardware structures. However, to generate high-performance designs, we need to analyze the properties of the individual kernel and perform additional optimizations. For instance, consider the lines 9 and 10 of the normalization application in Figure 4.4. After optimizations in the high-level compiler, these two DSL-operations are implemented as the fused kernel, `x232x235`, shown in Figure 4.5. The hardware implementation of this kernel will read will read `data`, a large `vector` data structure stored in the external memory, sequentially, compute the maximum and minimum values and store the results as `max` and `min`. A relatively straightforward implementation of this module produced from a HLS tool will read each element from the `data` separately and perform the computation, as shown in Figure 4.7a.

Instantiate a cache for sequentially accessed data structures

However, since the elements in `data` are accessed over a shared bus, each read transaction entails overheads due to the bus protocol and the latency of the external memory. But, since the kernel accesses this data structures sequentially, we can use burst communication to reduce these overheads. To implement this, we first analyze the data access pattern of each data structure used in the kernel and add a local cache to those that are accessed sequentially, e.g., `data` in kernel `x232x235`. Now, as depicted in Figure 4.7b, data requests from/to this data structure is served from the local cache and, in the event of a cache-miss, the cache is filled/flushed using burst-transfer from/to the external memory.

Utilize a local buffer and buffer manager to improve data-bandwidth

Although the cache enables us to use burst-transfers, having to check for cache hit/miss on every access incurs some performance overheads. To avoid this, we can perform a more detailed analysis of the access patterns, and use this information to replace the cache with a simple local buffer and an associated buffer manager, as shown in Figure 4.7c. The difference between the two is that buffer manager *knows* the access pattern to the data structure and uses this information to deliberately move data from/to the local buffer without having to check on each access. To implement this, as done in the case of the cache, we use compiler analysis to identify data structures that are sequentially accessed from the kernel and then generate a local buffer and associated buffer manager for them. However, unlike in the case of the cache, we modify only some sequential accesses to utilize this buffer and the buffer management is performed based on the data requirements of specific accesses. When there are multiple sequential accesses from the kernel to the same data structure, the access that occurs most frequently (i.e., occurring at the inner-most loop-level) is given priority to use this buffer. Additionally, when the buffer holds valid data, all other read accesses to the same data structure, including the non-sequential ones, will first check the buffer before going to the shared memory. Similarly, all other write accesses from the kernel that occur when the buffer hold valid data will use the buffer similar to a write-through cache with no-write-allocate policy. Accesses that occur when the buffer does not hold valid data (i.e., accesses in the kernel code that are outside the scope of the buffer management code) will directly access the shared memory. In the case of using the cache, these accesses would have caused cache-pollution [Handy, 1998] and deteriorated the performance of the application; therefore, by using the local buffer and buffer manager, we overcome this problem.

Perform loop-sectioning to improve parallelism in computation

Using the cache and local buffer can improve the data bandwidth to the hardware module. We will still not get the most out of this higher data bandwidth by only depending on optimization directives in the HLS, such as loop-unrolling and loop-pipelining. To obtain better computational throughput, in addition to using these directives, we need to refactor the kernel computation into multiple sections that are each specialized to exploit the different amounts of parallelism available in the input data; this is the same as the loop-sectioning optimization done for SIMD processors. After applying this optimization, as shown in Figure 4.7d, the generated hardware will selectively use dedicated parallel processing units when the input data has sufficient parallelism to improve the overall processing throughput. In our example, when possible, the parallel processing unit will read entire lines of data from the local buffer and utilize balanced reduction trees to compute the results and will fall back to the less parallel unit when there is insufficient data to feed the parallel unit.

All the aforementioned optimizations were applied in our tool flow while generating the input program for the HLS tool. To enable these optimizations, we added additional compiler analysis into Delite. Figure 4.8 depicts some of the key data-structures that were added facilitate these compiler analyses. Among them, the `LoopInfo` holds the

Algorithm 4.1: Identifying Sequential Accesses in the Kernels

LOOP_INVARIANT function checks if a given symbol is an invariant in the current loop. It takes as parameters, the symbol to check (sym) and the current loop counter ($loop\_iter\_sym$).

```
function LOOP_INVARIANT($sym, loop\_iter\_sym$)
    ▷ Symbol in invariant if it is never modified in the loop-body
    if $sym$ is a constant value then
        return true
    else if $sym$ is same as $loop_i ter_s ym$ then
        return false
    else if value of $sym$ is never updated inside the loop-body then
        return true
        ▷ Else, check if it is computed using other invariant symbols
    else
        $V \leftarrow$ Set of symbols used to compute the value of $sym$
        $invariance \leftarrow$ true
        for all $v \in V$ do
            $sym\_invar \leftarrow$ LOOP_INVARIANT($v, loop\_iter\_sym$)
            $invariance \leftarrow invariance \wedge sym\_invar$
        end for
        return $invariance$
    end if
end function
```

CHECK_SEQUENTIAL_ACCESS function checks if accesses using a given indexing symbol will sequentially access the elements in a data structure. It takes as arguments, the indexing symbol ($index\_sym$) and the current loop counter ($loop\_iter\_sym$).

```
function CHECK_SEQUENTIAL_ACCESS($index\_sym, loop\_iter\_sym$)
    ▷ Access is sequential if the index is the loop-counter
    if $index\_sym$ is same as $loop\_iter\_sym$ then
        return true
        ▷ Or, it is computer as loop-counter + loop-invariant symbol
    else if $index\_sym$ is computed as ($a$+$b$) in the IR then
        $a, b \leftarrow$ symbols used to compute $index\_sym$
        $a\_seq \leftarrow$ CHECK_SEQUENTIAL_ACCESS($a, loop\_iter\_sym$)
        $b\_invar \leftarrow$ LOOP_INVARIANT($b, loop\_iter\_sym$)
        $b\_seq \leftarrow$ CHECK_SEQUENTIAL_ACCESS($b, loop\_iter\_sym$)
        $a\_invar \leftarrow$ LOOP_INVARIANT($a, loop\_iter\_sym$)
        return $(a\_seq \wedge b\_invar) \vee (a\_invar \wedge b\_seq)$
    else
        return false
    end if
end function
```

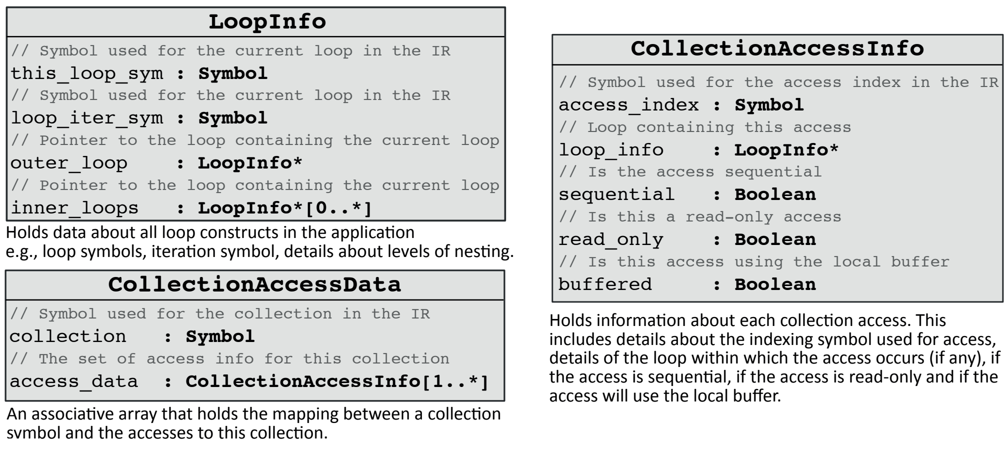
Algorithm 4.2: Selecting Data Structure Accesses to Use the Local Buffer

BUFFER_ACCESSES function selects the data structure accesses that will use the local-buffer. It takes as arguments the symbol to the data structure ($data\_sym$) and the loop info of current loop ($loop\_info$).

```
function BUFFER_ACCESSES($data\_sym$, $loop\_info$)
    ▷ We need to use the local-buffer for most frequent access
    ▷ Check if an inner-level loop can buffer this data structure
    $buffered\_at\_inner \leftarrow$ false
    for all $inner\_loop \in loop\_info.inner\_loops$ do
        $buffered \leftarrow$ BUFFER_ACCESSES($data\_sym$, $inner\_loop$)
        $buffered\_at\_inner \leftarrow buffered\_at\_inner \lor buffered$
    end for

    ▷ If an inner-level loop can use the buffer, do nothing
    if $buffered\_at\_inner$ then
        ▷ Return false since not accesses were buffered
        return false

        ▷ Else, select the most frequent sequential accesses to buffer
    else
        ▷ Select the set of most frequent sequential accesses
        $A \leftarrow$ Get sequential accesses on given $data\_sym$
        $B \leftarrow$ Group $A$ based on the index symbol used for the access
        ▷ Note: $C$ will be $\emptyset$ if $A$ or $B$ is $\emptyset$
        $C \leftarrow$ Select group in $B$ with highest element count
        ▷ Set the selected accesses to use the local-buffer for buffering
        for all $c \in C$ do
            Set access in $c$ to buffered
        end for
        if $C$ is $\emptyset$ then
            ▷ Return false since no access will use the local-buffer
            return false
        else
            ▷ Return true since some accesses will use the buffer
            return true
        end if
    end if
end function
```



```
LoopInfo
// Symbol used for the current loop in the IR
this_loop_sym : Symbol
// Symbol used for the current loop in the IR
loop_iter_sym : Symbol
// Pointer to the loop containing the current loop
outer_loop    : LoopInfo*
// Pointer to the loop containing the current loop
inner_loops   : LoopInfo*[0..*]
```

Holds data about all loop constructs in the application
e.g., loop symbols, iteration symbol, details about levels of nesting.

```
CollectionAccessData
// Symbol used for the collection in the IR
collection   : Symbol
// The set of access info for this collection
access_data  : CollectionAccessInfo[1..*]
```

An associative array that holds the mapping between a collection
symbol and the accesses to this collection.

```
CollectionAccessInfo
// Symbol used for the access index in the IR
access_index : Symbol
// Loop containing this access
loop_info    : LoopInfo*
// Is the access sequential
sequential   : Boolean
// Is this a read-only access
read_only    : Boolean
// Is this access using the local buffer
buffered     : Boolean
```

Holds information about each collection access. This
includes details about the indexing symbol used for access,
details of the loop within which the access occurs (if any), if
the access is sequential, if the access is read-only and if the
access will use the local buffer.

Figure 4.8: Data structures used to perform program analysis. This figure shows the three main data structures that were added to perform the program analysis and determine the data structures that will use the local-buffer. In the figure, `LoopInfo` structure held information about the loop structure in the kernel. `CollectionAccessInfo` held details information all the accesses to the different data structures in the program. This includes computed details such as if access pattern is strictly sequential and if the data-structure was buffered using the local buffer. `CollectionAccessData` is an associative array that holds the association between the symbols for data structures in the program and the accesses on these data structures.

details about the loop structure in the kernel. It contains details such as the symbol for the loop in the IR, the loop iteration counter (loop-counter), the outer loop that contains the current loop and the set of loops inside the current one. The `CollectionAccessData` is an associative array that maintains information about all the data structures (i.e., collections) in the program and all the data accesses associated to these data structures. The information about each data structure (i.e., collection) access within the program is stored in separate instances of `CollectionAccessInfo`. This includes the index used for the access, the details of the loop (i.e., pattern) from which the access occurs, if the access is sequential, if the operation is a read and if the access will use the local-buffer. To determine which data structure accesses are sequential, we use the CHECK_SEQUENTIAL_ACCESS function whose algorithm is given in in Figure 4.1. Note that the check for sequential accesses and loop-invariance is much simpler because it only considers the restricted scenarios that occur in the IR constructed from computational patterns; this is simplified analysis is another benefit that comes from decomposing applications into computational patterns. Based on the information of sequential accesses to the data structures, we utilize the BUFFER_ACCESSES function to determine the accesses that can benefit the most from using a local-buffer along with a a custom buffer manager. The simplified algorithm for the BUFFER_ACCESSES function given in Figure 4.2.

While all these optimizations improve the performance of the hardware kernels, but they also consume more resources. To implement an application on the FPGA, we need to partition the limited resources on the device among all the hardware kernels in the application. To facilitate this, during kernel-synthesis, we generate multiple hardware variants for each kernel in the application by applying these optimizations in different combinations. The selection of the specific variants to use in the final design is deferred to the system-synthesis step.

Data management

Memory allocation for primitive datatypes and collections

The hardware kernels generated from OptiML use scalar datatypes or collections, such as `array`, `vector`, `matrix` and user-defined compositions of these to exchange data with each other. Among them, the scalar datatypes have predefined sizes and the space needed to store them is much smaller than the available on-chip memory capacity. So, they are statically allocated in a shared on-chip memory to reduce the data-access latency and make it easy to access this data from the different kernels. In the case of the collections, they each contain a small amount of metadata, which include details such as the length, stride and number of rows/columns for matrices, and a contiguous block of raw-data. The size of this metadata is fixed and known at compile-time and, therefore, it is statically allocated in the on-chip memory. The raw-data, however, is typically very large and its size is known only during the runtime. Therefore, it is dynamically allocated in the larger shared external memory during the application execution. To perform dynamic allocation, this external memory is managed as a single circular buffer with fresh allocations happening only at the head of this buffer and deallocation occurring both at the head and tail. To dynamically allocate memory from the kernels, we store the *head-pointer* of this circular buffer in the shared on-chip memory; this head-pointer points to the first free location at the head of the circular buffer and marks the beginning of the unallocated section of the buffer. During the application run, each kernel allocates memory for new data structures by updating the value of this head-pointer. Memory deallocations can be performed either by the kernel, when the data structure is known to be only valid within the execution context of that kernel, or by the control circuit that is aware of the full application context and, therefore, knows when a data structure is not needed anymore and can be safely deallocated. The memory deallocation by the control circuit is covered in detail when discussing the control circuit generation.

4.4.4 System Synthesis

Inputs to the system synthesis step

The system-synthesis step, indicated as C in Figure 4.2, has three responsibilities: (1) selecting the kernel variants to use in the design; (2) interconnecting them to the other system components and interfaces; and (3) generating the control circuitry for the hardware design. The system-architecture template, which for OptiML applications is shown in Figure 4.6, provides this step with the information needed to generate the fixed subsystem and the strategy for generating the flexible subsystem. The target-configuration contains information about the resources on the target (e.g., LUTs, DSP blocks and BRAMs), the target-specific IP modules (e.g. DRAM controller, on-chip memory, soft-core processor) to use as well as their individual configuration parameters.

System Configuration

Determining the optimum system configuration

As noted earlier, the kernel-synthesis generates multiple variants for each parallel kernel in the application. The system-synthesis, therefore, needs to find the ideal *system configuration* to maximize the application's performance; the system configuration defines the specific kernel-variants that will be used to generate the hardware design. However, a valid system configuration must pick one variant per hardware kernel and, at the same time, ensure that all the selected kernel-variants for the given application will fit within the limited resources of the target FPGA. We formulate this as a knapsack optimization problem [Martello and Toth, 1990] where the objective is to maximize the performance and the constraint is to ensure that the system design can be implemented within the limited resources available on the target device.

In our flow, for each kernel we perform a design space exploration by generating all the different variants for it. Then, we utilize the performance and resource estimates produced by the HLS tool and use an *Integer Linear Programming* (ILP) solver to determine the optimal system configuration. However, the initial resource estimation in the HLS tool is based on models and often inaccurate. To overcome this problem, we use the feature in the HLS tool to execute the complete FPGA implementation flow and obtain more accurate estimates for resources and maximum clock frequency for each design. But, this process consumes a significant amount of time and makes exhaustive design space exploration for an application, sometimes, prohibitively long. In our toolchain, expert users can provide parameters to constrain the design space and limit the variants that will be explored for each kernel to expedite the process. Using the exploration data, we first prune designs that either exceed the resources of the target FPGA or will not run at the target clock frequency. The remain designs are used by the ILP solver to find the optimum system configuration.

Control Circuit Generation

Generating the control infrastructure in the hardware design

The control circuit for the hardware design is automatically generated from the dependency-graph of the application obtained from the compiler. This circuit performs two essential tasks: scheduling the kernel execution and freeing the dynamically allocated memory. To schedule the kernel execution, the dependency-graph of the application is analyzed to determine the control and data dependencies among the kernels. Based on this information, the generated schedule tries to maximize the performance by executing multiple kernels in parallel, when possible.

In order to free the dynamically allocated memory, during the application execution, the control circuit keeps track of the amount of memory allocated by each kernel. To achieve this, since the dynamically allocated memory is managed as a circular buffer, the control-circuit compares the values to the *head-pointer*, which always points to the first free location in the memory, before and after each kernel execution and can track these changes to determine the memory address range used to store that kernel's data. During the control circuit generation, variable lifetime analysis is performed by analyzing the data dependencies in the application's dependency-graph. Performing such a lifetime analysis is easy since Delite's dependency-graph provides clear and complete information about shared data structures in the application as well as the nature of a access to it from each kernel. The tool uses this information to statically determine the schedule for freeing memory blocks and generates this into the control circuit. Moreover, since the control circuit is always aware of the amount of memory used by the application, it can detect memory overflows and also terminate the application execution, if needed. For OptiML applications, this control circuit is implemented using the (soft-core) processor that executes the sequential kernels. Hence, the complete program for this processor is generated during the system generation step. To control the kernel execution from this processor, the system-architecture for OptiML applications, shown in Figure 4.6, includes a control interface module.

4.5 Evaluation Results

In this section, we will first evaluate the benefits of the optimizations discussed in Section 4.4 for each parallel pattern we can synthesize into custom hardware. Then, we will use four OptiML applications to illustrate the range of solutions, with different performance and resource utilizations, we can easily generate from our tool. Additionally, to provide a broader perspective on the quality of the generated hardware systems, we will compare their performance and energy efficiency to running the same applications on a laptop CPU.

4.5.1 Evaluation Setup and Methodology

Evaluation on the FPGA

All the hardware designs generated using the proposed methodology were written as OptiML applications. These were compiled using the Delite compiler which we modified for the purpose of generating hardware. During the compilation, the kernel-synthesis step generates multiple variants for each parallel kernel in the application and synthesize them into hardware modules using Vivado HLS (2013.4) [Xilinx, 2013a]. The system-synthesis step then automatically generates the final hardware design using the specific kernels variants in the system configuration. This step also generates the code for the (soft-core) processor in the design that executes the sequential kernels and controls the overall hardware execution flow. To measure the performance of the generated designs, they were synthesized into a bitstream using Xilinx Vivado Design Suite [Xilinx, 2013b] and implemented on VC707 and ZC706 development boards from Xilinx. The VC707 board houses a Virtex7 FPGA (XC7VX485T), and ZC706 board has a Zynq FPGA (XC7Z045) which contains a hardened ARM processor; both the devices were fabricated with a 28nm process technology. The hardware energy consumption values reported are based on the worst-case estimates obtained from the Vivado Design Suite. It includes both the static and dynamic energy consumed in all the components implemented on the FPGA, such as the kernels in the application, the (soft-core) processor, local memory and the DRAM controller.

Evaluation on a CPU

To measure the performance on the CPU, we manually implemented each application in C++ and used OpenMP API [Dagum and Menon, 1998] for multi-threaded execution. The execution time and energy consumption were then measured by running these applications on an Intel Sandy Bridge Core i7-2620M laptop CPU running at 2.7GHz and fabricated with a 32nm process technology. The energy consumption was measured using LIKWID performance tool [Treibig et al., 2010]. This tool uses Intel's *Running Average Power Limit* (RAPL) energy consumption counters to measure the energy expended in the CPU package (which also includes the on-chip DRAM controller) for executing each application program.

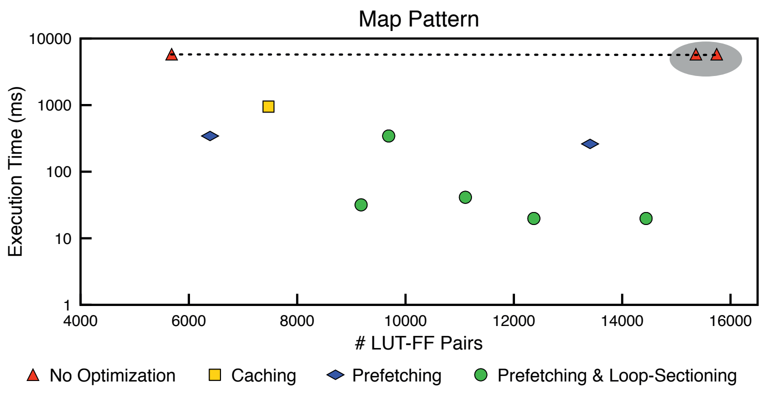
4.5.2 Evaluating with Microbenchmarks

Evaluating the benefits of premeditated optimization parallel patterns

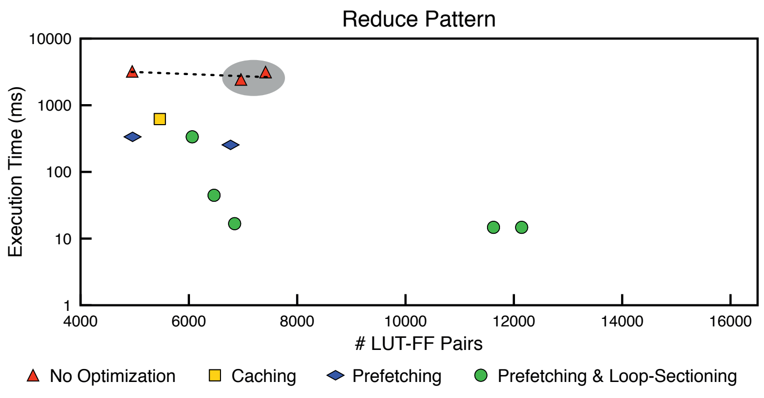
To understand the benefit of the proposed optimizations, we consider three patterns (`map`, `reduce` and `foreach`) separately and study how their performance and resource usage are affected by these optimizations[2]. For each case, we additionally consider the different alternatives that can be generated by applying generic optimizations, such as loop-unrolling and loop-pipelining, to see how this would impact the results. Since the proposed optimizations only target sequentially accessed data structures, we use an array of 10 million integers which is accessed sequentially from each of these patterns. The VC707 development board was used for this evaluation. In the case of `map` and `foreach` patterns, we increment the elements in this array by a constant value and store them back into the memory[3]. In the case of the reduce pattern, we add

[2]We do not consider the `zipWith` pattern separately because it is also implemented in hardware by extending the `map` pattern.

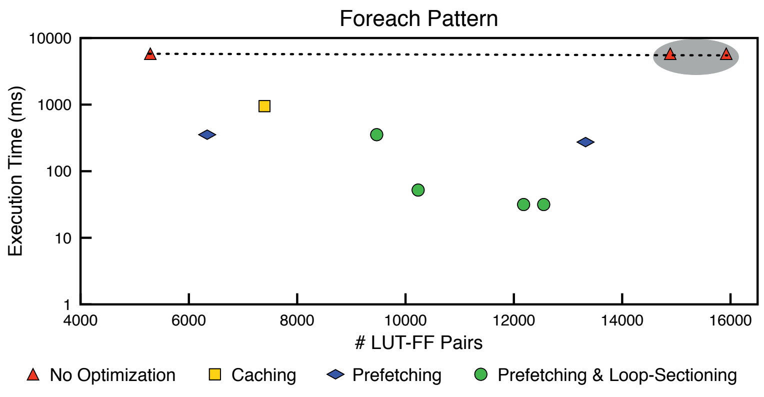
[3]The difference between these patterns this is that `map` creates a new array as output while `foreach` overwrites an already existing one.



(a) Map Pattern



(b) Reduce Pattern



(c) Foreach Pattern

Figure 4.9: Performance-area trade-off between the different variants. The proposed optimizations progressively improve the performance of the generated kernels, but they also consume additional resources. In all cases, adding the local buffer and applying loop-sectioning produces the highest performance and the unoptimized variant uses the lowest area. The data-points in the gray region reveal that using general-purpose optimizations (loop-unrolling, loop-pipelining) alone does not yield substantial improvement.

up all the elements in the array to produce a single value which is then stored into the memory. Figure 4.9 reports the results from this evaluation.

In all three cases, the hardware generated without applying any optimization needs the least resources, but also has lowest performance. However, the more important finding is that the data-points in the gray regions of the figure show that using generic optimizations like loop-unrolling and loop-pipelining on this unoptimized version has little effect on the performance. This is because applying these optimizations creates parallel processing resources, but without additional information the HLS tool performs each memory accesses sequentially providing no performance improvement in the `map` and the `foreach` and only minor improvement in the `reduce`. This clearly demonstrates that blindly applying generic optimizations without considering the hardware-level details may not produce better results. The results further show that adding a local cache to each sequentially accessed data structure and using burst transfers helps to improve the performance significantly. But, using a local buffer along with a buffer manager, instead of the cache, can achieve even better performance with lesser resources. The performance improves from avoiding the cache lookup overhead on each access, and the resource usage is lowered because the buffer manager is simpler to implement compared to the cache control logic; this is because the memory transfers in the buffer manager are statically determined. The highest performance in all three cases is obtained when the local buffer and buffer manager are used in conjunction with the loop-sectioning optimization, which achieves between 180× to 290× speedup over the unoptimized case. The is because loop-sectioning creates blocks with fixed amounts of parallelism which the HLS tools can leverage to obtain higher performance.

4.5.3 Evaluating with Application Benchmarks

Applications used for benchmarking

We performed the full system evaluations using the following four OptiML applications that utilize the features we currently support for hardware generation.

1. *Nearest Neighbor* application finds from a set of data-points the one that is closest to a given point.
2. *Outlier Counter* application uses the criterion proposed by Knorr

Table 4.1: Datatype, computation patterns and hardware modules in each application

Application	**Datatype**	**Map/ Zipwith**[3]	**Reduce**	**Foreach**	**Hardware Kernels**
Nearest Neighbor	integer	3	1	0	2
1-D Correlation	float	8	5	0	4
Outlier Counter	integer	4	2	0	2
1-D Normalization	integer	1	2	1	3

Table 4.2: Percentage utilization of resource in each application

Apps.	**Soft-Core Processor (%)**			**Unoptimized HLS (%)**			**Optimized HLS (%)**		
	LUTs+FFs	BRAMs	DSPs	LUTs+FFs	BRAMs	DSPs	LUTs+FFs	BRAMs	DSPs
Nearest Neighbor	7.01	0.18	4.76	11.27	0.61	4.76	24.62	23.11	6.50
1-D Correlation	7.01	0.18	4.76	15.69	0.96	4.76	58.99	17.89	11.21
Outlier Counter	7.01	0.18	4.76	12.15	0.82	4.76	26.37	29.96	6.50
1-D Normalization	7.01	0.18	4.76	12.53	0.29	4.76	47.59	10.68	7.28

and Ng [Knorr and Ng, 1997] to count outliers in a given data-set.

3. *1-D Correlation* computes the cross-correlation between two large data-sets.
4. *1-D Normalization* applies a linear transforms to a given data-set to fit it within a user provided upper and lower bounds.

Table 4.1 lists the datatype, the parallel patterns and number of separate hardware kernels in each of these applications. The number of hardware kernels is always lower than the number of computational patterns since Delite's built-in fusion optimization, as we saw in the normalization application, can sometimes fuse multiple independent patterns into a single more complex kernel.

Comparing Different the Hardware Designs

Evaluating the benefit of premeditated optimizations on complete applications

To illustrate the range of solutions we can generate, for each application we generate two specific hardware implementations: one that uses hardware kernels without any of our proposed optimizations and the other uses hardware kernels that benefit from optimization discussed in Section 4.4 to achieve the highest performance. Among these, the former design represents the performance one can expect by naïvely using the HLS tool. However, as revealed in Figure 4.9, the performance

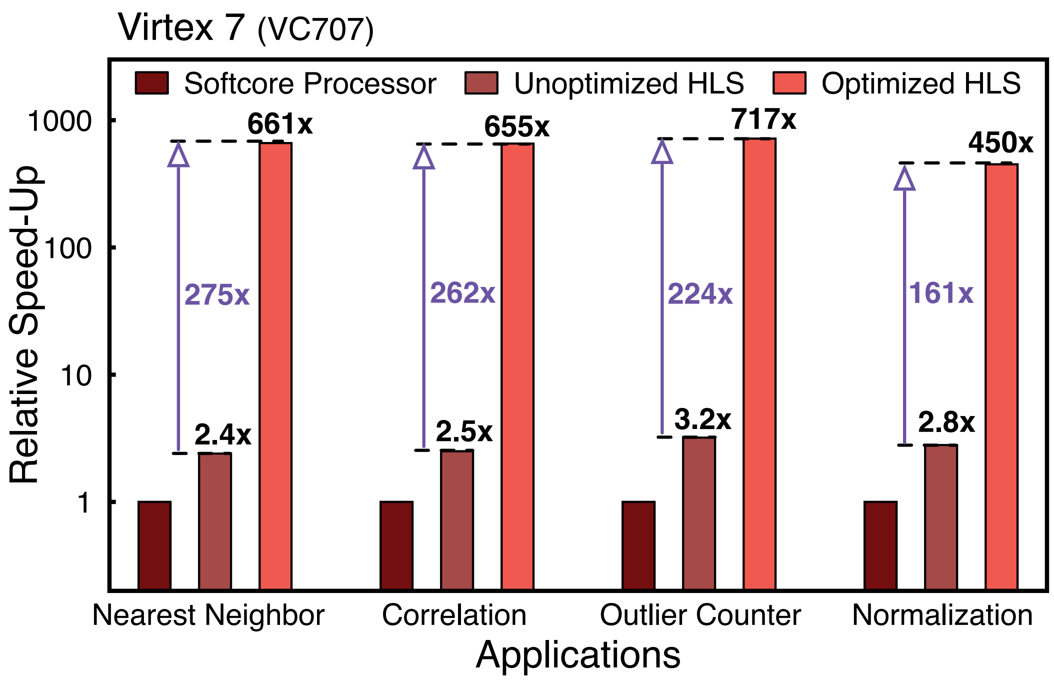


Figure 4.10: Performance comparison among hardware implementations on VC707. The execution performance on the soft-core processor was used a baseline to calculate the speed-up of the other implementations. The design using hardware modules without premeditated optimizations achieve only between 2.4× to 3.2× speed-up. With premeditated optimizations, the performance of the hardware modules improve and the overall speed-up is between 450× to 717× that of the soft-core processor. However, the most important observation is that our premeditated optimizations can achieve between 161× to 275× performance improvement for these applications.

of hardware kernels without premeditated optimizations remains relatively constant despite applying generic optimizations; therefore, we always use the kernel-variants that require the least resources in this design. The added benefit of this is that we are now comparing among two Pareto-optimal design points that can be generated using our tool flow; therefore, we get an impression range of solutions with different performance and resource requirements our tool flow can produce. To generate the hardware kernels in the implementation achieving highest performance, the tool performed optimizations based on the properties of the patterns, explored the design space and used the ILP solver to select the variants that maximized the overall application performance while targeting an FPGA utilization factor of 80% on each target. To serve as a reference, we executed each application using only the (soft-

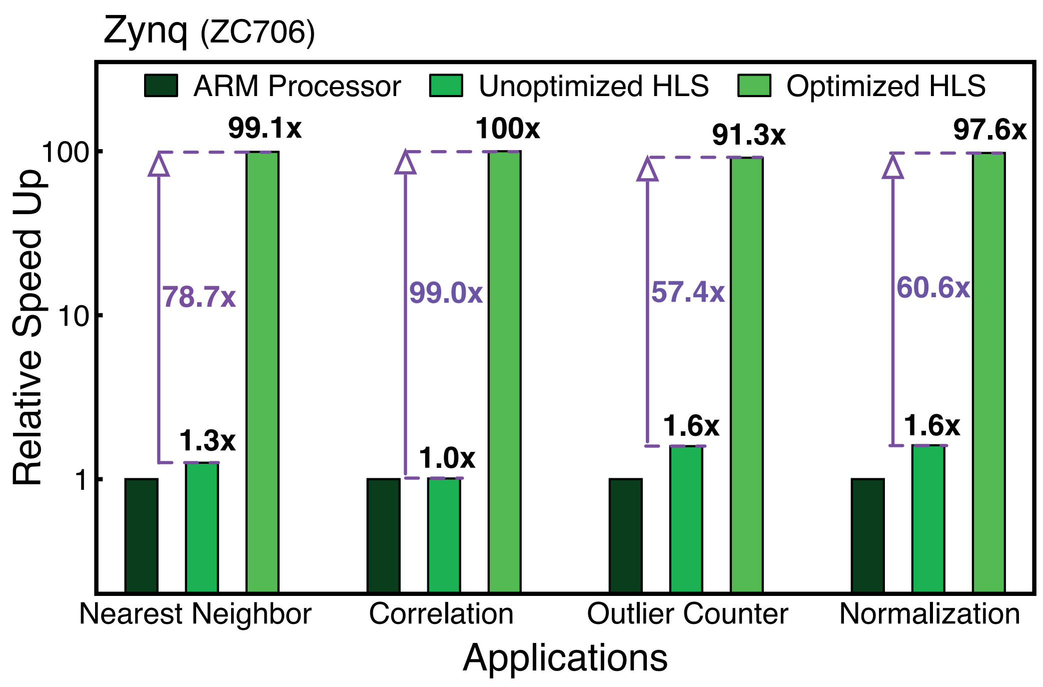


Figure 4.11: Performance comparison among hardware implementations on ZC706. The execution performance on the ARM processor was used a baseline to calculate the performance gains of other implementations. The design using hardware modules without premeditated optimizations achieve only between 1.0× to 1.6× improvement. But, with premeditated optimizations, the performance of the hardware modules improve and the overall speed-up is between 91× to 100× that of the ARM processor. The most important result is that our premeditated optimizations can achieve between 57× to 99× performance improvement for these applications.

core) processor without any optimizations (i.e., using the `-O0` flag); this was done to ensure that the acceleration potential shown in the figure are not skewed due to compiler optimizations.

Comparing implementations on a VC707

Figure 4.10 compares the performance of the implementations on the VC707 development board and Table 4.2 lists the resources utilized for each one. From the figure, the designs without premeditated optimizations is between 2.4 to 3.2 times faster than the processor while utilizing slightly less than twice the resources. However, the high-performance implementation achieves between 450 to 717 times the soft-core processor's performance, albeit by using more resources, and clearly illustrates the benefit our approach. Comparing across applications, the highest performance gain was for the outlier counter application where the execution time was dominated by a kernel containing nested patterns.

Due to the significant performance benefit obtained for this kernel, the overall execution time of the application was improved. The nearest neighbor also has a similar, albeit simpler, kernel, but this kernel had a lesser influence on the overall execution time of the application; hence, the performance gain for this application was limited. However, comparing among the hardware implementations for the same application, the simpler kernels of the nearest neighbor enabled the tool to use the highest performing kernel variants while implementing the optimized design to deliver the highest improvement. The lowest performance improvement was obtained for the normalization application we described in Section 4.3.3. This is because the normalization application contains a `foreach` pattern and, as seen in Figure 4.9, the relative performance improvement for the `foreach` is the lowest among the patterns.

Comparing implementations on a ZC706

Considering the ZC706 development board, the hardened ARM processor enabled the software execution performance to improve considerably[4]. As seen in Figure 4.11, the designs without premeditated optimizations only achieve between 1.0 to 1.6 times the performance of the processor while those one with premeditated optimizations are between 91 to 100 times better. This again shows that the premeditated optimizations can significantly improve the performance of the hardware implementations. The performance gains for the optimized designs were restricted on the Zynq due to the limited data bandwidth available from the programmable logic part of the chip, where the hardware designs are implemented, and the hardened memory controller that is collocated with the ARM processor. To verify if the performance improvements were constrained due to data bandwidth, we implemented each application on the VC707 board by using the same hardware modules that were used in the optimized Zynq designs and measured the performance. For every application, the execution performance on VC707 board was higher than that on the Zynq, ranging between 1.4$\times$ to 3.8$\times$, despite using the same hardware modules; this clearly illustrates how performance is limited due to data bandwidth on Zynq board. Comparing the performance gains of unoptimized hardware design on the ZC706 with the VC707, the gains for the correlation application has dropped because of the superior performance of the hardened floating point units on the ARM, and improvement for normalization

[4]The software running on the processor was compiled using the `-O0` flag.

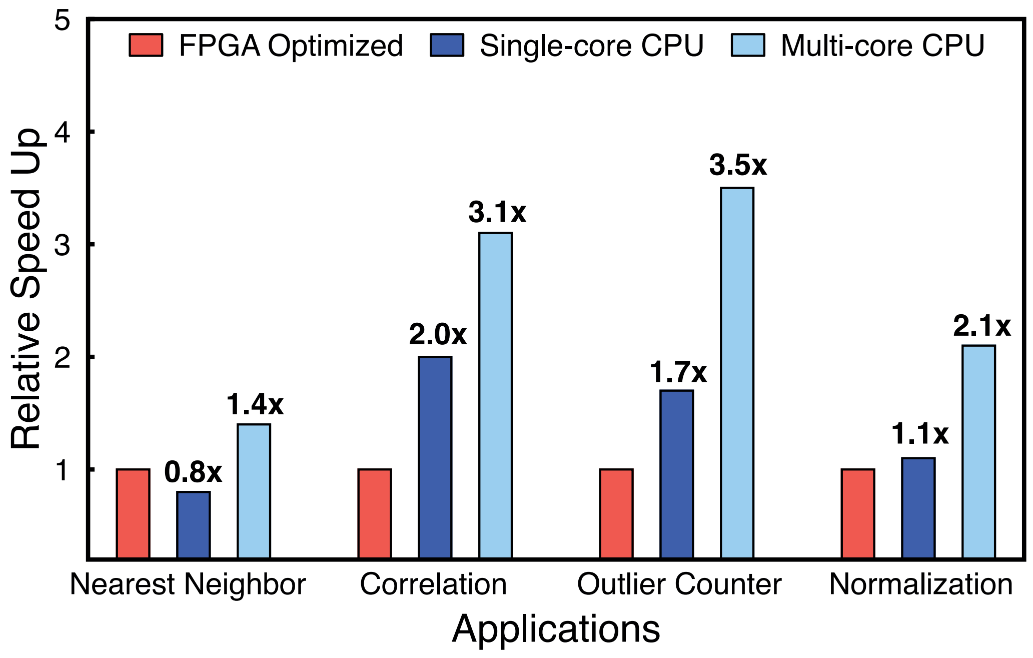


Figure 4.12: Performance comparison between FPGA and CPU. The CPU achieve better execution performance compared the FPGA. However, for some applications, such as Nearest Neighbor and Normalization, the FPGA is able to reach the performance on a single-core implementation. Between the CPU implementations, as expected, the multi-core implementations outperforms the single-core ones.

has increased because the ARM did not have a division unit while the soft-processor has one.

These results show that our methodology can easily generate a variety of different solutions from a high-level DSL program that achieve different trade-offs between performance and resource usage. More importantly, it demonstrates that by decomposing the application into a set of well understood computation patterns, we can automatically generate high-performance hardware systems.

Comparing the Hardware Design with a Processor

Comparing the FPGA design with running the application on a CPU

To provide more insight into performance and energy efficiency of the generated solutions, we compare the highest performing FPGA designs on the VC707 with a laptop CPU (Intel Core i7-2620M). The results on Figures 4.12 and 4.13 shows that when we use the highest-performing

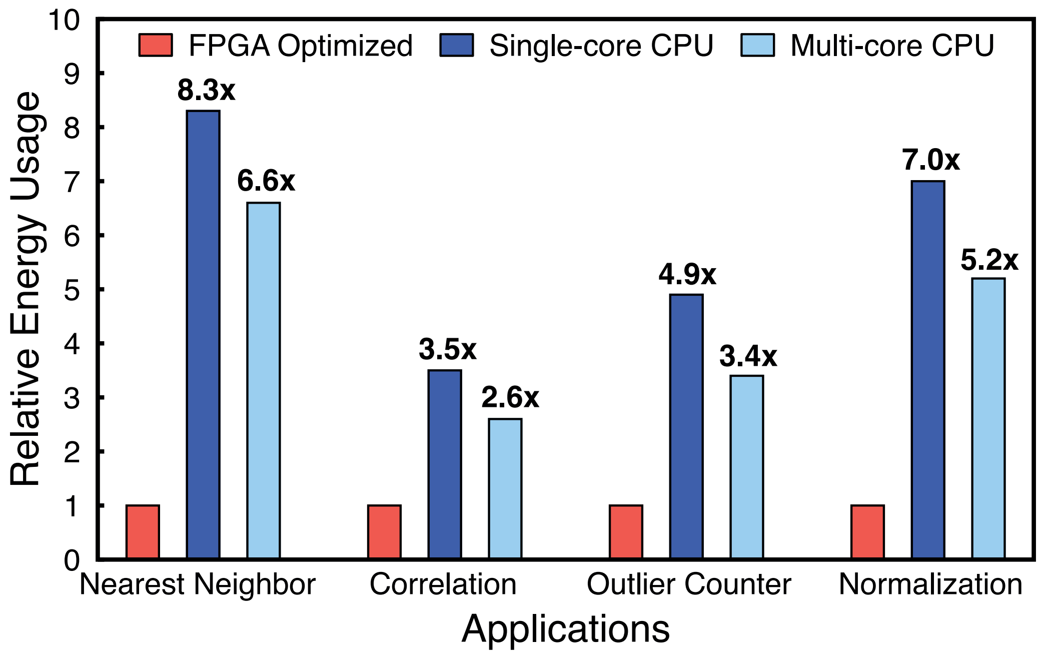


Figure 4.13: Energy-efficiency comparison between FPGA and CPU. For applications, the FPGA implementations are more energy-efficient compared to the best CPU implementations, by a factor ranging from 2.5 to 6.6. Among the CPU implementations, the multi-core implementations are more energy-efficient compared to the single-core ones.

variants in the FPGA design, such as in the case of nearest neighbor and normalization, FPGA's execution performance is quite close to the single-threaded execution on the laptop CPU. This performance gap increases in the outlier counter and correlation applications because of using lower performing variants in the hardware design. In terms of energy efficiency, however, the FPGA clearly outperforms the CPU for all applications. The lowest improvement was observed for correlation application where the ASIC implementation of floating-point units on the CPU gives it a definite edge over the FPGA. The figure also shows that multi-threaded execution can improve the performance and energy efficiency of the CPU, but the FPGA is still the more energy-efficient platform.

Understanding the performance difference

To understand the cause of the performance difference, we need to take a closer look at the applications we used for the evaluation. All these applications have a low computational complexity and process large amounts of data that is stored in the external DRAM. Therefore, its execution performance on any platform depends greatly on the platform's effective memory bandwidth. The maximum memory bandwidth available to the CPU is 21.3GB/s, compared to the theoretical maximum of 6.4GB/s available to the hardware design. This along with the highly tuned memory architecture of the CPU, enables it to have a higher effective bandwidth and, thereby, achieve better performance. Moreover, in the hardware designs, the DRAM controller accounts for between 50% to 70% of the total energy consumption in the system. On the CPU, this controller is implemented using as an ASIC and, therefore, is much more energy-efficient. This suggests that implementing this DRAM controller as an hard IP will enable FPGAs to achieve a better performance and a much higher energy efficiency for such applications.

4.6 Discussion

Key insights

Domain-specific synthesis tools can significantly reduce the entry barrier for a user without hardware design knowledge, i.e, domain experts and users with a software development background, to using FPGAs. The approach presented in this chapter enables the automatic generation of high-performance FPGA designs from high-level DSL specifications. This DSL specification is intuitive to express for application developers and also shields them from the hardware details. By decomposing the DSL operations into well understood computational patterns, i.e., `map`, `zipWith`, `reduce` and `foreach`, we enable the efficient hardware generation for many DSL operations using a small set of computational patterns. Our results indicate that the proposed optimizations are effective and that our approach can produce complete hardware designs to target FPGAs. We also demonstrate that these designs can offer a better energy efficiency compared to a laptop CPU. Such a tool flow that generates hardware systems from computational patterns can be easily reused by other domain-specific tools where the DSL-operation can be mapped to the same set of patterns; for instance, other DSLs build into Delite will be able to directly utilize the tool flow described in this chapter. Moreover, the tool flow itself can be extended by adding more patterns to support hardware generation for even more domains. The task of mapping new DSL operations into existing patterns will not require hardware design knowledge because computational patterns are essentially algorithmic methods that can be easily understood by domain-experts and software developers. Thereby, such a tool flow can become a shared infrastructure that significantly reduces the effort needed to develop new domain-specific tool flows.

What is next? The hardware designs we generated in this chapter, use only single hardware modules per parallel operation and exploit the parallelism within that module to achieve high performance. However, such single module designs did not fully utilize the resource available on the FPGA as well as the data access bandwidth on the interconnect bus. Additionally, the performance of a single module can sometimes be quite low when there are non-sequential data accesses from the modules where the long data access latency will cause data starvation and limit performance. To overcome these problems, we need to parallelize computations across multiple modules. We will investigate the possibility of extending this hardware synthesis flow to automatically generate designs that parallelize computations across multiple modules in the next chapter. Additionally, to keep the flow easy to use for the end-user, we want to generate these multi-module designs from the same high-level functional specifications. We explore the potential for this extension in Chapter 5.

5 Leveraging Multi-Module Parallelism with Computational Patterns

FPGAs can offer great performance for parallel applications by performing computation in a spatially parallel manner. In order to fully benefit from this spatial parallelism, the designer must carefully consider how the application is mapped on the device and how its computational throughput is matched with the available data access bandwidth. This often requires a detailed analysis of the application and can be quite tedious, even for hardware design experts. To improve productivity of designers and to enable people without hardware design expertise to fully benefit from FPGAs, we need to develop tools and automated methodologies to correctly parallelize applications on an FPGA. The work explored in Chapter 4 only exploited parallelism within a hardware module generated by the HLS tool. However, the effective parallelism in the computation within one module can depend heavily on the data access bandwidth to the module, which in turn can depend on the module's data access patterns. Data access patterns are particularly important when the data is accessed from an external memory with long access latencies. Moreover, the overall bandwidth utilization by a single HLS generated module can be much smaller than what is afforded by the system bus [Zhang et al., 2015]. Exploiting parallelism across multiple modules is one way to alleviate these problems.

5.1 Motivation

Matrix multiplication using computational patterns

Using the approach developed in Chapter 4, we can easily write a program to compute $C = A \times B$, where A, B and C are floating point matrices; Figure 5.1 shows the part of this program that computes C. This

```
// A and B are input matrices and this code computes C as A*B
// rows_A is the number of rows in matrix A and cols_B number
   of columns in B
val C = (0::rows_A, 0::cols_B){ (i,j) =>
        // Compute the elements of C a dot-product between rows
              of A with columns of B
     (A.getRow(i) * B.getCol(j)).sum
  }
```

Figure 5.1: Matrix multiplication in OptiML. This high-level program provides only the functional specifications and contains no hardware level details making it easy to specify for application developers. Here, matrices `A` and `B` are floating point matrices that are defined elsewhere and this program computes the elements of matrix `C` by computing the dot-products between the rows of `A` with the columns of `B`.

program will be decomposed by our high-level compiler into a set of nested computational patterns as shown in Figure 5.2. Here, each cell in *C* is the dot-product of a row of *A* and a column of *B*; this computation can be composed using a `zipWith` pattern, which performs the element-wise multiplication between a row of *A* and a column of *B*, followed by a `reduce` pattern that adds the results from the multiplication. To complete the matrix multiplication, the `zipWith` and `reduce` patterns are nested inside two `map` patterns, one iterating along the columns of *B* and the other along the rows of *A*[1]. These patterns reveal the parallelism in the computation (e.g., the separate multiplications in `zipWith` can be parallelized) and expose how the data is consumed and produced in the process (i.e., *A* is read row-wise and *B* is read column-wise and used to produce a completely new matrix *C*). By leveraging these properties, a purely functional application description can be automatically translated into a well structured and annotated program that will produce a good quality hardware module using current HLS tools.

Data starvation limits computational throughput

Although the computational patterns make it possible to generate a highly parallel hardware module, the execution performance still depends on the data access pattern, dependency between accesses and even latency of the operations. For instance, in the matrix multiplication, if matrices *A* and *B* are stored in row-major order, the columns of *B*

[1]In the code, two `map` patterns actually iterate through all the elements of output matrix. The value is each element is computed, by the nested `zipWith` and `reduce` patterns, as the dot-product between the corresponding row *A* and column of *B*

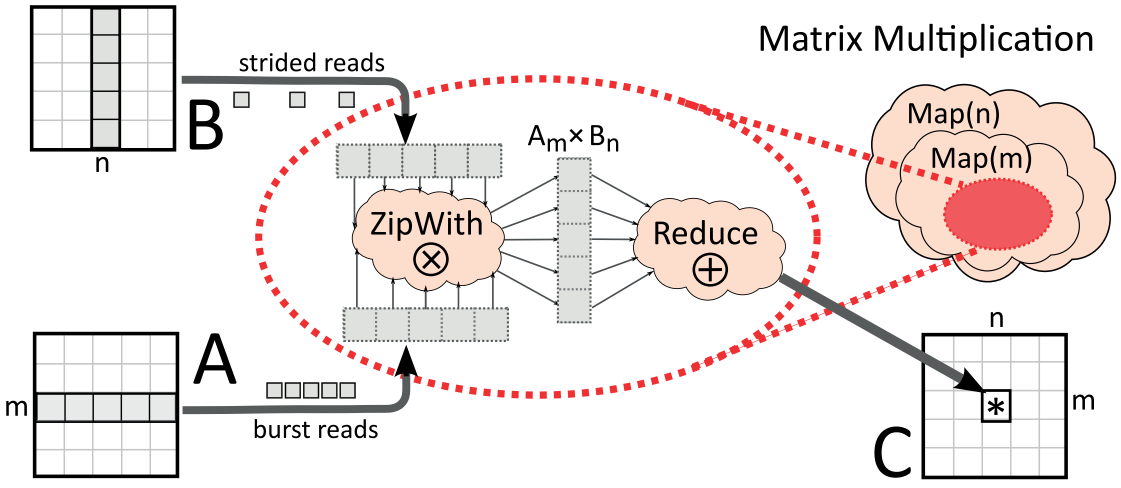


Figure 5.2: Each result cell of the matrix multiplication is computed using two parallel computational patterns: `zipWith` and `reduce`. Here, the `zipWith` performs element-wise multiplication of the rows of matrix A with the columns of matrix B. The `reduce` sums all the results from the `zipWith` to compute the cell value.

need separate strided accesses that will underutilize the bus-bandwidth and significantly diminish the performance of the module[2]. So, despite the parallelism available in the computation, performance of the module will remain fixed due to data starvation. Therefore, while the pattern-based approach can vary the amount of parallelism (e.g., loop unrolling) exploited to generate multiple implementations of the hardware module (i.e., variants), the performance does not improve; Figure 5.3 (single module design) provides experimental evidence of this effect of data starvation where further parallelization consumes additional resources but does not deliver better performance.

Balance computational throughput with communication bandwidth with multiple modules

In such counterintuitive situations, we may still improve performance by parallelizing computations across multiple hardware modules, similar to parallelizing a computation across multiple CPUs, and improve the aggregate throughput. Since parallelizing computations results in the need for synchronization, to achieve this on FPGA, the designer must identify synchronization requirements in the application and build custom synchronization schemes. Sometimes, there are even less obvious needs for synchronization, such as false sharing [Bolosky and Scott, 1993] due to mismatches between the widths of the system bus

[2]We are using a simplistic implementation of matrix multiplication here to illustrate the idea. An optimized implementation is also considered later.

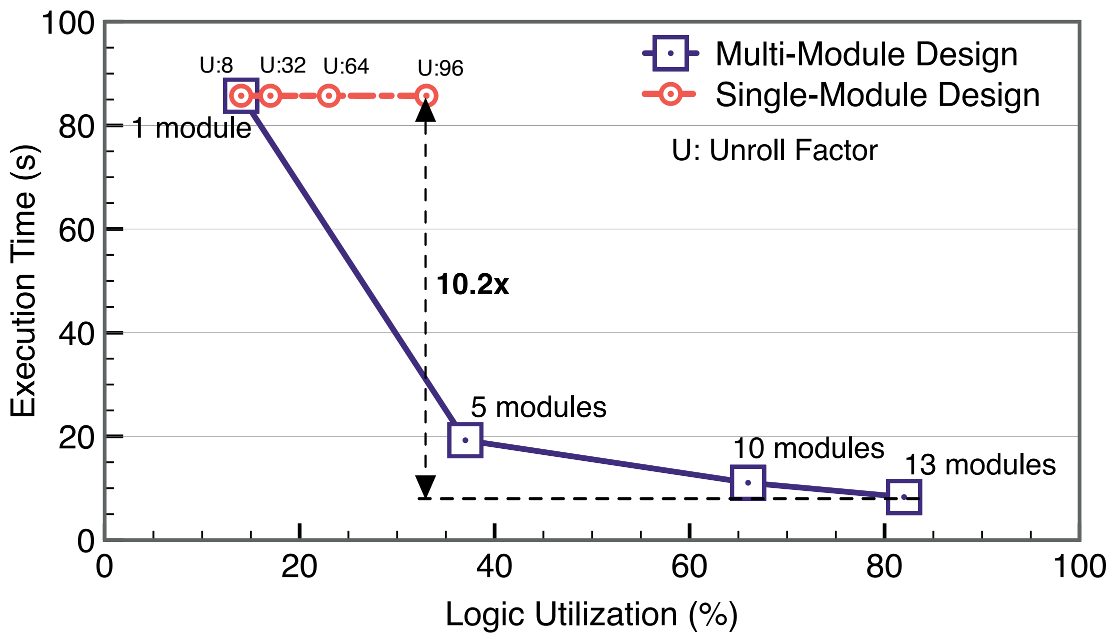


Figure 5.3: The design with only a single module performs poorly due to data starvation. Further parallelization will consume additional resources, but not deliver better performance. The design with multiple modules is able achieve higher aggregate bandwidth and, therefore, better performance.

and the datatype in the computation. Moreover, to efficiently parallelize computation across modules, there is also a need for good work partitioning schemes. All these issues make the task of manually parallelizing the computation tedious, error prone and, above all, hard to accomplish without hardware design expertise.

Overview of the approach

This work presents a new approach to automatically parallelize applications that are composed using computational patterns. The approach includes, (1) automatic extraction of synchronization requirements in the kernel when parallelized across multiple hardware modules, (2) design space exploration using *Integer Linear Programming* (ILP) to find the set of module variants that obtain the best performance for a given application, and (3) automatic generation of the complete system using the selected module variants along with all the essential synchronization hardware. We perform this automation by leveraging the properties of the computational patterns, specifically how data is consumed and produced, and properties of the system architecture, such as the data-allocation strategy and width of the system-bus. By employing this approach on the matrix multiplication kernel, we are able to exploit the parallelism in the outer-most level `map` pattern to parallelize the computation across multiple modules and achieve better performance, as shown in Figure 5.3.

Outline

In the rest of this chapter, Section 5.2 takes a look at the modifications that will be needed to the hardware generation flow from Chapter 4 to automatically generate systems where operations are parallelized across multiple modules. Section 5.3 explains the properties of the computational patterns, and Section 5.4 discusses how our methodology uses these properties to parallelize kernels composed from patterns across multiple hardware modules. We demonstrate the benefit of the approach in Section 5.5 by presenting the performance improvements we achieve on seven different applications. Finally, Section 5.6 reviews the key insights from this work.

5.2 Modifications to the Hardware Generation Flow

The toolchain developed in Chapter 4 needs to be modified to support multi-module hardware generation. In this section, we will first review the overall flow and then discuss where modifications are needed.

Reviewing the hardware generation flow

Figure 5.4 illustrates, in a more simplified form, how the hardware generation flow in Chapter 4 was implemented. In this flow, application programs are first compiled by a high-level compiler into parallel and serial kernels. The parallel kernels contain one or more parallel computational patterns, such as `map`, `reduce`, `zipWith` and `foreach`, that have well understood properties, such as the nature in which it produces or consumes data or the parallelism in its operation; matrix multiplication is an example of such a parallel kernel. The toolchain leverages these properties to automatically infer the suitable optimizations for each patterns in the kernel and then utilizes an HLS tool, in our case Vivado HLS [Xilinx, 2013a], to generate a highly parallel hardware module. Additionally, it can vary the amount of parallelism exploited in the generated hardware to create multiple implementations, which are called *variants*, for each kernel. After the variants for the parallel kernels in the application are generated, one variant is selected per kernel and then connected within a system architecture template using wide, high-bandwidth buses to complete the system design. This template provides shared memories, clock and control circuitry, and a (soft-core) processor. Figure 5.5 shows the system design for the matrix multiplication

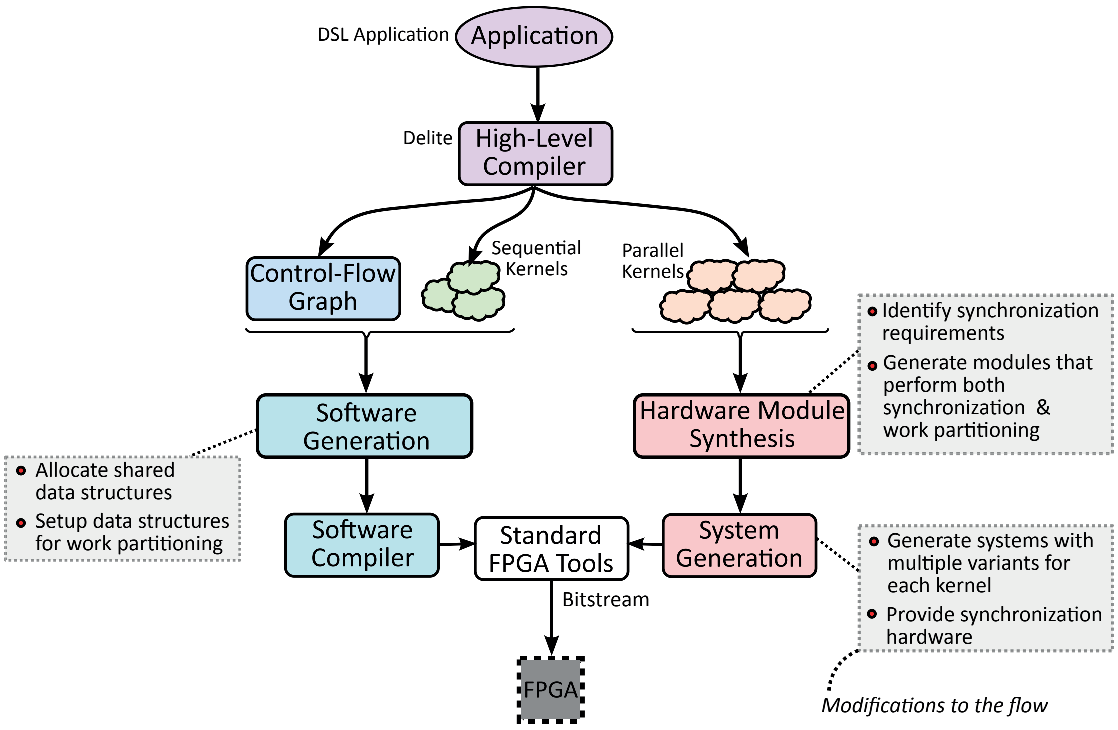

DSL Application
Application
Delite
High-Level Compiler
Control-Flow Graph
Sequential Kernels
Parallel Kernels
Identify synchronization requirements
Generate modules that perform both synchronization & work partitioning
Software Generation
Hardware Module Synthesis
Allocate shared data structures
Setup data structures for work partitioning
Software Compiler
Standard FPGA Tools
System Generation
Generate systems with multiple variants for each kernel
Provide synchronization hardware
Bitstream
FPGA
Modifications to the flow

Figure 5.4: The Delite compiler decomposes the high-level application into parallel and sequential patterns, and an associated control-flow graph. The parallel patterns undergo hardware synthesis to generate hardware modules and become part of the hardware design. The sequential patterns and the schedule generated from the control-flow graph are generated as a software program that runs on a (soft-core) processor in the system.

application. The (soft-core) processor in this system design is used to execute the sequential kernels in the application and for orchestrating the execution of the hardware modules. Our toolchain automatically generates software for this processor from the sequential kernels and the application's control-flow information.

Modifying the hardware generation flow for multi-module parallelization

The hardware system generated in Chapter 4 only used a single hardware module for each parallel kernel in the application. In order to generate systems where one or more of these kernels are parallelized across multiple hardware modules (i.e., multi-module systems), this hardware generation flow needs to be modified as indicated in Figure 5.4. When the computation is parallelized across multiple modules, these modules will need to share data structures. Therefore, to avoid data races, these modules need to perform synchronization and guard accesses to these shared data structures. To achieve this automatically,

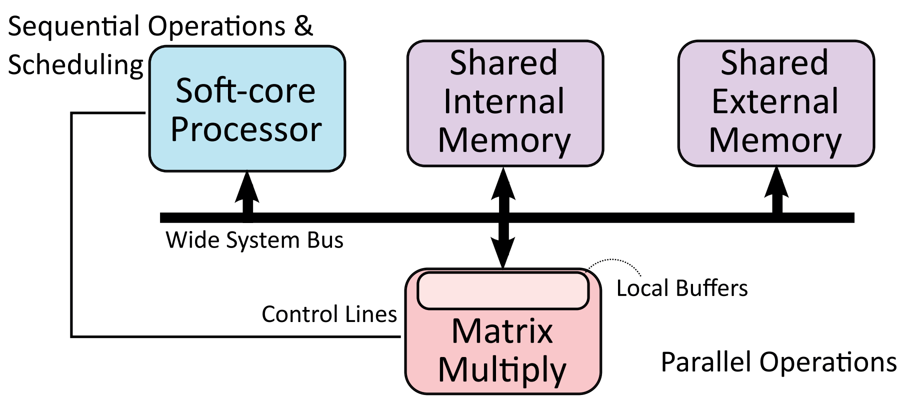


Figure 5.5: The hardware system generated for the matrix multiplication has the hardware module implementing the multiplication and a (soft-core) processor for the scheduling the hardware module and executing the sequential patterns, if any. The data for the computation is stored in the shared memory that is accessed over the wide system bus. Each hardware kernel also has a local memory used for buffering input data and holding intermediate results.

during hardware generation, we need to analyze the accesses to data structures and selectively modify some accesses to use appropriate synchronization schemes. During the hardware system generation, to facilitate synchronization between modules, additional synchronization hardware, e.g., hardware mutexes, needs to be included in the system. Since these systems can now use multiple variants for each parallel kernel, the system generation process must decide the exact number and types of variants for each kernel that would maximize the application performance. To optimally leverage these multiple modules, we need to develop workload partitioning schemes and integrate them into the generated hardware modules. Moreover, to support this workload partitioning scheme, the software generation flow must also be modified to initialize additional data structures that are used in the scheme before starting the hardware modules. Furthermore, some tasks that were previously performed by the hardware modules, such as allocating shared data structures and updating their metadata, will now be done in software; this change is required because these tasks need to be done only once during the execution of the hardware module and we do not want to replicate these tasks across multiple modules.

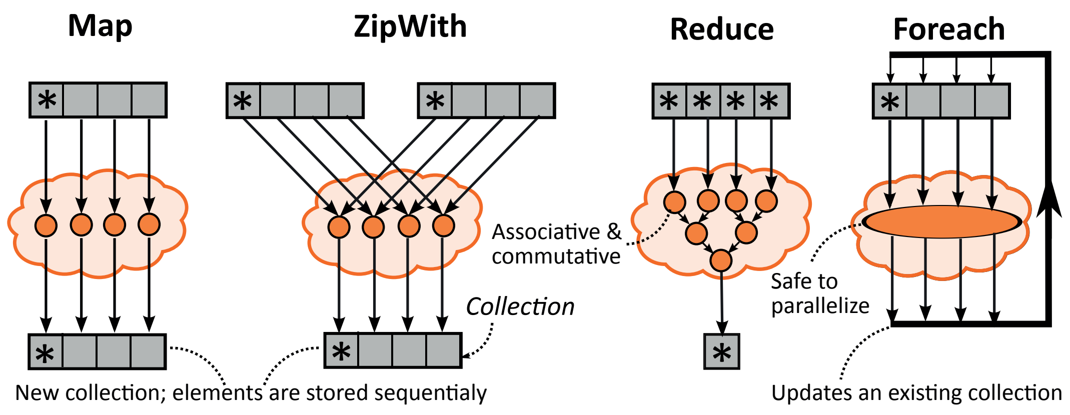

Map
ZipWith
Reduce
Foreach
Associative & commutative
Collection
Safe to parallelize
New collection; elements are stored sequentialy
Updates an existing collection

Figure 5.6: In the `map` and `zipWith` patterns, the output collection is stored in sequential locations. The `reduce` uses a binary operation that is associative and commutative and, therefore, the operations can be reordered. And, the `foreach` can update the values of an existing collection in any order but the operations can be parallelized.

5.3 Data Access Properties of Parallel Patterns

Data Access Properties of Parallel Computational Patterns

As we saw in Chapter 4, the parallel kernels extracted by the Delite compiler contain one or more computational patterns, such as `map`, `zipWith`, `reduce` and `foreach`. These patterns, as described in Section 4.3.2, provide certain guarantees regarding the nature of their computation and how they produce or consume data. As explained before, the `map` and `zipWith` patterns always use a pure (side-effect free) function to create a fresh collection. Additionally, as illustrated in Figure 5.6, the result values in both `map` and `zipWith` are stored in sequential locations with the result from the i^{th} computation being stored to the i^{th} location. The `reduce` pattern computes a single element by applying a binary function that is both associative and commutative to the elements in a collection. We levearge this property to reorder the elemental operations in `reduce` and still produce the correct result. The `foreach` pattern is typically used to modify values in an existing collection by applying an impure function (with side-effects) to each element, such as to change all the negative numbers in an existing array and set them to zero. Unlike the `map` and `zipWith`, `foreach` can update the values of the result collection in any order. However, we restrict the programming model to ensure that the operations in the `foreach` can be executed in parallel without problems[3].

[3] The user has the responsibility to ensure that performing the operations in `foreach` in parallel will not produce incorrect results.

These computational patterns make it easy to identify the parallelism in the application and expose it to the HLS tool. But, as illustrated in Section 5.1 with the matrix multiplication, the performance of the resulting hardware module still depends on aspects such as the data access pattern, interdependencies between accesses, and latency of the operations. In the next section, we will see how the properties of these patterns can be leveraged to parallelize computations across multiple modules.

5.4 Parallelizing Computation Across Multiple Modules

The key focus of this work is to overcome the problem of low performance from a single hardware module by parallelizing computation across multiple modules. In this section, we discuss this approach and detail how we leverage the properties of the computational patterns and that of the system architecture to automate this process.

5.4.1 Identifying Synchronization Requirements

To parallelize kernels across multiple modules, we identify how each kernel uses and produces data and utilize synchronization schemes to guarantee correct use of shared data structures. Since these kernels are composed of computational patterns, we utilize the properties of these patterns to infer how data is consumed and produced in the kernel. We use this knowledge in conjunction with the properties of the system, such as data allocation strategy and width of system-bus, to correctly identify the synchronization requirements between the modules.

Synchronization Rules for Simple Patterns

Synchronization rules for kernels with single patterns

In the case of kernels with a single pattern, if this is a `reduce`, it operates on a collection to compute a single new result. When parallelized, multiple modules update this result, as shown in Figure 5.7(a); therefore, we need to use a mutex, to avoid data races. If, however, the kernel has a `map`, `zipWith` or a `foreach` pattern, each elemental operation uses distinct elements from the input collection(s) to compute independent elements in the output collection. Hence, one might naïvely

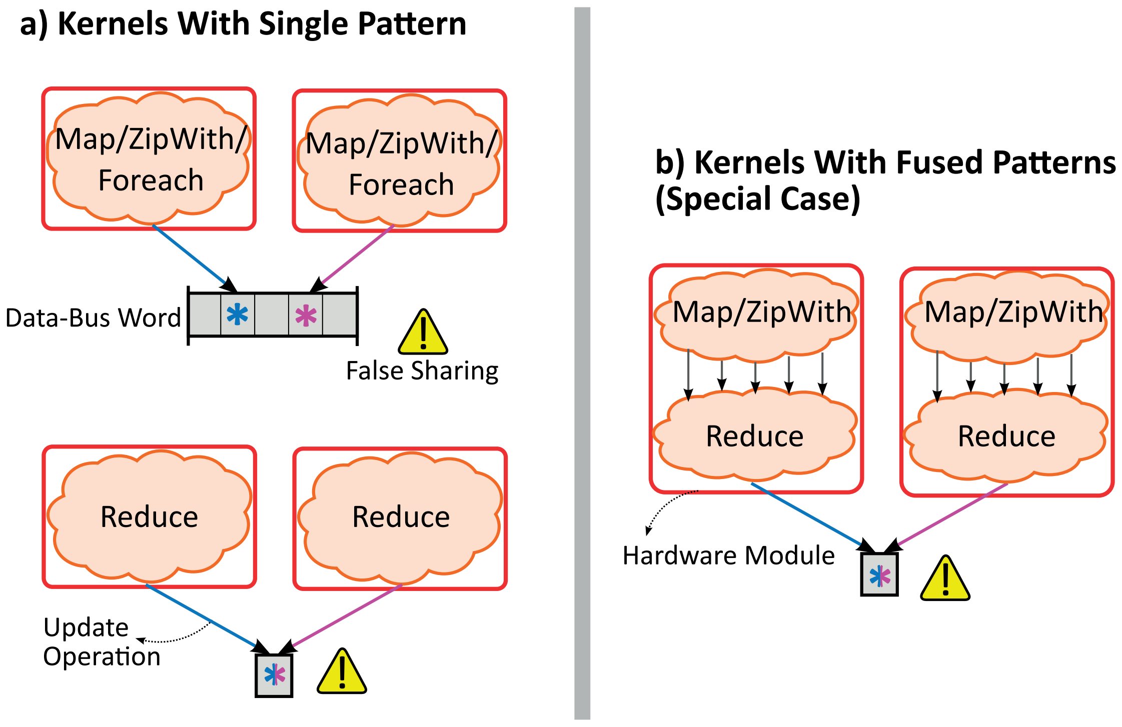


Figure 5.7: Synchronization rules for simple patterns: a) In kernels with a single pattern, we can have have data races either due to shared output variable or false sharing. To prevent these races, we need to use mutexes for all writes. b) When a `map` or `zipWith` is “fused” with a `reduce`, the data from the former is directly consumed by the `reduce`. Therefore, only the output from `reduce` needs to use a mutex.

assume that there is no need for any synchronization. However, this would be incorrect if there is a mismatch between the size of datatype used and the width of system bus; since the latter is typically much larger in order to maximize data bandwidth, this can create false sharing problems [Bolosky and Scott, 1993]. False sharing occurs because each elemental update to the output collection will need to perform a read-modify-write operation which will give rise to data races if multiple modules update the same bus-word simultaneously. This is shown in Figure 5.7(a). Some bus protocols provide data masks to selectively update specific bytes in a bus-word, however, we can have collections of datatypes that are smaller than a byte making such schemes insufficient. To illustrate with an example, if an output from a `foreach` is a collection of boolean values, the modules will need to update single bits in this collection and this is only possible with a read-modify-write operation. If another module is simultaneously updating another bit in

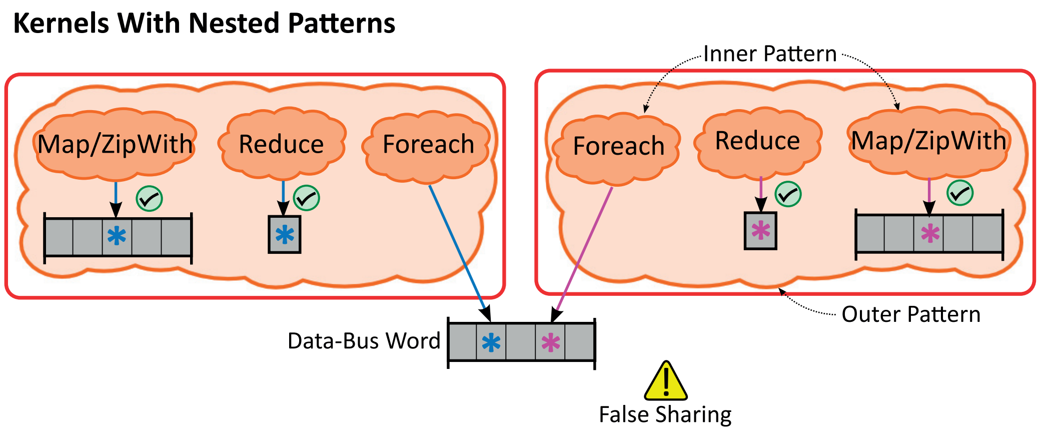


Figure 5.8: Synchronization rules for nested patterns: When computational patterns are nested inside another pattern, the synchronization rules for the pattern at the outermost level is exactly the same as in the simple case. However, for inner level patterns, synchronization is only needed when it is a `foreach` pattern that updates a data structure with global scope.

the same data-bus word, they can inadvertently overwrite each other's results. Therefore, a mutex is necessary to make this read-modify-write operation atomic.

Synchronization Rules for Fused Patterns

Synchronization rules for kernels with fused patterns

While generating parallel kernels, the Delite compiler sometimes *fuses* multiple patterns together so that they can execute in parallel. For instance, as we saw in Section 4.3.3, if we compute the minimum and maximum from the same collection, the two `reduce` patterns might get fused into a single kernel to compute both the minimum and maximum values in parallel. If the fused patterns are completely independent, they each retain the synchronization requirements they had in the simple case. The exception to this is when a `map` or `zipWith` is fused with `reduce` and the latter directly consumes the data from the former. In this case, as shown in Figure 5.7(b), if the output of the `map` or `zipWith` never write into the shared memory, only the output from the `reduce` would need to use a mutex. We see an example of such a fused pattern in the matrix multiplication where during the dot-product computation the `reduce` pattern directly consumes the data from the `zipWith`. However, in that specific case, the fused operation is nested within other patterns and therefore has slightly different rules as we will see next.

Synchronization Rules for Nested Patterns

Synchronization rules for kernels with nested patterns

When computational patterns are nested, such as in matrix multiplication where the dot-product is nested inside `map` patterns, the outermost pattern retains the same synchronization requirements as they would have had in the single pattern case. As shown in Figure 5.8, if the inner-level pattern is either a `map`, `zipWith` or a `reduce`, the new data it produces is visible only within the execution context of a single computation of the outer-level pattern and hence disjoint. Therefore, when this kernel is parallelized across multiple hardware modules, data produced by the inner-level patterns in different modules is completely disjoint (i.e., located at different bus addresses) and does not need synchronization. However, to guarantee this, the data allocation strategy we use ensures that data created by the inner-level patterns from different modules never share the same data-bus word. Thus, in the matrix multiplication, the fused `zipWith` and `reduce` that calculate the dot-product at the inner-level do not need any synchronization. However, when the result matrix is updated by the outer-level `map` pattern, the different modules need to synchronize using a mutex. This is advantageous since it enables the inner-level computation that executes more often to progress in parallel without any synchronization overheads. If, however, the inner-level pattern is a `foreach`, it can update any collection that was allocated before and, therefore, the potential exists for data races between the `foreach` patterns in different modules that write to same collection. As a result, `foreach` patterns, even when they are nested inside other patterns, need to use a mutex for synchronization if they write to a collection that is visible outside the outermost pattern (i.e., a collection with a global scope).

5.4.2 Reducing Synchronization Requirements

Synchronization, while needed for correct execution, serializes operations across the modules and, therefore, diminishes performance benefits of parallelization. Hence, it is better to reduce synchronization requirements as much as possible.

Reducing synchronization requirements for `reduce`

When a kernel with a `reduce` pattern at the outermost level is parallelized across multiple modules, a straightforward optimization is to provide each module with a local data structure to hold partial results. This permits the different modules to operate in parallel and synchro-

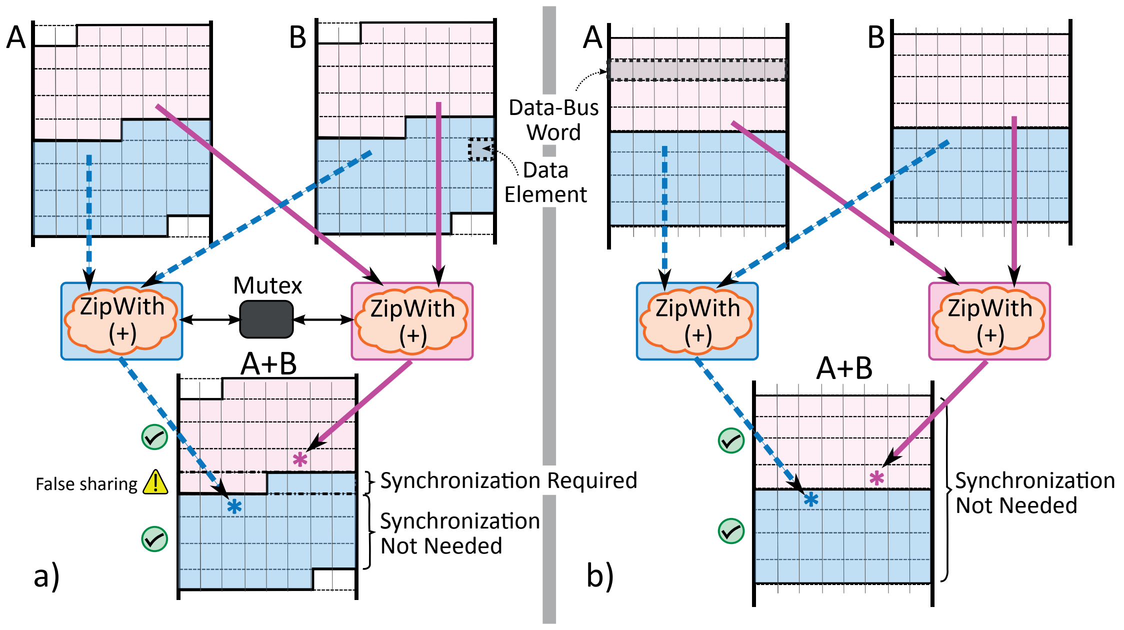


Figure 5.9: Vector addition is used as an example to illustrate the opportunities to reduce synchronization overheads. a) The write from each hardware module updates sequential locations; therefore, the modules can only have false sharing at the edges of their respective sequential ranges. A hardware mutex is used to make updates to these edge elements atomic. b) The access range of each module is aligned to data-bus width and the work unit size is chosen appropriately to eliminate the need to use synchronization hardware.

nization is only needed at the end of the computation when the different partial results are combined and written to the shared data structure. This optimization is possible because the elemental computation in `reduce` is both associative and commutative; the order in which the operations are performed and later combined does not matter.

Reducing synchronization requirements for `map`, `zipWith` and `foreach`

If the parallel operation is a `map` or a `zipWith` pattern, we know that the i^{th} elemental operation will utilize the i^{th} element(s) of the input collection(s) to produce the i^{th} result in the output collection. We utilize this knowledge while partitioning the kernel's computation so that each module is given a contiguous range (from i^{th} to $(i+n)^{th}$) of the computation; this ensures that the same module writes n sequential values in the output collection. Therefore, the writes from different modules can interfere with each other only on the edges of their respective sequential ranges, avoiding synchronization for the non-edge writes. Unlike the `map` and `zipWith`, the `foreach` can update a preallocated collection in any order and is, therefore, not compatible for this optimization. We overcome this problem with additional compiler analysis to identify if the collection written to by the `foreach` is updated sequentially and selectively apply this optimization. Figure 5.9 (a) illustrates the benefit of this optimization by considering the case of vector addition implemented with a `zipWith` pattern. In this case, since each module updates a sequential range of locations, the only possibility for false sharing arises at the boundaries of these ranges when different modules need to write to the same data-bus word. Therefore, the writes to the other locations do not need any synchronization. This optimization is utilized in the matrix multiplication example to reduce the synchronization requirements of the outermost `map` pattern.

A special situation arises when the writes from the patterns are sequential and we can statically determine that an output collection is accessed starting from a memory address that is aligned to the data-bus width; this is sometimes possible for `array` and `vector` datatypes when they do not have runtime determined variables used in their access. In this case, by controlling the work assigned to each module, we can ensure that last memory address written to by the different modules aligns with the end of the data-bus word. In such a case, the results from different modules never overlap, thereby avoiding the need for any synchronization between them. Figure 5.9 (b) illustrates this case, again, using vector addition.

These relaxed synchronization rules are used in our high-level compiler to identify synchronization requirements and correctly parallelize kernels across multiple modules.

5.4.3 Generating the Complete System

The synchronization requirements determined by the high-level compiler are used to generate hardware modules that correctly acquire mutex locks while updating shared data structures and, therefore, can execute in parallel. Furthermore, the toolchain described in Section 5.3 produces multiple variants (hardware implementations) for each parallel kernel by varying the degree of parallelism (i.e., loop-unrolling and pipelining) exploited in the variant. The additional ability to parallelize computations across multiple modules further widens the design space making it hard to find the optimum *system configuration*, which

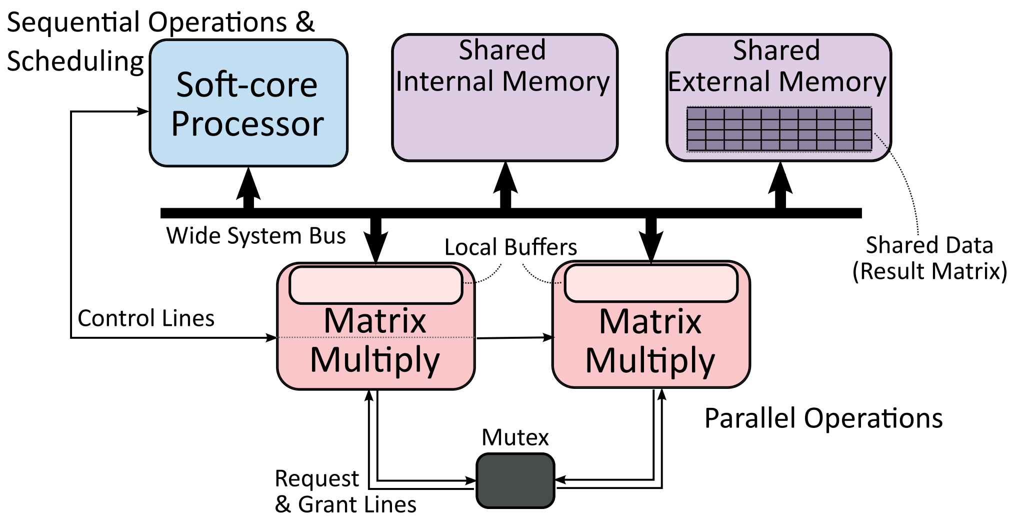


Figure 5.10: Multiple hardware modules can be used to improve the performance of matrix multiplication. The system uses a mutex to synchronize these modules while writing to the result matrix in the shared memory. Each module connects to this mutex using separate request and grant lines.

is the number and types of variants used per parallel kernel, in the final system design.

Design Space Exploration

We utilize the performance and resource estimates from the HLS tool to guide the design space exploration. To find the optimum configuration, we model this as an *Integer Linear Programming* (ILP) problem and use an ILP solver, as done by Graf et al. [Graf et al., 2014]. In the ILP formulation, we try to maximize the performance of the application while ensuring that at least one variant is selected for each parallel kernel and the design fits on the FPGA device. However, this modeling is approximate since it is based only on static analysis and it assumes that each extra hardware module provides the performance improvement as indicated in the HLS estimates; the latter implies that external factors, such as maximum system bandwidth and contention between modules over shared data, do not significantly affect performance. To make this assumption reasonable, we address the bandwidth problem with an ILP constraint to ensure that the total bandwidth used by all modules of a given kernel is less than the maximum bandwidth of the system bus. For overcoming the contention problem, we use the information about the synchronization requirements to identify kernels where the computation at the innermost level of nesting 1) performs 'locked' updates to shared datastructures and 2) contains no high-latency operations (e.g., reads from external memory). Since such kernels are likely to have contention when parallelized, we add a constraints to ensure that we only use one hardware module for these kernels. With these additional constraints, the ILP solver finds the optimal configuration of the system design for the application based on the model.

System Generation

The configuration found by the ILP solver is used to select the hardware modules and integrate them into a system-level template that connects them to the other shared components (as done in toolchain described in Section 5.3). For instance, Figure 5.10 shows the automatically generated system for matrix multiplication with the kernel parallelized across two modules. To automatically add hardware mutexes and connect them to the different modules, we utilize the synchronization information for each kernel. While connecting the hardware mutexes, we use the performance estimates from the HLS tool to provide a higher priority to kernel variants that achieve higher performance; this ensures that the high performance variants are given priority when multiple modules contend to acquire mutex locks.

5.4.4 Managing the Multiple Modules

Now that we can generate complete systems where the kernel computations are parallelized across multiple hardware modules, we need a work sharing scheme to partition the work among these modules. In our toolchain, the Delite compiler represents the work for each computational pattern as iterations over a sequential index range. Therefore, we could statically divide this range into parts that are assigned to separate hardware modules. But, the processing times can vary widely due to conditional and data-dependent operations within the kernel. Additionally, for each kernel, we can have hardware modules with different processing performance. To tolerate this variability, we employ a dynamic load balancing scheme with a central task-pool [Korch and Rauber, 2004] to distribute the work. To implement this, we store the iteration index range for the outermost pattern, which is parallelized across the modules, in a shared data structure (task-pool). During execution, each module dynamically updates this index range, taking away a small portion of the work for execution. Since the modules that finish faster take away portions more frequently, we achieve the load balancing.

```
// A and B are input matrices and
// C is the output matrix computed as A*B
// Furthermore, D is matrix of precomputed values of C and
// D_valid indicates if the value if D is valid.
// rows_A is the number of rows in matrix A and
// cols_D number of columns in D
val C = (0::rows_A, 0::cols_D){ (i,j) =>
    // Compute the dot-product only when D_valid(i,j) is zero
    if( D_valid(i,j) == 0 ) {
      // Compute the elements of C as a dot-product
      // between rows of A with columns of B
      (A.getRow(i) * B.getCol(j)).sum
    } else {
      // The precomputed value in D(i,j) as the result of
      // the dot-product computation
      D(i,j)
    }
 }
```

Figure 5.11: Modified matrix multiplication. This application computes matrix `C` by multiplying matrices `A` and `B`. However, if matrix `D_valid` indicates that the precomputed results of some cells exists in matrix `D`, it copies these values directly from `D`.

To illustrate the benefit of this scheme, consider the application in Figure 5.11, which is a slightly modified variant of the matrix multiplication example we saw earlier. In this application, the matrix D holds precomputed results for some cells in C and values in D_{valid} state if the corresponding cells in D hold valid data. Therefore, for cells where the entry in D_{valid} is one, the multiplication result can be directly copied over from the matrix D, thereby, avoiding the costly dot-product computation. Hence, the processing time for different rows of C can vary significantly based on the number of zeros in D_{valid} making static work partitioning suboptimal. Figure 5.12 compares the execution times for computing matrix C when it is parallelized across multiple modules (ranging from one to four) with static and dynamic workload partitioning schemes. Here, the execution time remains fixed with static workload partitioning because the D_{valid} matrix we used has zero entries only in the first quarter of its rows resulting in a unfair work distribution that put all the heavy computation on the same hardware module. With dynamic workload partitioning, however, the modules take away

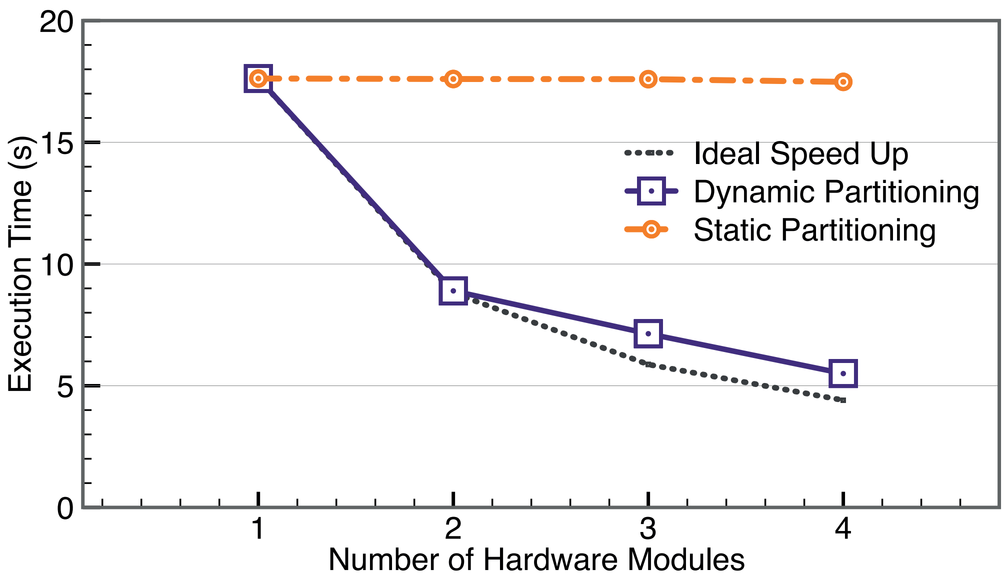


Figure 5.12: The static workload partitioning is inefficient because the computational workload in the kernel is data dependent. In the specific example used, the static partitioning scheme assigns all the heavy computation to the same hardware module eliminating all the benefits of parallelization. In the dynamic partitioning, the modules take away smaller portions of computation and compete with each other to complete the task leading to a more fairer work distribution among the modules and, as a result, achieve better execution time. The contention over access to the shared task-pool in the dynamic partitioning scheme results in a small overhead compared to the ideal parallelization.

smaller units of work each time and compete to complete the execution. This results in a fairer workload distribution and, consequently, a smaller execution time. But, since all the modules need to access and update the task-pool dynamically, the accesses to this task-pool must be guarded using a mutex and this results in a slight increase in the execution time compared to the ideal speed-up indicated in the figure.

We augmented the toolchain described in Chapter 4 using these concepts to automatically generate designs that parallelize computation across multiple modules.

5.5 Evaluation Results

In order to illustrate the benefits of parallelizing computation across multiple modules, we select seven applications from linear algebra, signal processing and graph processing domains.

5.5.1 Evaluation Setup and Methodology

Hardware Setup and Methodology

All the applications were written in OptiML [Sujeeth et al., 2011]. Although OptiML is primarily a language for machine learning, it supports a rich set of datatypes and operations that are sufficient to develop these benchmark applications. The applications were compiled using our toolchain that was augmented with the ideas presented in Section 5.4 and generated into hardware designs. The toolchain used Vivado HLS 2013.4 [Xilinx, 2013a] to synthesize parallel kernels in the application into hardware modules and Vivado Design Suite [Xilinx, 2013b] to connect these hardware modules within a system design template and generate the FPGA bitstream. The generated bitstreams were executed on the Xilinx VC707 development board that houses a XC7VX485T device and has 1GB of DRAM. Each application's performance was measured with hardware counters during execution and the resource consumption values are from the post-implementation reports generated by Vivado Design Suite.

In order to evaluate the benefits of our approach, we compared the performance obtained by utilizing only a single hardware module for each parallel kernel (single module design) with that of the multi-module design. We used the toolchain described in Chapter 4 to automatically generate single module designs. Then, we used the modified toolchain that was extended with the ideas presented this chapter to generate the multi-module designs. Note that for both cases we started from exactly the same application specifications written in OptiML. Furthermore, in each case, the toolchain automatically generated the hardware design after performing the design space exploration and determining the configurations that maximized the application performance by using up to 80% of the resources on the FPGA.

5.5.2 Evaluating with Application Benchmarks

Applications used for benchmarking

We used the following applications to evaluate our approach:

1. *Matrix Multiplication–unoptimized* (**MMuopt**) is a linear algebra application that multiplies two square floating-point matrices with 250,000 elements each. This is the running example we have used throughout this chapter.

Table 5.1: Parallel operations and computational patterns in each application

Application	Parallel Ops.	Map	ZipWith	Reduce	Foreach
MMuopt	3	3	1	1	0
MMopt	3	3	1	1	1
ACorr	5	6	1	5	0
PRank	6	5	0	3	0
PFrnd	6	4	0	0	5
TCount	3	6	0	1	0
BFS	9	6	0	1	3

2. *Matrix Multiplication–optimized* (**MMopt**) is a more optimized version of the matrix multiplication that buffers the columns of the second matrix and uses the buffered data to compute multiple cells of the result matrix.
3. *1-D Autocorrelation* (**ACorr**) is a signal processing application that computes the autocorrelation of a 20,000-element floating-point vector.
4. *PageRank* (**PRank**) is a popular graph algorithm used in search engines that iteratively computes the weights of each node in the graph based on the weights of nodes in its in-neighbor set (nodes with edges leading to it). We used a graph with 1,000,000 nodes.
5. *Potential Friends* (**PFrnd**) uses the principle of triangle completion in graphs to recommend new connections (friends) for each node. We used a 15,000-node graph.
6. *Triangle Counter* (**TCount**) counts the number of triangles in a graph with 1,000,000 nodes.
7. *Breadth First Search* (**BFS**) computes the distance of every node in a 1,000,000 node-graph from a given source node.

Since we did not have a mechanism to read data from an external source, the generation of test data (e.g., input matrices, vector and arbitrary graphs) was made a part of the application. But, this did not affect the results since the time needed for data generation is insignificant compared to total execution time. Table 5.1 lists the number of parallel kernels and the computational patterns in each application. Performance results for these designs are shown in Figure 5.13 and their resource consumptions are reported in Table 5.2.

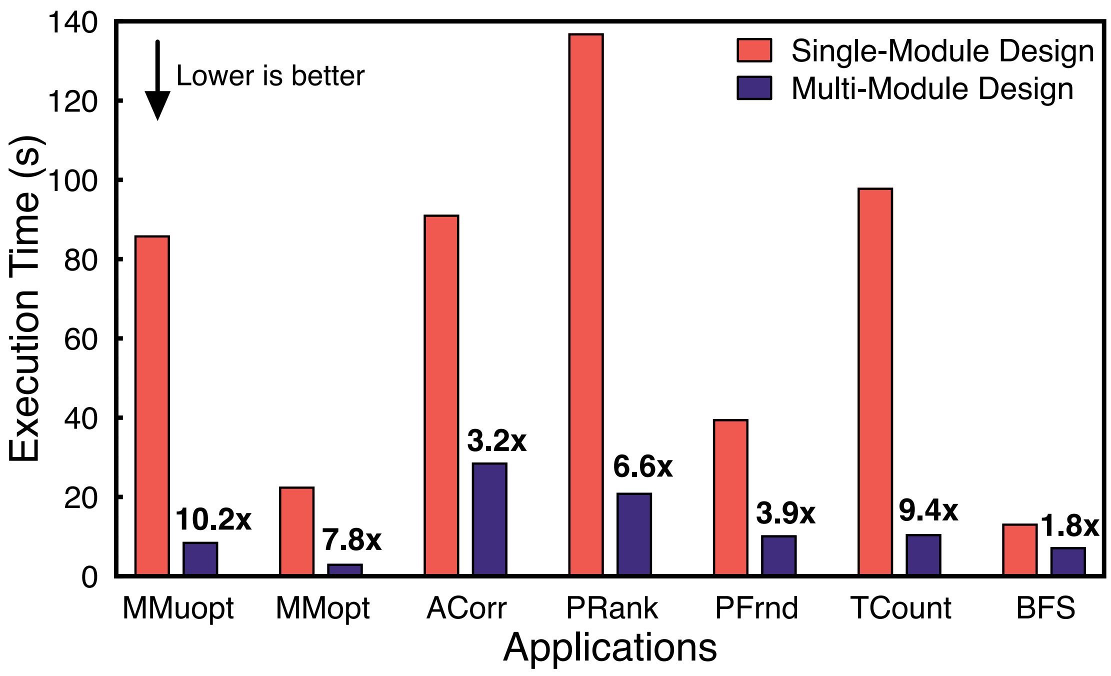


Figure 5.13: The figure shows the relative performance of the designs with a single module per kernel and those with multiple modules per kernel. Since the HLS generated modules in these applications do not fully utilize the bandwidth of the system bus, parallelizing kernels across multiple modules always achieves higher performance.

Comparing Single-Module and Multi-Module Designs

Performance comparison between the linear algebra applications

Across all applications, as expected, the performance obtained by exploiting multi-module parallelism was better than that of the single module design. This is because the hardware modules in these single module designs had low effective parallelism, either due to irregular or strided access patterns, or due to the long latency of the computation; hence, they did not fully utilize the system-bus bandwidth. The highest performance improvement was for **MMuopt** where the performance of the single module design was limited due to the strided access pattern, as discussed in the Section 5.1. The multi-module parallelization improved the aggregate data bandwidth to this kernel and, thereby, its performance, as shown in Figure 5.3; the optimizations discussed in Section 5.4.2 reduced the synchronization needed in the outer-level `map` of the matrix multiplication kernel and aided to deliver this increased performance. In **MMopt**, the overall performance of the matrix multiplication improved because buffering the columns of the second matrix

Table 5.2: Percentage utilization of resource in each application

Apps.	**Single Module Design (%)**				**Multi-Module Design (%)**			
	LUTs	FFs	BRAMs	DSPs	LUTs	FFs	BRAMs	DSPs
MMuopt	13.60	6.65	14.56	1.79	82.10	37.88	59.42	21.07
MMopt	15.26	7.49	14.56	1.79	71.60	3.70	48.35	16.25
ACorr	29.62	15.97	18.45	4.07	69.42	36.77	46.50	8.00
PRank	44.36	22.35	23.79	7.57	67.08	30.56	53.20	3.39
PFrnd	63.27	31.40	18.54	14.71	80.62	34.48	51.84	4.32
TCount	51.92	28.04	13.40	14.25	89.07	34.56	58.06	2.36
BFS	65.65	29.82	32.04	0.32	73.45	33.64	63.50	1.18

reduced the total data read for the computation; yet, the multi-module design achieved better performance due to the parallelization and the resulting improvement in bandwidth utilization.

Performance comparison between the signal processing application implementations

In the **ACorr**, the kernels have very regular access patterns. However, due to the long latency of the floating point operations, the data bandwidth used by the single module design was lower than the available bus bandwidth. Multi-modules designs improved the bandwidth utilization, and the optimization discussed in Section 5.4.2 enabled the most critical kernels, which contained fused `zipWith-reduce` operations, to operate in parallel with very little synchronization overhead.

Performance comparison between the graph processing applications

In the graph applications, **PRank**, **PFrnd**, **TCount** and **BFS**, the performance improvements were again due to improved data access bandwidth to critical kernels with irregular access patterns. Additionally, the optimizations discussed in Section 5.4.2 were able to remove all the synchronization requirements for the most critical kernel in **PRank** and significantly reduced the synchronization requirements for **TCount**, improving their performances. In **PFrnd** and **BFS**, the ILP solver correctly allocated more resources to the most critical kernels in these applications and achieved 8× and 3.7× improvements, respectively, for these kernels. But, as a side-effect, the resources allocated to the less critical kernels reduced and their performance degraded, diminishing the overall improvement to 3.9× and 1.8×, respectively as seen in Figure 5.13. These results demonstrate the performance benefits of the multi-module parallelization approach proposed in this work.

Considering resource consumption, the exact values of resources used varied significantly due to the timing and resource optimizations done by the FPGA tool. However, as seen in Table 5.2, the single kernel designs were not able to achieve better performance, in spite of having unused resources. In comparison, the ability to use multiple modules in our approach enabled the ILP solver to find better design points that made better use of the FPGA resources.

5.6 Discussion

Key insights

FPGAs can be customized into application-specific architectures and leverage spatial parallelism to deliver high performance for some applications. But, to achieve high performance from these devices, application developers need to correctly parallelize computation across multiple modules and carefully balance the computational throughput with data bandwidth. Additionally, developers need to identify synchronization requirements in the application and build custom synchronization schemes, which is both tedious and error prone and above all, hard to achieve without hardware design expertise. Therefore, automated techniques to parallelize applications on FPGAs are vital to enable application developers without hardware design knowledge to benefit from this device. In this chapter, we developed techniques to parallelize operations composed from computational patterns across multiple hardware modules. In order to achieve this, we leveraged the properties of the computational patterns and system architecture to detect synchronization requirements and to automatically generate hardware that uses synchronization primitives where needed to correctly parallelize computation. Moreover, we also identify cases where these synchronization requirements can be relaxed or entirely avoided. This results in a new ability to generate complete computing systems that are several times more efficient than those previously achieved. Most importantly, this completely automated method liberates the application developer from *any* understanding of the hardware platform.

6 Conclusions

Making FPGAs programmable to the masses

Microsoft is introducing FPGAs in data centers and Intel is packaging FPGAs with high-end processors: there is today a unique window of opportunity for reconfigurable technology in the general computing world. To facilitate the rapid adoption and successful deployment of this technology, we need to enhance the ability of the application programmers to build applications targeting FPGAs without *any* hardware design experience. Domain-specific languages that are tuned to specific application domains offer abstraction from low-level hardware details and make it easier to write applications. Additionally, the toolchain can leverage domain-awareness to perform optimization as well as automate the generation of a complete hardware design solution, thereby, reducing the need for hardware design expertise. Therefore, development of tools based on domain-specific languages can play a big role in improving the adoption of FPGAs in different domains. However, there has been only a limited adoption of this approach, primarily due to the high cost and effort needed to develop such tools. The techniques presented in this thesis will considerably reduce the effort needed to develop such tools and, therefore, pave the way to a more widespread adoption of FPGAs in many new application areas.

6.1 Summary

Reducing the effort to build domain-specific tools

In Chapter 3, we illustrated how language embedding and type-directed staging can be used to cost-efficiently develop domain-specific tools. We demonstrated these ideas by using an example of developing a tool to optimize and generate hardware designs for simple matrix expressions. In this work, language embedding enabled us to quickly develop a DSL (e.g., the MatrixDSL we developed to specify matrix expressions) and type-directed staging enabled us to develop a compiler for programs written in this DSL. In our approach, we developed the toolchain by composing reusable optimization modules which could be easily reused to reduce the incremental effort needed to develop new tools. Additionally, we integrated with external tools, such as LegUp and FloPoCo, when possible to further bring down the development effort.

Using computational patterns to create hardware systems

Applications from many domains can be represented using a small number of computational patterns [Asanovic et al., 2006, McCool et al., 2012]. Therefore, in Chapter 4, we developed a methodology to generate hardware implementations by decomposing high-level DSL applications into computational patterns. In this approach, we leveraged the properties of these patterns to perform optimizations and hardware-software partitioning to produce high-performance hardware systems. We developed a tool based on this approach and integrated it into the Delite compiler infrastructure [Sujeeth et al., 2014] to synthesize complete hardware systems for OptiML [Sujeeth et al., 2011] applications. We also demonstrated that this tool could automatically perform optimizations and explore the design space to generate high-quality hardware designs targeting two different implementation targets. These automatically generated designs were up to two orders of magnitude better in performance compared to the unoptimized designs and were more energy-efficient compared to a commercial muti-core CPU.

Improving the parallelism of hardware systems created from computational patterns

In Chapter 5, we extended the methodology developed in Chapter 4 to automatically parallelize the computation across multiple hardware modules. This enabled the designs to better exploit the spatial parallelism of the FPGA to overcome performance bottlenecks, e.g., data starvation caused by non-sequential data access patterns and long memory access latency, that considerably diminished application performance. To achieve this, we exploited the data access properties of the computational patterns to automatically identify and reduce synchronization requirements among the multiple hardware modules. We extended the toolchain developed in Chapter 4 based on this idea to automatically generate hardware systems that parallelize computations across multiple hardware modules. These systems included hardware primitives to perform synchronization and used dynamic workload partitioning schemes to deliver performance improvements. Our evaluation results show that multi-module parallelization improved the utilization of the FPGA and delivered better performance compared to the designs without this parallelization (i.e., designs produced by the tool in Chapter 4).

6.2 Benefits of the Approach

Improved productivity and accessibility non-hardware-designers

Both MatrixDSL we used in Chapter 3 and OptiML we used in Chapters 4 and 5 have syntaxes that are customized to the specific application domain. For instance, their syntaxes supported domain-specific datatypes (e.g., `matrices` and `vectors`) as well as operations on these datatypes that made it easier for users to develop applications. Additionally, both the DSLs enabled users to start from a purely functional specifications that did not include any low-level hardware details, yet they produced high-quality hardware designs. These DSLs illustrate how our methodology can be used to provide a convenient interface to elicit application specifications from users without hardware design expertise.

Improved design quality

In Chapter 3 we showed that domain knowledge can be leveraged to perform optimizations (e.g., utilizing rules of matrix algebra to perform matrix level optimizations) to produce significantly better quality results. Later, in Chapters 4 and 5 we showed that by mapping domain operations into computational patterns we can generate high-quality hardware circuits to implement them. To achieve this, the domain operations were decomposed into a set of computation patterns that captured the essential properties of the computation. Then, by leveraging the properties of the patterns, we could automatically perform optimizations and parallelize the computation on the FPGA to achieve a high performance and very good energy efficiency.

Complete hardware solution

The methodology presented in Chapters 4 and 5 utilized domain knowledge to automatically select a system design template for the hardware design. This system design template provided a clear strategy to integrate together different parts of the application, including those that were generated as hardware circuits or mapping into software, to automatically produce a complete hardware system. Additionally, the template included connections to board-level components and ports (e.g., external memory, UART and JTAG) to ensure that the hardware design could be directly implemented on the target FPGA board. For instance, the tool we developed uses variants of a system design template to automatically generate designs targeting VC707 and ZC706 development boards; the VC707 has a Virtex7 device and the ZC706 contains a Zynq device where the FPGA was integrated with hardened ARM processor and memory controller.

Extensibility

Another strength of our proposed approach is its extensibility. As we demonstrated in Chapter 3. the ideas of language embedding and type-directed staging can be used to cost-efficiently develop DSLs and compilers for these DSLs. This can make it easier to extend the approach to new application domains that can benefit from FPGAs. To support new application domains, there might be a need to add new computational patterns to represent that operations in this domain. But, as proposed in Chapter 4, one can leverage the properties of these new patterns to generate efficient hardware circuits for them. Additionally, if these patterns provide guarantees on how it consumes and produces data, the ideas presented in Chapter 5 can be used to create multi-module designs from these patterns to benefit from the spatial parallelism in FPGA.

6.3 Future Work

New domains and target devices

Driven by energy and performance limitations, today, the world is turning to FPGAs to meet the computational needs of future applications; Catapult [Putnam et al., 2014] from Microsoft and Xeon+FPGA [Gupta, 2015] from Intel are fine examples of this trend. In this regard, this work addresses the vital issue of making tomorrow's computing infrastructure programmable to application developers. But, additional work needs to be done to evaluate how well this approach can be extended to new application domains and target devices. For instance, on the Xeon+FPGA the applications must be partitioned between a powerful CPU and an FPGA device and on the Catapult, there are multiple interconnected FPGAs that can be used to accelerate applications.

Integrated debugging infrastructures

To successfully develop applications for sophisticated targets, users will need capable infrastructures to easily debug applications; this includes debugging both application functionality and its performance. As noted in Chapter 2, there are some efforts on developing debugging facilities for specific HLS tools. It might be interesting to probe how such a facility can be cost-efficiently developed for domain-specific hardware synthesis tools.

Application-specific memory systems

In typical computing systems, the performance and efficiency of the memory system are critical. Our results from Chapter 4 demonstrated how the superior memory system in the CPU gave it an edge over our FPGA designs. Our focus in the work was on generating application-specific computing units and did not investigate how memory systems can be customized for specific applications. There has been some work done in this area [Parashar et al., 2010, Chung et al., 2011], but these efforts do not fully automate the creation of application-specific memory systems. This may be another useful direction to explore in the future.

Reconfigurable technology in the form of FPGAs or similar fabrics can make computing systems faster and more energy-efficient. However, there are still serious hurdles that need to be surmounted before FPGAs can become an integral part of a standard computing system. This thesis represents a step toward the removal of a crucial obstacle by making reconfigurable computing systems easy to program.

Bibliography

Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. Jet: An embedded DSL for high performance big data processing. In *International Workshop on End-to-End Management of Big Data (BigData 2012)*, number EPFL-CONF-181673, 2012.

Andreas Agne, Markus Happe, Ariane Keller, Enno Lubbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. ReconOS: An operating system approach for reconfigurable computing. *IEEE Micro*, 34(1):60–71, 2014.

Altera Corporation. Quartus II Handbook Version 13.0, 2013. URL https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/archives/quartusii_handbook_archive_130.pdf. [Online, accessed 31-Jan-2016].

Altera Corporation. Altera SDK for OpenCL best practices guide, May 2015. URL https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf. [Online; accessed 31-Jan-2016].

Chris Anderson. The end of theory: The data deluge makes the scientific method obsolete, June 2008. URL http://www.wired.com/2008/06/pb-theory. [Online; posted 23-Jun-2008].

João Andrade, Nithin George, Kimon Karras, David Novo, Vitor Silva, Paolo Ienne, and Gabriel Falcão. From low-architectural expertise up to high-throughput non-binary LDPC decoders: Optimization guidelines using high-level synthesis. In *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications*, pages 1–8, London, September 2015.

David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. Achieving programming model abstractions for reconfigurable computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(1):34–44, January 2008. ISSN 1063-8210. doi: 10.1109/TVLSI.2007.912106.

Anna Antola, Marco Domenico Santambrogio, Marco Fracassi, Pamela Gotti, and Chiara Sandionigi. A novel hardware/software codesign methodology based on dynamic reconfiguration with Impulse C and CoDeveloper. In *Proceedings of the 3rd Southern Conference on Field Programmable Logic*, pages 221–224. IEEE, 2007.

Ken Arnold, James Gosling, David Holmes, and David Holmes. *The Java Programming Language*, volume 2. Addison-Wesley, 1996.

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from Berkeley. Technical Report Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

Joshua Auerbach, David F Bacon, Perry Cheng, and Rodric Rabbah. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. *SIGPLAN Notices*, 45(10): 89–108, 2010.

Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Design Automation Conference*, pages 1212–21, San Francisco, Calif., June 2012.

David Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *ACM Queue*, 11(2):40, 2013.

Gordon Bell, Tony Hey, and Alex Szalay. Beyond the data deluge. *Science*, 323(5919):1297–1298, 2009. ISSN 0036-8075. doi: 10.1126/science.1170411. URL http://science.sciencemag.org/content/323/5919/1297.

Peter Bellows and Brad Hutchings. JHDL-an HDL for reconfigurable systems. In *Proceedings of the 6th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 175–184. IEEE, 1998.

Brahim Betkaoui, David B Thomas, and Wayne Luk. Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. In *Proceedings of the 2010 International Conference on Field Programmable Technology*, pages 94–101. IEEE, 2010.

Robert H Bishop. *Learning with LabVIEW*. Prentice Hall, 2014.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *ACM SIGPLAN Notices*, volume 34, pages 174–184. ACM, 1998.

William J Bolosky and Michael L Scott. False sharing and its effect on shared memory performance. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems*, 1993.

Gordon J. Brebner and Weirong Jiang. High-speed packet processing using reconfigurable computing. *IEEE Micro*, 34(1):8–18, 2014. doi: 10.1109/MM.2014.19. URL http://dx.doi.org/10.1109/MM.2014.19.

Nazanin Calagar, Stephen D Brown, and James H Anderson. Source-level debugging for FPGA high-level synthesis. In *Proceedings of the 24th International Conference on Field-Programmable Logic and Applications*, pages 1–8. IEEE, 2014.

Calypto Design Systems. *Catapult Product Family*. Calypto Design Systems, 2014. URL http://calypto.com/en/page/leadform/31. [Online; accessed 31-Jan-2016].

Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 33–36, Monterey, Calif., February 2011. ACM.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. *SIGPLAN Notices*, 46(8):47–56, 2011.

Viraphol Chaiyakul, Daniel D. Gajski, and Loganath Ramachandran. Minimizing syntactic variance with assignment decision diagrams. Technical Report TR-92-34, Department of Information and Computer Science, University of California, Irvine., 1992. URL http://www.cecs.uci.edu/~cad/publications/tech-reports/1992/TR-92-34.syntactic_variances.ps.gz.

Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, efficient data-parallel pipelines. In *SIGPLAN Notices*, volume 45, pages 363–375. ACM, 2010.

Doris Chen and Deshanand Singh. Using OpenCL to evaluate the efficiency of CPUs, GPUs and FPGAs for information filtering. In *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*, pages 5–12. IEEE, 2012.

Jongsok Choi, Stephen Brown, and Jason Anderson. From software threads to parallel hardware in high-level synthesis for FPGAs. In *Proceedings of the 2013 International Conference on Field Programmable Technology*, pages 270–277, Kyoto, Japan, December 2013. IEEE.

Eric S Chung, James C Hoe, and Ken Mai. CoRAM: An in-fabric memory architecture for FPGA-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 97–106. ACM, 2011.

Alessandro Cilardo, Luca Gallo, Antonino Mazzeo, and Nicola Mazzocca. Efficient and scalable OpenMP-based system-level design. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 988–991. IEEE, 2013.

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

Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. GAUT: A high-level synthesis tool for dsp applications. In *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter 9, pages 147–169. Springer, 2008. doi: 10.1007/978-1-4020-8588-8_9.

Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From OpenCL to high-performance hardware on FPGAs. In *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*, pages 531–534, Porto, Portugal, September 2012. IEEE.

Leonardo Dagum and Ramesh Menon. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

Florent De Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design and Test of Computers*, 28(4):18–27, 2011.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Jan Decaluwe. MyHDL: A Python-based hardware description language. *Linux journal*, 2004 (127):5, 2004.

Steven Derrien, Sanjay Rajopadhye, Patrice Quinton, and Tanguy Risset. High-level synthesis of loops using the polyhedral model. In *High-Level Synthesis: From Algorithm to Digital Circuit*, chapter 12, pages 215–230. Springer, 2008. doi: 10.1007/978-1-4020-8588-8_12.

Stephen A Edwards. The challenges of synthesizing hardware from C-like languages. *IEEE Design and Test of Computers*, 23(5):375–386, 2006.

Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *Proceedings of the 20th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 47–56. ACM, 2012.

Nithin George, David Novo, Tiark Rompf, Martin Odersky, and Paolo Ienne. Making domain-specific hardware synthesis tools cost-efficient. In *Proceedings of the 2013 International Conference on Field Programmable Technology*, pages 120–127, Kyoto, December 2013.

Nithin George, Hyoukjoong Lee, David Novo, Tiark Rompf, Kevin Brown, Arvind Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *Proceedings of the 24th International Conference on Field-Programmable Logic and Applications*, pages 1–8, Munich, September 2014.

Nithin George, HyoukJoong Lee, David Novo, Muhsen Owaida, David Andrews, Kunle Olukotun, and Paolo Ienne. Automatic support for multi-module parallelism from computational patterns. In *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications*, pages 1–8, London, September 2015.

Jeffrey Goeders and Steve JE Wilton. Using dynamic signal-tracing to debug compiler-optimized HLS circuits on FPGAs. In *Proceedings of the 23rd IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 127–134. IEEE, 2015.

Maya Gokhale, Jan Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proceedings of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 49–56, Napa Valley, Calif., April 2000. IEEE.

Sebastian Graf, Michael Glas, Jurgen Teich, and Christoph Lauer. Multi-variant-based design space exploration for automotive embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1–6. IEEE, 2014.

David Greaves and Satnam Singh. Kiwi: Synthesis of FPGA circuits from parallel programs. In *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 3–12. IEEE, 2008.

Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Springer, 2010.

Prabhat K Gupta. Xeon+FPGA platform for the data center. In *The Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, volume 119, 2015.

Robert J Halstead and Walid Najjar. Compiled multithreaded data paths on FPGAs for dynamic workloads. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, page 3. ACM, 2013.

Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1815968. URL http://doi.acm.org/10.1145/1815961.1815968.

Jim Handy. *The Cache Memory Book*. Morgan Kaufmann, 1998.

Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. PARO: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications. In *Proceedings of the 4th International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, pages 287–293. Springer, March 2008. doi: 10.1007/978-3-540-78610-8_30.

Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley, 2003.

Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 2011. ISSN 0036-8075. doi: 10.1126/science.1200970. URL http://science.sciencemag.org/content/332/6025/60.

Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. Optimus: Efficient realization of streaming applications on FPGAs. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 41–50. ACM, 2008.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, March 2007. ISSN 0163-5980. doi: 10.1145/1272998.1273005.

Aws Ismail and Lesley Shannon. FUSE: Front-end user framework for O/S abstraction of hardware accelerators. In *Proceedings of the 19th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 170–177, 2011.

Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C Programming Language*, volume 2. Prentice-Hall, 1988.

Srinidhi Kestur, John D Davis, and Oliver Williams. BLAS comparison on FPGA, CPU and GPU. In *Proceedings of the 2010 IEEE Annual Symposium on VLSI*, pages 288–293. IEEE, 2010.

Edwin M Knorr and Raymond T Ng. A unified notion of outliers: Properties and computation. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 219–222, Newport, Calif., August 1997. American Association for Artificial Intelligence.

Jonathan G Koomey, Christian Belady, Michael Patterson, Anthony Santos, and Klaus-Dieter Lange. Assessing trends over time in performance, costs, and energy use for servers. Technical report, Lawrence Berkeley National Laboratory, Stanford University, Microsoft Corporation, and Intel Corporation, 2009.

Matthias Korch and Thomas Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience*, 16(1):1–47, 2004.

Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007. doi: 10.1109/TCAD.2006.884574.

Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004.

HyoukJoong Lee, Kevin J Brown, Arvind K Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

YY Leow, CY Ng, and W.F Wong. Generating hardware from OpenMP programs. In *Proceedings of the 2006 International Conference on Field Programmable Technology*, pages 73–80. IEEE, 2006.

Sen Ma, Zeyad Aklah, and David Andrews. A run time interpretation approach for creating custom accelerators. In *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications*, pages 1–4. IEEE, 2015.

Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design and Test of Computers*, 26(4):18–25, 2009.

Maxeler Technologies. Maxcompiler. Technical report, Maxeler Technologies, 2011. URL https://www.maxeler.com/media/documents/MaxelerWhitePaperMaxCompiler.pdf.

Michael D McCool, Arch D Robison, and James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.

Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 706–706. ACM, 2006.

Mentor Graphics. DK Design Suite–Handle-C, 2015. URL http://s3.mentor.com/public_documents/datasheet/products/fpga/handel-c/dk-design-suite/dk-ds.pdf. [Online; accessed 31-Jan-2016].

Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

Peter Milder, Franz Franchetti, James C Hoe, and Markus Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 17(2):15, 2012.

Karthik Nagarajan, Brian Holland, Alan D George, K Clint Slatton, and Herman Lam. Accelerating machine-learning algorithms on FPGAs using pattern-based decomposition. *Journal of Signal Processing Systems*, 62(1):43–63, 2011.

Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A survey and evaluation of FPGA high-level synthesis tools. PP(99):1–1, 2016. ISSN 0278-0070. doi: 10.1109/TCAD.2015.2513673.

Chris J Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, et al. Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the 9th*

Annual IEEE/ACM International Symposium on Code Generation and Optimization, pages 224–235. IEEE, 2011.

Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing.* O'Reilly Media, Inc., 1996.

Rishiyur Nikhil. Bluespec system verilog: Efficient, correct RTL from high level specifications. In *Proceedings of the 4th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 69–70. IEEE, 2004.

NVIDIA Corporation. CUDA C programming guide, 2015. URL http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical Report LAMP-REPORT-2004-006, School of Computer and Communications, Ecole Polytechnique Fédérale de Lausanne (EPFL), 2004. URL http://www.scala-lang.org/docu/files/ScalaOverview.pdf.

Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. Spiral in Scala: Towards the systematic construction of generators for performance libraries. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, pages 125–134, New York, March 2013. ACM.

Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D Antonopoulos. Synthesis of platform architectures from OpenCL programs. In *Proceedings of the 19th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 186–193. IEEE, 2011.

Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and W-MW Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *Proceedings of the 7th IEEE Symposium on Application Specific Processors*, pages 35–42, San Francisco, Calif., July 2009. IEEE.

Angshuman Parashar, Michael Adler, Kermin Fleming, Michael Pellauer, and Joel Emer. LEAP: A virtual platform architecture for FPGAs. In *The First Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, 2010.

Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. Generating configurable hardware from parallel patterns. *Computing Research Repository*, abs/1511.06968, 2015. URL http://arxiv.org/abs/1511.06968.

Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st Annual International Symposium on Computer Architecture*, pages 13–24. IEEE, 2014.

Oliver Reiche, Moritz Schmid, Frank Hannig, Richard Membarth, and Jürgen Teich. Code generation from a domain-specific language for C-based HLS of hardware accelerators. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–10. IEEE, 2014.

Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Communications of the ACM*, 55(6):121–130, 2012.

Tiark Rompf, Arvind K Sujeeth, Nada Amin, Kevin J Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 48, pages 497–510. ACM, ACM, 2013.

Kyle Rupnow, Yun Liang, Yinan Li, and Deming Chen. A study of high-level synthesis: Promises and challenges. In *Proceedings of the IEEE 9th International Conference on ASIC*, pages 1102–1105. IEEE, 2011.

Hayden Kwok-Hay So. *BORPH: An Operating System for FPGA-based Reconfigurable Computers*. ProQuest, 2007.

Richard Stallman et al. Using the GNU compiler collection, 2015. URL https://gcc.gnu.org/onlinedocs/gcc-5.3.0/gcc.pdf. [Online; accessed 31-Jan-2016].

John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(1-3):66–73, 2010. doi: 10.1109/MCSE.2010.69.

Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education, 1986.

Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. OptiML: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning*, pages 609–616, 2011.

Arvind K Sujeeth, Kevin J Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13 (4s):134, 2014.

Synopsys Inc. Synphony C compiler, 2014. URL http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Documents/synphonyc-compiler-ds.pdf. [Online, accessed 31-Jan-2016].

Mingxing Tan, Bin Liu, Steve Dai, and Zhiru Zhang. Multithreaded pipeline synthesis for data-parallel kernels. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, pages 718–725. IEEE, 2014.

The MathWorks. HDL Coder: Getting started guide, September 2015. URL http://cn.mathworks.com/help/pdf_doc/hdlcoder/hdlcoder_gs.pdf. [Online, accessed 31-Jan-2016].

Simon Thompson. *Haskell: The Craft of Functional Programming*, volume 2. Addison-Wesley, 1999.

Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 39th International Conference on Parallel Processing Workshops*, pages 207–216. IEEE, 2010.

Justin L Tripp, Maya B Gokhale, and Kristopher D Peterson. Trident: From high-level language to hardware circuitry. *Computer*, 40(3):28–37, 2007.

Guido Van Rossum and Fred L Drake. *An Introduction to Python*. Network Theory Ltd. Bristol, 2003.

Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. In *Proceedings of the 18th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 127–134. IEEE, 2010.

Kazutoshi Wakabayashi. CyberWorkBench: Integrated design environment based on C-based behavior synthesis and verification. In *IEEE VLSI-TSA International Symposium on VLSI Design, Automation and Test*, pages 173–176. IEEE, 2005.

Ying Wang, Xuegong Zhou, Lingli Wang, Jian Yan, Wayne Luk, Chenglian Peng, and Jiarong Tong. SPREAD: A streaming-based partially reconfigurable architecture and programming model. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2179–2192, 2013.

Felix Winterstein, Kermin Fleming, Hsin-Jung Yang, Samuel Bayliss, and George Constantinides. MATCHUP: Memory abstractions for heap manipulating programs. In *Proceedings of the 23rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 136–145. ACM, 2015.

Xilinx. Vivado design suite user guide: High-level synthesis, November 2013a. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug902-vivado-high-level-synthesis.pdf. [Online, accessed 31-Jan-2016].

Xilinx. Vivado design suite user guide: Getting started, November 2013b. URL http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug910-vivado-getting-started.pdf. [Online, accessed 31-Jan-2016].

Xilinx. Software defined specification environment for networking (SDNet), March 2014. URL http://www.xilinx.com/support/documentation/backgrounders/sdnet-backgrounder.pdf. [Online, accessed 31-Jan-2016].

Xilinx. SDAccel Development Enviroment, October 2015. URL http://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf. [Online, accessed 31-Jan-2016].

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, volume 8, pages 1–14, 2008.

Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 23rd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 161–170. ACM, 2015.

Curriculum Vitae

Areas of Specialization

High-level synthesis, domain-specific languages, hardware design, reconfigurable computing and computer architecture.

Education

2010–2016	**PhD in Computer and Communication Sciences** *Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland* *Thesis:* FPGAs for the Masses: Affordable Hardware Synthesis from Domain-Specific Languages *Supervisor:* Paolo Ienne
2007–2009	**MSc in Communication Engineering** *Technische Universität München (TUM), Germany* *Thesis:* Implementing Logic on the Routing Infrastructure of an FPGA *Supervisors:* Ulf Schlichtmann and Paolo Ienne
2000–2004	**BTech in Electronics and Communication Engineering** *Model Engineering College, India* *Graduation Project:* Implemented a Logic Analyzer using an FPGA

Publications

Nithin George, HyoukJoong Lee, David Novo, Muhsen Owaida, David Andrews, Kunle Olukotun, and Paolo Ienne. Automatic support for multi-module parallelism from computational patterns. In *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications*, pages 1–8, London, September 2015

João Andrade, Nithin George, Kimon Karras, David Novo, Vitor Silva, Paolo Ienne, and Gabriel Falcão. From low-architectural expertise up to high-throughput non-binary LDPC decoders: Optimization guidelines using high-level synthesis. In *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications*, pages 1–8, London, September 2015

Nithin George, Hyoukjoong Lee, David Novo, Tiark Rompf, Kevin Brown, Arvind Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *Proceedings of the 24th International Conference on Field-Programmable Logic and Applications*, pages 1–8, Munich, September 2014

Nithin George, David Novo, Tiark Rompf, Martin Odersky, and Paolo Ienne. Making domain-specific hardware synthesis tools cost-efficient. In *Proceedings of the 2013 International Conference on Field Programmable Technology*, pages 120–127, Kyoto, December 2013

Yehdhih Ould Mohammad Moctor, Nithin George, Hadi Parandeh-Afshar, Paolo Ienne, Guy Lemieux, and Philip Brisk. Reducing the cost of floating-point mantissa alignment and normalization in FPGAs. In *Proceedings of the 20th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 255–64, Monterey, Calif., February 2012

João Andrade, Nithin George, Kimon Karras, David Novo, Vitor Silva, Paolo Ienne, and Gabriel Falcão. Fast design space exploration using Vivado HLS: Non-binary LDPC decoders. In *Proceedings of the 23rd IEEE Symposium on Field-Programmable Custom Computing Machines*, page 97, Vancouver, Canada, May 2015