5-2017

A Hybrid Partially Reconfigurable Overlay Supporting Just-In-Time Assembly of Custom Accelerators on FPGAs

Zeyad Tariq Aklah
University of Arkansas, Fayetteville

A Hybrid Partially Reconfigurable Overlay Supporting Just-In-Time Assembly of Custom Accelerators on FPGAs

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Engineering

by

Zeyad Tariq Aklah
University of Basrah
Bachelor of Science in Computer Engineering, 2002
University of Basrah
Master of Science in Computer Engineering, 2008

May 2017
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council

Dr. David Andrews
Dissertation Director

Dr. John Gauch
Committee Member

Dr. Michael Gashler
Committee Member

Dr. Xuan Shi
Committee Member

Abstract

The state of the art in design and development flows for FPGAs are not sufficiently mature to allow programmers to implement their applications through traditional software development flows. The stipulation of synthesis as well as the requirement of background knowledge on the FPGAs' low-level physical hardware structure are major challenges that prevent programmers from using FPGAs. The reconfigurable computing community is seeking solutions to raise the level of design abstraction at which programmers must operate, and move the synthesis process out of the programmers' path through the use of overlays. A recent approach, Just-In-Time Assembly (JITA), was proposed that enables hardware accelerators to be assembled at runtime, all from within a traditional software compilation flow. The JITA approach presents a promising path to constructing hardware designs on FPGAs using pre-synthesized parallel programming patterns, but suffers from two major limitations. First, all variant programming patterns must be pre-synthesized. Second, conditional operations are not supported.

In this thesis, I present a new reconfigurable overlay, URUK, that overcomes the two limitations imposed by the JITA approach. Similar to the original JITA approach, the proposed URUK overlay allows hardware accelerators to be constructed on FPGAs through software compilation flows. To this basic capability, URUK adds additional support to enable the assembly of pre-synthesized fine-grained computational operators to be assembled within the FPGA.

This thesis provides analysis of URUK from three different perspectives; utilization, performance, and productivity. The analysis includes comparisons against High-Level Synthesis (HLS) and the state of the art approach to creating static overlays. The tradeoffs conclude that URUK can achieve approximately equivalent performance for algebra operations compared to HLS custom accelerators, which are designed with simple experience on FPGAs. Further, URUK shows a high degree of flexibility for runtime placement and routing of the primitive operations. The analysis shows how this flexibility can be leveraged to reduce communication overhead among tiles, com-

pared to traditional static overlays. The results also show URUK can enable software programmers without any hardware skills to create hardware accelerators at productivity levels consistent with software development and compilation.

Acknowledgements

I would like to thank my academic advisor, Professor David Andrews, for his extraordinary care and support. His guidance helped me throughout the research and the writing of this thesis.

Beside my advisor, I would like to extend my thanks to my thesis committee: Professor John Gauch, Dr. Gashler, and Dr. Shi.

Additionally, I would like to thank the HCED-Iraq for providing me the opportunity to pursue my Ph.D.

My sincere thanks also goes to the staff of the Sponsored Students Program at the University of Arkansas, especially to the director of the program, Gloria Flores Passmore, and the program coordinator, Catherine Cunningham.

Last but not least, I would like to thank my family for their everlasting support.

Contents

1 Interoduction **1**
- 1.0.1 Compilation Challenges . . . 4
- 1.0.2 Technical Challenges . . . 5
- 1.1 Just-In-Time Assembly . . . 5
- 1.2 Thesis Statement . . . 6
  - 1.2.1 Thesis Contributions and Organization . . . 7

2 Background **9**
- 2.1 FPGAs Overview . . . 9
- 2.2 CGRAs Overview . . . 11
- 2.3 High Level Synthesis . . . 13
- 2.4 Overlays . . . 14
  - 2.4.1 JIT FPGA . . . 15
  - 2.4.2 Virtual FPGAs . . . 15
  - 2.4.3 ZUMA: . . . 16
  - 2.4.4 Intermediate Fabrics . . . 16
  - 2.4.5 Synthesis-Free JIT Compilation: . . . 16
  - 2.4.6 SCGRA . . . 17
  - 2.4.7 QuickDough . . . 17
  - 2.4.8 QUKU: . . . 18
  - 2.4.9 Soft Processors . . . 18
- 2.5 Domain Specific Languages . . . 19
  - 2.5.1 FSMLang: . . . 19
  - 2.5.2 Chisel: . . . 20
  - 2.5.3 Aspen: . . . 20
  - 2.5.4 Lime: . . . 21
  - 2.5.5 Delite: . . . 21

3 Just-In-Time Assembly **23**
- 3.1 Introduction . . . 23
- 3.2 JITA Approach . . . 23
- 3.3 Compilation Flow . . . 24
- 3.4 JITA Overlay . . . 25
  - 3.4.1 PR Tiles . . . 28
  - 3.4.2 Programmable Switch . . . 28
  - 3.4.3 Local Memory . . . 29
- 3.5 Run Time Interpreter . . . 29
- 3.6 Summary . . . 34

4 Proposed Solution **37**
- 4.1 Introduction . . . 37
- 4.2 Hardware Design Flow . . . 38

4.3 URUK Architecture . . . 40
4.3.1 Tile Structure . . . 42
4.3.2 PR Regions . . . 42
4.3.3 Configurable Switches: . . . 43
4.3.4 Memory Interface . . . 45
4.3.5 Tile Controller . . . 45
4.3.6 Tile Instruction Sets . . . 46
4.4 Design Automation . . . 50

5 URUK Compilation Flow 51
5.1 Introduction . . . 51
5.2 URUK Parallelism . . . 54
5.3 Conditional Operations . . . 54
5.4 Domain Specific Langauges . . . 54
5.5 Data Flow Graph . . . 55
5.5.1 Pattern Based . . . 55
5.5.2 Operator Based . . . 56
5.6 Example 1 . . . 57
5.7 Example 2 . . . 60

6 Evaluation 66
6.1 Benchmark: . . . 66
6.2 HLS Implementation . . . 67
6.3 URUK Implementation . . . 68
6.4 Prototyping System . . . 70
6.5 Performance Evaluation . . . 74
6.5.1 Pattern Based . . . 77
6.5.2 Operator Based . . . 81
6.5.3 Optimization . . . 82
6.6 Productivity . . . 84
6.7 Dynamic vs. Static . . . 85
6.7.1 Area . . . 88
6.7.2 Routing Data . . . 88
6.7.3 Parallelism . . . 89

7 Conclusion 90
7.1 Summary . . . 90
7.2 Future Work . . . 91

References 93

List of Figures

1.1 General Processors vs. FPGAs Compilation Flows. 3

2.1 General FPGA Architecture. 11

2.2 General CGRA Architecture. 12

3.1 Design Approach. 24

3.2 Compiler Flow and VAM Call Generation. 26

3.3 3×3 Tile Array and Interconnect Network. 27

3.4 Switch Routing. 28

3.5 VAM Run Time Interpreter Configuration Steps. 30

3.6 Design Portability with JIT. 31

3.7 VAM Run Time Interpreter Configuration Steps For Inner Product. 33

3.8 All variant patterns must be pre-synthesized. Modfied source from OptiML[58]. . . 36

3.9 Composing conditional operations problem. Modfied source from OptiML[58]. . . 36

4.1 DSL parallel programming patterns. 38

4.2 Hardware design flow for the overlay static logic and partial bitstreams. 39

4.3 Patterns/Operators HLS template. 40

4.4 2D URUK Overlay Structure. 41

4.5 Three tile interconnect types. 44

4.6 Tile Controller. 46

4.7 URUK instructions set and operations code 47

5.1 URUK compilation flow. 52

5.2 A DSL source code. 58

5.3 Data Flow Graphs (DFGs) for the code in Figure 5.7 58

5.4 Place&Route DFGs in Figure 5.8 59

5.5 Instructions and executable binaries for the two placement examples in Figure 5.9 59
5.6 Instructions and executable binaries for the two placement examples in Figure 5.9 60
5.7 OptiML Example 61
5.8 Data Flow Graph for the code in Figure 5.7 61
5.9 Place&Route DFGs in Figure 5.8 62
5.10 Instructions and executable binaries for the pattern based placement in Figure 5.9 (a). 63
5.11 Instructions and executable binaries for the operator based placement in Figure 5.9 (b). 64

6.1 Prototype System. 70
6.2 The DFG of CP1 function based on pre-synthesized (a) Patterns and (b) Operators, and the placement on 3×3 overlay. 72
6.3 The DFG of CP2 function based on pre-synthesized (a) Patterns and (b) Operators, and the placement on 3×3 overlay. 73
6.4 The DFG of CP3 function based on pre-synthesized (a) Patterns and (b) Operators, and the placement on 3×3 overlay. 74
6.5 The DFG of CP1 function based on pre-synthesized (a) Patterns and (b) Operators, and the placement on 3×3 overlay. 75
6.6 Vector Addition and Matrix Multiplication DFGs, Pattern Based. Also, the three mapping methods, USingle, UDouble, and UQuad. 76
6.7 The execution time of the 7 benchmark functions. Only CP1 function was not implemented in UQuad due to the limited number of tiles. The HLS accelerators were not optimized. 4K data elements(32-bit integers) are used. 79
6.8 The speed ups of the HLS and URUK (USingle, UDouble, and UQuad) implementations over software versions of (a) CP1 and (b) CP2 functions. 79
6.9 The speed ups of the HLS and URUK (USingle, UDouble, and UQuad) implementations over software versions of (a) CP3 and (b) CP4 functions. 80

6.10 The speed ups of the HLS and URUK (USingle, UDouble, and UQuad) implementations over software versions of (a) Vector add and (b) Vector multiply functions. . 80
6.11 Compares the execution time and speed up of UPB and UOB implementations using 4K data elements(32-bit integers). 82
6.12 The execution time of the optimized and non-optimized HLS of 64x64 matrix multiply also the speed up compared to MicroBlaze and the three methods of URUK overlay implementations using 4K data elements(32-bit integers). 83
6.13 The required design experience on FPGA to optimize the 64x64 matrix multiply, in Figure 6.12, on both HLS and URUK to gain speed up. 83
6.14 (a) The HLS design steps including estimated time; (b) URUK compilation steps including estimated time. 86
6.15 Dynamic vs. static overlays 87

List of Tables

2.1 DSLs for design applications on FPGAs. 19

3.1 VAM Calls 32

4.1 Conditional branching instructions. 49

6.1 Synthetic Benchmark Functions 67

6.2 Resource Utilizations of HLS Full Accelerators on Vertix7 68

6.3 Resource utilizations of Computational Operators on Virtex7 68

6.4 Tile's Resource utilization on Virtex7 69

6.5 Resource utilizations of Programming Patterns on Virtex7 69

6.6 Tile Utilizations of Benchmark Functions on URUK Overlay. 70

Terms and Definitions

API Application Programmer Interface. The defined interface of a piece of software, often times a library or operating system.

CP Functions that include Compound Patterns.

CPU Central Processing Unit. A programmable hardware component, often referred to as processor or core.

DMA Direct Memory Access. Often referring to hardware devices that can perform memory-to-memory operations without processor assistance.

DFG Data Flow Graph.

DSP Digital Signal Processor. A processor specialized for signal processing, often featuring vector and multiply-accumulate operations.

FIFO First-In, First-Out. A hardware component or data structure that exhibits First-In, First-Out behavior (e.g. a queue).

FPGA Field Programmable Gate Array. A hardware chip whose functionality can be changed post-fabrication.

GPU Graphics Processing Unit. A hardware component specialized for graphics processing.

HDL Hardware Description Language (e.g. VHDL or Verilog).

HLL High-Level Language.

HLS High Level Synthesis. Synthesis the hardware design from hight level programming language. Or the abbreviation of the hight level synthesis tools named Vivado HLS.

HPC High Performance Computing.

HW Abbreviation for Hardware.

ICAP Internal Configuration Access Port

ISA Instruction Set Architecture. Also known as the instruction set of a particular processor.

JIT Just In Time. A compilation approach.

JITA Just-In-Time Assembly of hardware custom accelerators.

JVM Java Virtual Machine

LUTs Look up tables.

MM Matrix Multiply.

MPSoC Multiprocessor System-on-Chip.

NUMA Non-uniform Memory Access

OS Operating System.

P2P point to point

PAR place and routing

PE processing element

PIIL Platform Independent Interpreter Language. A language used in the run time interpreter, which is platform independent.

PRA Partial Reconfigurable Accelerator

PR Partial Reconfiguration

SMP Symmetric Multiprocessing

SW Abbreviation for Software.

UDouble mapping two copies of the DFG pattern(s) on the overlay tiles.

UOB URUK Operator Based.

UPB URUK Pattern Based.

UQuad mapping four copies of the DFG pattern(s) on the overlay tiles.

URUK An ancient city of Sumer in southern Mesopotamia, where the author of this work was born.

USingle mapping data flow graph's pattern(s) on the overlay tiles without doubling.

VADD Vector Addition.

VMUL Vector Multiplication.

VAM Virtual Accelerator Machine

XDC Xilinx Design Constraint

Chapter 1

Interoduction

Two trends are driving the pursuit of next generation computer architectures for data centers. The first trend is not new; the size, complexity, and diversity of the software applications running across distributed nodes as well as the data sets processed by these applications continues to increase. A little over a decade ago, our semiconductor industry switched our fundamental processing capabilities from scalar processors to manycores, or chips with multiple cores to meet these growing demands. Manycores can support scalable program concurrency through increasing the number of processor cores that can be fabricated within a chip.

The second trend is being driven by growing concerns over the relatively inefficient levels of energy efficiency achieved by todays computer systems. The CRA working group report entitled "Revitalizing Computer Architecture Research for Next Generation Systems" called this out as a grand challenge problem for their "System 2020 Vision". They put forth the challenge of creating a new featherweight supercomputer architecture that can achieve 0.001 nJ/op [22]. This is four orders of magnitude improvement over today's systems.

While the switch from scalar processors to manycores promised a more scalable solution for next generation data centers, it turned out to be no panacea when viewed through the lens of energy efficiency. Simply stated, Dennard scaling ended. Informally Dennard scaling states that as feature sizes of transistors are shrunk, the associated voltage and current scale down proportionally. Effectively under Dennard scaling the power would remain constant for a constant area of silicon. The ending of Dennard scaling posed immediate problems for the success of the newly evolving manycore era. The end of Dennard scaling resulted in a phenomena that became known as Dark Silicon [26], or the inability to turn on available cores due to energy limitations as well as lack of available concurrency within the program.

To date, there has been no definitive answer on how to eliminate Dark Silicon, or field an architecture that can reach an energy efficiency of 0.001 nJ/op. What has occurred has been a fairly rapid evolutionary change to mixes and types of traditional processors that are built into the manycore chips. Providing systems with mixed types of processors can increase energy efficiency by increasing the use of the transistors that we can turn on. Manycores with mixes of processor types are referred to as heterogeneous manycores. By including heterogeneous components the data center can exploit a broader range of parallelism and at multiple levels of granularity. Modern heterogeneous systems include general purpose processors to exploit to support multithreading, as well as Graphics Processor Units (GPUs) to exploit data parallelism. This richer set of resources is an improvement compared to the initial manycore chips, which simply replicated a standard, or homogenous, general purpose processor.

However heterogeneous systems are limited in their ability to exploit all available types of parallelism and achieve new levels of energy efficiency. The problem is that modern workloads contain what can be referred to irregular types of parallelism that cannot be efficiently computed by fixed general purpose and data parallel types of processors. Custom hardware accelerators in the form of Application Specific Integrated Circuits (ASIC's) can be created to tailor transistors and wires to better match the data and control flow patterns of the irregular parallelism. This can result in better energy efficiency per a given set of transistor and wire resources. However, these circuits require long system development cycles and exhaustive pre and post silicon verification procedures. Moreover, the very fact that ASICs are tailored for specific applications limit their flexibility and reusability compared to more general-purpose processors. ASICs can cost upwards of $ 50 M to design, and the development and test times are too long to attempt redesigns at anything close to the rate that the algorithms and application programs are modified.

The concept of custom accelerators is still being pursued by our semiconductor industry, but using Field Programmable Gate Arrays (FPGAs) in place of ASICs. FPGAs are not as dense or fast as an ASIC but they do provide acceptable levels of performance increases through customization,

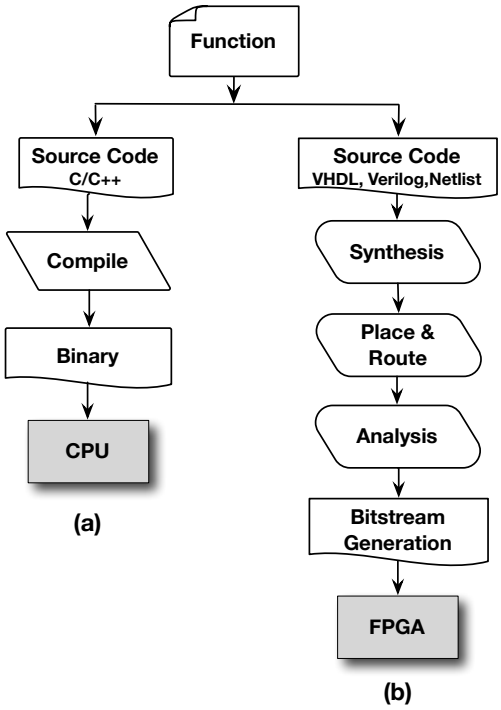


Figure 1.1: General Processors vs. FPGAs Compilation Flows.

while still providing higher flexibility and much shorter design times compared to ASICs. Our semiconductor industry is willing to trade some peak performance for design flexibility and increased developer productivity. This tradeoff was highlighted when Microsoft revealed Catapult, a prototype server with FPGAs to accelerate their Bing search engine [53]. Following the disclosure of Catapult, Intel announced acquisition of Altera, one of two major FPGA vendors, and created HARP, a new compute node that will integrate an FPGA. This trend is continuing. Recently, Amazon announced the EC2 F1 which is a compute instance with FPGAs that allow programmers to create custom hardware accelerators for their applications.

Unfortunately the state of the art in design and development tools for FPGAs are not sufficiently mature to allow application programmers that comprise the workforce in the data center to easily write, compile and run code on the FPGAs. Enabling FPGAs to be part of the solution for building energy efficient next generation systems will require successfully resolving two long standing research challenges that have so far prevented reconfigurable computing from becoming mainstream.

1.0.1 Compilation Challenges

The compilation flow for FPGAs is different from the traditional compilation for general processors as shown in the Figure1.1. The design flow for FPGAs still require bit level manipulation during design time using synthesis, place and route which are time consuming processes. These processes need to be repeated again and again with every changes in the source code or design constraints. It may take from several minutes to several hours depends on the design size. This produces a large compilation gab between FPGAs and general processors. Despite the great enhancement in the CAD tools by using High Level Synthesis (HLS), the design has to go through the traditional FPGAs compilation phases, which are synthesis, place and route.

Thus a large productivity gap exists between programming gates within FPGAs and generating executables for general processors. This is true despite the great advancement provided by High Level Synthesis (HLS), tools. These tools allow functionality to be expressed in C like programming languages instead of traditional Hardware Description Languages (HDLs) such as Verilog or VHDL. While this allows programmers to express applications in a form closer in look and feel to the languages used to program processors, the designs still must still be created in hardware centric CAD tools, still require knowledge of digital design, and ultimately pass through the inefficient and time consuming steps of synthesis, place and route.

To bridge the compilation gab, other options has been suggested such as constructing an intermediate fabric (Overlay) on top of the FPGA resources to raise the level of abstractions and route in word level rather than in bit level. Theoretically, this is one of the viable solutions to redeem the gab and reduce the compilation time for FPGAs. However, overlays can be designed in many different ways to virtualize the original FPGAs resources. The costs and benefits can vary for each way. In this work, we are exploring the advantages and disadvantages of each way of design that is included in this study. Also, this study shows the tradeoffs among performance, resource utilization, and compilation complexity.

1.0.2 Technical Challenges

The current state-of-the-art programming languages, design abstractions and design flows used for programming FPGAs evolved from VLSI chip design and require hardware development skills. FPGA designers must still learn hardware programming models and digital design. This is a concern as the United States Bureau of Labor Statistics reported in 2015 that the United States employed approximately 85,000 hardware engineers compared to 1.3M software programmers [31]. The reconfigurable computing community put great effort to increase productivity and bring software developers to program FPGAs using High Level Synthesis(HLS). High Level Synthesis has been considered as a robust compilation technology to increase productivity. The HLS provides more familiar syntax to software programmers by using Clike languages such as SystemC, HandleC, CC++,.. etc. However, it failed at getting programmers to use FPGAs because it,also, requires hardware background knowledge such as the type of interface between hardware components, timing analysis, physical constraints, resource utilization,.. etc. Moreover, the design still has to go through synthesis and place and route.

If FPGAs are to become part of the infrastructure for data center and warehouse scale computers, the large cadre software programmers must be given access to these devices through their accepted practices and tools. This requires giving them a path to programming FPGAs that bypasses CAD tools, the need to understand hardware programming models, and synthesizing, placing and routing each new design. The next section briefly describes a recent proposed idea to move the synthesis process out of the way of programmers.

1.1 Just-In-Time Assembly

Recently Just-in-Time (JIT) techniques have been proposed for assembling pre-built circuits at runtime within FPGAs [43], [45], and [44]. The idea is that pre-synthesized parallel patterns such as map, reduce, foreach, filter..etc can be made available within libraries and then placed

into the FPGA by a runtime interpreter. JIT holds promise for effectively moving the synthesis process out of a programmers path and allowing hardware circuits to be compiled and interpreted. JIT techniques have been applied to programming predefined overlay components such as ALUs as well as moving bitstreams into and out of partial reconfiguration regions. While this approach allows programmers to compile accelerators it suffers from the following drawbacks:

- *All Variants of Programming Patterns Must be Synthesized*.
- *Cannot Compose Simple Conditionals with Pre-synthesized Programming Patterns*.

1.2 Thesis Statement

Enabling software developers to apply their skills over FPGAs continues to be an unreached research objective in the reconfigurable computing community. JIT Assembly holds promise for effectively moving the synthesis process out of a programmers path and allowing hardware circuits to be compiled and interpreted. To extend the JIT Assembly approach and support moving the synthesis process out of the way of programmers, I am proposing Reconfigurable Overlay,URUK, which composes fundamental computational operators instead of full patterns . This thesis evaluates the new JIT Assembly by considering the following questions:

- Can URUK eliminate the challenges that result from composing pre-synthesized parallel patterns while still preserving all the productivity benefits of the original JIT approach?
- Can URUK allow conditionals to be composed with the synthesized programming patterns without generating multiple bitstreams for each case?
- How much time does it take to construct an accelerator using the new compilation flow targeting URUK compared to Vivado HLS?
- How will performance and resource utilization be affected compared to full custom designed modules using Vivado HLS as well as the original JIT approach?

- What are the costs and benefits of considering Partial Reconfiguration techniques as part of the overlay dynamic system?

1.2.1 Thesis Contributions and Organization

Throughout the exploration of this work, I have made the following set of contributions and published them in top-tier conferences.

- *A Dynamic Overlay to Support Just-In-Time Assembly:* In this work, a dynamic overlay is designed to support Just-In-Time assembly by composing hardware operators to construct full accelerators. The hardware operators are pre-synthesized bitstreams and can be downloaded to Partially Reconfigurable(PR) regions at runtime [3].

- *A Run-time Interpretation Approach For Creating Custom Accelerators:* We provided a new approach in which hardware accelerators can be built and run using compilation and run time interpretation. Also, we demonstrated that our approach can enable software programmers without any hardware skills to create hardware accelerators at productivity levels consistent with software development and compilation [43].

- *Composing Pre-Synthesized Building Blocks at Run-Time:* We demonstrated a technique to move synthesis out of the programmers path by composing pre-synthesized building blocks using a domain-specific language that supports programming patterns tailored to FPGA accelerators. Our results show that the achieved performance of run time assembling accelerators is equivalent to synthesizing a custom block of hardware using automated HLS tools [44].

- *Just-In-Time Assembly of Custom Accelerators:* We demonstrated that Synthesis can be eliminated from the application programmers path by becoming part of the initial coding process when creating the programming patterns that define a Domain Specific Language. Programmers see no difference between creating software or hardware functionality when

using the DSL. A run time interpreter is introduced that assembles hardware accelerators within a configurable tile array of partially reconfigurable slots at run time [45].

- *A Flexible Multilayer Perceptron Co-processor for FPGAs:* We designed a Multilayer Perceptron Co-processor (MLPCP) targeting FPGAs that is configurable during design time and programmable during runtime. The MLPCP can be reprogrammed at run time to rapidly change network topologies and use different activation functions. It promotes design reusability and allows application developers to change parameters of a given network without the need to resynthesize [2].

The rest of this thesis is organized as follows. Chapter 2 gives background on fine-grind reconfigurable architectures, and Course-Grind Reconfigurable Accelerators as well as providing a survey on the start-of-the-art approaches on constructing intermediate fabrics, overlays, FPGA virtualizations and programming modules. Chapter 3 provides background on the original Just-In-Time Assembly (JITA) of custom accelerators. Next, Chapter 4 presents the proposed solution as well as the new overlay architecture including the overlay instruction sets. Chapter 5 provides a guideline for the compilation process when targeting URUK overlay. Chapter 6 describes the evaluation methods and dissusses the results. Finally, Chapter 7 gives answers to the thesis questions and potential future work.

Chapter 2

Background

This chapter provides an overview of reconfigurable hardware architectures including Field Programmable Gate Arrays (FPGAs) and Coarse-Grained Reconfigurable Architectures(CGRAs). Additionally, the compilation flow and the productivity challenges of FPGAs are discussed. Then, it presents the effort of the reconfigurable computing community on raising the design abstraction level and increasing productivity. Moreover, the start-of-the-art approaches on constructing intermediate fabrics, overlays, and programming modules are presented.

2.1 FPGAs Overview

Field-Programmable Gate Arrays (FPGAs) are electrically programmable silicon devices that can be configured to implement almost any complex digital circuits or systems. An FPGA is a two-dimensional array of logic units and electrically programmable routing interconnects [34]. Logic units comprise Configurable Logic Blocks(CLBs), Digital Signal Processors (DSPs), Block RAMs (BRAMs), Input-Output Buffers (IOBs), and Digital Clock Managers (DCMs). These logic units can be configured and connected to implement different combinations of sequential and combinational circuits to provide different functionalities ranging from one simple gate to a sophisticated microprocessor. CLBs include a number of Lookup Tables (LUTs) that can be programmed to implement any boolean expression. The routing interconnects consist of variant length wire segments and electrically programmable switches, which can switch on and off the connection between logic units in bit level. Figure 2.1 shows a general example of FPGAs which includes a two-dimensional grid of CLBs surrounded by I/O blocks. The grid is wired and connected by programmable switches. The "programmable/reconfigurable" term in FPGAs refers to the capability of forming a new digital circuit on a chip after fabrication by programming the interconnect

switches and changing the behavior of the logic units [34].

Routing interconnect makes up 90 percent of the total area of FPGAs[60]. In consideration of that FPGAs architecture meant to be general and capable of implementing any digital circuit, the routing interconnect must be very flexible to adapt to a large variety of circuits. During the design phase, Computer Aided Design (CAD) tools search for an optimal solution to place specific logic into specific configurable units and wire them at bit level. With every possible place and route solution, the CAD tools evaluate design constraints to meet the specifications. For instance, the tools perform timing analysis for every place and route to meet the timing constraints, and if the design violates timing, the tools will seek for another possible solution for place and route. These processes consume a large amount of time, which ranges from several minutes to days depending on the design size.

Typically, FPGAs are programmed with Hardware Description Languages (HDLs) such as VHDL and Verilog. The traditional FPGAs' design flow includes four stages as follows. First, designers write their code in VHDL or Verilog and set timing and I/O constraints of the design. Second, CAD tools apply logic synthesis, which translates a source description code written in an HDL into a set of Boolean gates and Flip-Flops. Then, synthesized design is passed to the implementation stage which includes place and route processes. The place and route are the heaviest tasks in the whole design flow because tools are looking for the best solution in a large search space. After the implementation, bitstreams can be generated which will be downloaded to the target FPGA.

To enable programmers to access FPGAs and increase their productivity, we need to eliminate two major challenges: First, we need to allow programmers to write their applications in high level languages. Second, we should move the synthesis, place and route out of the programmers' way. The next sections present the effort of the reconfigurable computing community to phase out these challenges.

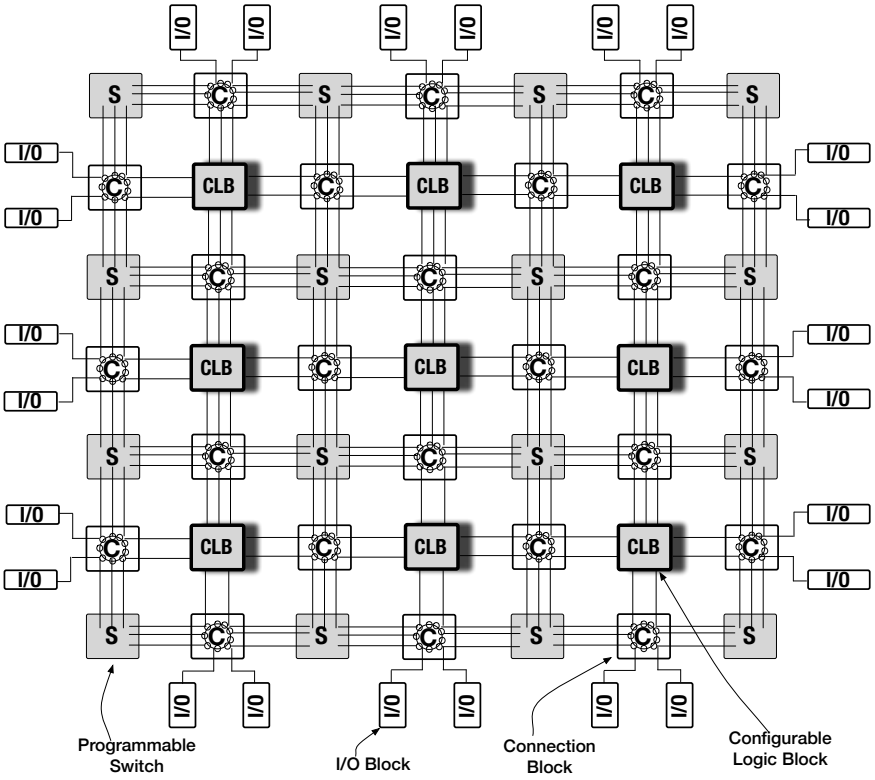


Figure 2.1: General FPGA Architecture.

2.2 CGRAs Overview

Coarse-Grained Reconfigurable Architectures(CGRAs) have been proposed as an alternative to the fine-grain architectures (FPGAs) to support faster compilation through raising the level of reconfigurability from bit-width to word-width granularity, which enables on-the-fly customization and reduces configuration overload. Particularly, CGRAs are designed to be customized on ASIC for specific applications that have inherent data-parallelism. CGRAs are mainly composed of Processing Elements(PEs), that include ALUs, multipliers, and shift registers connected by word width mesh-like interconnects and are controlled by resources managers and synchronization modules. Figure 2.2 shows a sample of a CGRA architecture. CGRAs can be either tightly coupled(e.g, Matrix[52], Chess[46], and DySER[30]), or loosely coupled (e.g. MorphoSys[56], CHARM[19], and PipeRench[29]). Loosely coupled CGRAs are more independent from the host CPU, and they have their own control flow.

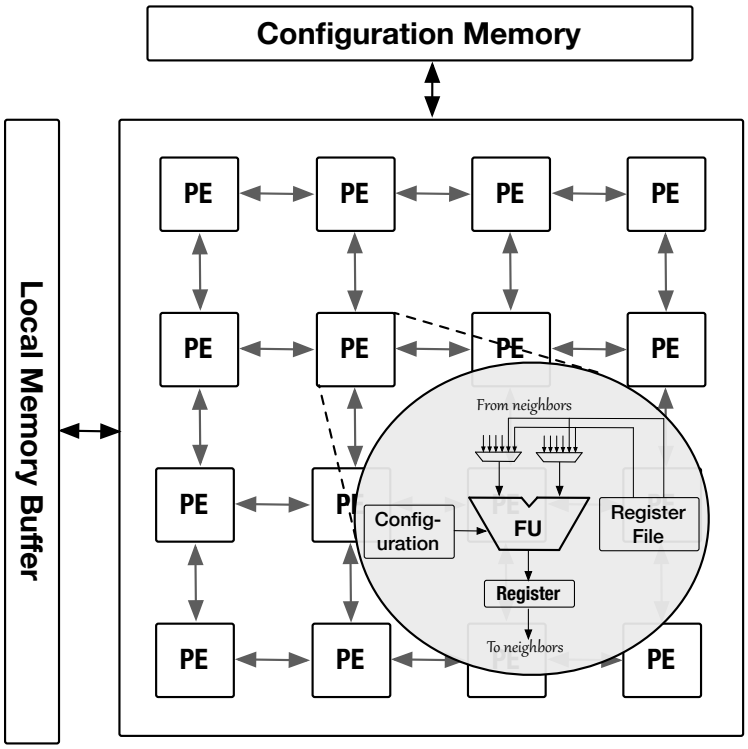


Figure 2.2: General CGRA Architecture.

CGRAs can be connected in a linear array (1D)based architecture such as RaPid[25] and PipeRench[29] or in 2D mesh interconnect such as Matrix[52] and MorphoSys[56]; The 1D architecture is suitable for computations that can be linearly pipelined. In contrast, it is not efficient to support block-based applications [62]. On the other hand, the 2D mesh architectures have a better support to block-based processing such as in multimedia applications.

CGRAs are typically configured using opcodes instead of Hardware Description Languages (HDLs), which leads to reducing compilation time through eliminating the extensive place and route processes. CGRAs instructions can be stored in a centralized or distributed instructions memory(s) [39]. Each computation unit interprets the assigned instructions and selects the right hardware operator for the specified operation. ALUs within CGRAs should have the hardware components for all supported operators on fabrics. Only selected components will be used while others occupy area while staying idle. CGRAs, in general, are limited to simple operations such as add, sub, mul, and logical operations. In order to support more computational operations, PEs will

become larger and may reach to the complexity of a processor.

Furthermore, CGRAs are limited to accelerating simple operations in nested loops. Most CGRAs do not support conditional operations and branching. Additionally, they lack the flexibility that FPGAs provide regarding user control over low-resource definition and allocation.

2.3 High Level Synthesis

The increase in silicon capacity and System-On-Chip (SOC) complexity has shifted interest toward a higher level of abstraction which is considered on of the powerful ways of regulating complexity and enhancing design productivity [18]. The reconfigurable computing community has addressed the productivity challenges within FPGAs design flow and tried to raise the abstraction level beyond Register Transaction Level (RTL) by using High Level Synthesis (HLS). The HLS tools translate untimed or partially timed functional specifications written in one of the high level languages such as C, C++, SystemC, Haskell, ..etc into low level fully timed RTL specifications. That leads to not only providing more familiar syntax to software programmers but also reducing the code density. For example, the conducted study from NEC[68] shows that large designs within around 300k lines of RTL code can be reduced by 7X-10X when using high abstractions. Further, the HLS flow reduces the required time for creating hardware as well as reducing verification time and facilitates resource utilizations and power analysis. Currently, several commercial HLS tools are available such as Vivado HLS, LegUp[14], and ROCCC[61].

In general, HLS has faced many challenges in translating source codes written in sequential languages into hardware description specifications. For instance, bit accuracy, timing, concurrency, and synchronization are not supported explicitly in Standard C/C++; while they are essential in hardware design [18]. On the other hand, pointers, recursion, polymorphism, and memory management are complex C/C++ constructs, which cannot be transformed easily into hardware. To overcome these challenges, several solutions have been applied, such as disallowing the use of dynamic constructs, using compiler directives and pragmas to explicitly specify concurrency, and

introducing hardware-oriented language extensions and libraries. HardwareC[33], SpecC[27], and HandelC are examples for hardware language extensions, and SystemC is an instance for hardware libraries.

Despite the great accomplishments in High Level Synthesis, software programmers still obligated to gain knowledge in hardware low level details to create efficient designs. In order to create an optimized design, programmers/designers should explicitly specify pragmas and directives (Vivado HLS), clock boundaries(Handel-C), and clock edges/events (SpecC, SystemC) in their source code. For example, in Vivado HLS, the user is required to understand which loop to unroll, when to pipeline operations, what suitable interfaces for the inputs and the outputs, and the tradeoffs among power, performance, and resources utilization. Furthermore, even through the HLS has reduced the hardware design time, the compilation time is still significant. In fact, the design still has to go through the extensive processes,which are synthesis, place and route.

The FPGAs research community has taken another path to reduce compilation time and increase productivity by using overlays, which are discussed in the next section.

2.4 Overlays

Intermediate Fabrics, or overlays, have been proposed to allow higher level computational components such as soft processors, and vector processors [67, 66], as well as Course Grained Reconfigurable Arrays(CGRAs) type structures [20, 21, 16], to be embedded within FPGAs. The potential advantage of such overlays is that circuits and hardware acceleration can be achieved through compilation instead of synthesis on existing FPGAs. Conventional approaches for enabling CGRAs on an FPGA are to replace LUTs and Flip Flops with small programmable computational units like ALUs as the compilation target. The ALUs are embedded within a network of switch boxes and channels. The computational units can be populated with programmable functions at a rough equivalence in circuit density to Medium Scale Integrated (MSI) components. Arithmetic and logical operations, as well as shift registers, and multipliers have been proposed. These units then

serve as the target for compilers to exploit loop level parallelism. The interconnect structures are defined to support wider word widths instead of bit level interconnections. Typically, overlays introduces some overhead inefficiencies due to the additional resources, routing delays between computational units, and limitations on the granularity of parallelism. Over the past years, several approaches for virtualizing FPGA resources and building overlays have been published. The next sections present some overlay projects proposed by the reconfigurable computing community.

2.4.1 JIT FPGA

The JIT FPGA approach presents a virtual FPGA, a synthesizable firm-core described in structural VHDL. It enables the development of standard hardware binaries, which provide portability among different FPGAs. The virtual FPGA is a fine-grained fabric with virtualized LUTs. The structure of one virtual LUT requires 100 physical FPGA LUTs. The JIT compiler for FPGAs begins with the standard hardware binary and performs mapping to place the hardware logic onto the virtual LUTs, then implements place and route. The virtual FPGA can achieve portability and programmability at the cost of area and performance, around 100x area overhead and 6x slow down in speed respectively [9, 40, 41, 42].

2.4.2 Virtual FPGAs

FPGAs' virtualization can be accomplished by constructing a coarse-grained architecture, which utilizes the low level hardware of FPGAs and provides high level of programmability. Metzner [51] presented a coarse-grained architecture that enables run-time dynamic hardware multithreading. The architecture includes computational elements connected in a 2D mesh network, which can route data among them. Virtualization also can be achieved through partial reconfiguration techniques as in OpenStack [13], which allows users to "boot" pre-designed custom hardware accelerators in a similar way of Virtual Machines.

2.4.3 ZUMA:

An open source embedded FPGA architecture constructs an overlay on top of existing FPGA resources and intends to achieve portability of designs and bitstreams among different vendors (Xilinx, Altera) and parts. It follows the philosophy of virtual machines in computing environments. The ZUMA architecture attains virtualization by taking advantage of reprogrammability of LUTRAMs provided in modern Xilinx and Altera FPGAs also forming a new programmable LUTs and routing MUXs. It requires a bout 40 physical FPGA LUTs to create one virtual ZUMA LUT. That puts around 40x area overhead and consumes more resources which basically increases power consumption [10, 63].

2.4.4 Intermediate Fabrics

Intermediate Fabrics(IF) provides a virtual intermediate layer between the underlying physical FPGA resources and user designs. The IF structure is nearly identical to conventional FPGAs structures; this is represented by distributed computational unites in a grid across the fabric, with switch boxes, connection boxes, and tracks. Similarly to FPGAs, IF is programmed using a configurable bitstream. On the contrary, the IF resources are not as general as the FPGA resources. In fact, computational unites and routing resources are specialized for a particular domains or applications. The specialization of the IF approach makes it effective when the application is matching an already pre-built fabric. A new fabric should be implemented if there is no match between the application and the current available fabrics. Additionally, the area overhead incurred by virtual fabrics is significant [20, 21].

2.4.5 Synthesis-Free JIT Compilation:

Synthesis-Free JIT Compilation [15] is another project that explored the feasibility of translating hot straight lines of code into Virtual Dynamically Reconfigurable(VDR) overlay, which consists

of an array of functional units that are interconnected by a set of programmable switches. The approach uses traces to capture the line of codes that are going to be executed frequently, and transforms them into Data Flow Graphs(DFG), which are then mapped to the VDR overlay. Functional units in the VDR are also limited to basic operations (e.g. addition, subtraction, and multiplication).

2.4.6 SCGRA

An FPGA based CGRA called Soft Coarse-Grained Reconfigurable Array(SCGRA) [38] is an intermediate compilation step that replaces compiling high-level applications directly to circuits into scheduling operational tasks targeting SCGRA overlay. The approach aims to promote design productivity. SCGRA focused on the hardware resource constrains, IO bandwidth constrains and the loop parallelism partition, whereas processing architectural design supports only simple logical and arithmetic operations, which limits the capability of processing complicated functions. Additionally, it does not support conditional branchings within loops.

2.4.7 QuickDough

Presents a design framework for constructing loop accelerators targeting an FPGA-based Soft Coarse-Grain Reconfigurable Array (SCGRA) overlay. During compilation, QuickDough framework transforms a high level loop into a Data Flow Graph(DFG), schedules the DFG nodes to the SCGRA and estimates the communication cost, and then selects an accelerator from a pre-built bitstream library. By taking the advantage of pre-built bitstreams, the framework aims to translate C-nested loops into hardware circuits supporting quick compilation for a hybrid CPU-FPGA system. In the same way, the work focuses on the automatic customization of the overlay hardware parameters, loop unrollment factors, and buffer sizing as well as hardware-software communication. Their results show that with the cost of 10 to 20 minutes in compilation overhead spent in customization, the performance was improved by 5x [37].

2.4.8 QUKU:

QUKU is a coarse-grained reconfigurable PE array (CGRA) overlaid on an FPGA. The main goal of this overlay is to reduce the reconfiguration time and increase the accessibility of FPGAs. By applying the model at an architectural level in QUKU, better hardware efficiency can be achieved for a wide domain of applications. A few widely used DSP algorithms have been presented to demonstrate the application of process network models to architectural template generation in QUKU [54, 55].

2.4.9 Soft Processors

Soft processors are considered one of the overlay forms that support fast application compilation. In addition to the commercial cores such as MicroBlaze form Xilinx [65] and Nios II from Altera [4], some other soft processors are provided by different research groups. For instance, an open source soft processor [32] is provided with RISC-V instruction set. It is a 4-stage pipeline, tightly-coupled architecture with FPGA accelerators. The processor is portable between FPGA platforms and can be synthesized to run at maximum frequency, 268.67 MHz.

Another tightly-coupled VLIW processor with a course-grind reconfigurable matrix called Architecture for Dynamically Reconfigurable Embedded Systems (ADERS) [49] was designed to simplify hardware-software programming models, scale down communication overhead, and essentially gain sharing resources. The same research group developed a framework [48], to compile C-source code, targeting the architecture along with a scheduling algorithm to exploit loop level parallelism [47]. The integration between the processor and the reconfigurable array as well as the overall system performance was the main concern in the ADERS project.

Several other soft processors published in the academia briefed as follows: DSP Extension Architecture (iDEA), a lightweight soft processor which takes advantage of DSP48E1 primitive in Xilinx FPGAs [17]; Octavo, a highly parametrized multi-threaded processor with ten pipeline

DSL	Language Syntax	Output	Reference
FSMLang	C-style	VHDL, Verilog, C (drivers)	[1]
Chisel	Scala	Verilog , C++ (simulation)	[7]
Kiwi	C# .Net	RTL netlist	[64]
Lime	Jave	VHDL, Verilog	[6]
Delite	Scala	C, C++, Scala, OpenCL	[12]

Table 2.1: DSLs for design applications on FPGAs.

stages implemented on Stratix IV FPGA [35]; CUSTARD, a customizable multi-threaded soft processor supporting hardware threads to be implemented in dedicated hardware [23]; Anjam and others [5] also presented a VLIW soft processor with dynamically adjustable issue width and cores through utilizing the partial reconfiguration feature in Xilinx FPGAs. In general, soft processors allow software developers to directly compile their source code into FPGAs and provide reusable overlay. However, soft processors are considered mostly suitable for embedded systems due to their low frequency and sequential execution.

2.5 Domain Specific Languages

Domain Specific Languages(DSLs) (e.g. Python, Snort, HTML) are common within software development. DSLs promote the use of languages tuned for the needs of specific application domains. Once created and tuned, the language promotes increased programmer productivity through appropriate abstractions and heavy reuse. DSLs are also being considered to generate accelerators within FPGAs. Table 2.1 summarizes the currently used DSLs for building accelerators on FPGAs.

2.5.1 FSMLang:

FSMLanguage is a domain-specific language (DSL) for describing finite-state machines. The language was developed in order to create a way for programmers to develop reusable representations of FSMs. The FSMLanguage compiler is capable of producing both software and hardware imple-

mentations of FSMLanguage programs. Both implementation types remain compatible with one another as the communication abstractions that are built in to FSMLanguage are able to cross the hardware/software boundary [1]. The language structure allows one to easily describe FSMs in a way that eliminates many of the common errors that occur when describing FSMs in typical HDLs. The FSMLanugage compiler automatically generates correct code for FSM reset, sensitivity lists, memory access schemes, FSM flip-flops, and state transitions. The abstractions for memories and channels allow programmers to use familiar, software-like constructs for describing timing and synchronization sensitive operations. Additionally, these abstractions are reusable and can be used in both software or hardware implementations of FSMLanguage programs.

2.5.2 Chisel:

To design more flexible hardware units, Chisel, a new hardware construction language that supports advanced hardware design, has been created. The goal of Chisel is to allow a designer to provide a procedural description of how the hardware should be instantiated, given a set of parameters that are fixed at the prototyping phase of the design. By embedding with the Scala programming language, Chisel can raise the level of hardware design abstraction by providing concepts including object orientation, function programming, parameterized types, and type inference[7]. Chisel can reduce the programming challenges through a high-speed C++ based cycle-accurate software simulator as well as low-level Verilog, which is designed to be mapped either to FPGAs or to standard ASIC flow for synthesis.

2.5.3 Aspen:

Abstract Scalable Performance Engineering Notation (Aspen) fills a gap between existing performance modeling techniques and rapid exploration of new algorithms and architectures [57]. In particular, both formal specification of application and an abstract machine model are need to analysis the performance behavior. In Aspen language, the modularity of performance can be achieved

by balancing the workload with the overall performance characteristics of main kernels [50]. However, only the control flow and data flow are expressed at a function or module level and the behavior is input-dependent and implementation-specific. Thus, Aspen is not able to analyze auto-tuning and projecting expected performance.

2.5.4 Lime:

Lime, developed at IBM Research, is a Java-compatible object-oriented language which targets heterogeneous systems with general purpose processors, FPGAs, and GPUs. Java bytecode can be generated by the Lime compiler, which allows a programmer to design a suitable Java program into a pattern amenable for heterogeneous parallel devices, and the program can run on any platform that supports a Java virtual machine. The compiler can also synthesize and then generate hardware designs for FPGAs. The Lime language exposes parallelism and computation explicitly with high-level abstractions. Streaming computation as well as vector operators are virtualized using some extra abstractions. Although the paradigms provide a very high-level abstraction, optimizations are limited due to the initiative point is focus on lower-level byte-code [24].

2.5.5 Delite:

Delite was essentially created as an open source compiler framework to build substantial concurrency languages for use on heterogeneous multiprocessor systems. Delite simplifies the definition and construction of a DSL language and includes the generation of the compiler for the new language. OptiML-a DSL for machine learning applications, OptiQL –a DSL for data querying, and OptiGraph –a DSL for graphic analytics have been created using Delite and are publicly available. Delite is built in a modular fashion to allow the insertion of unique domain specific optimizations to be included into the compiler flow. All DSLs then take advantage of the built in traditional lower level instruction optimizations, such as common subexpression elimination, loop fusion, etc. Delite's modular structure allows new backends to be easily added [12].

Current backends produce Scala, C++, and CUDA. Delite can be downloaded at http://stanford-ppl.github.io/Delite/index.html.

Chapter 3

Just-In-Time Assembly

3.1 Introduction

The earlier work by Sen Ma, Just-In-Time Assembly (JITA) [43, 45, 44], aims to move the synthesis process out of the programmers' path, increase application developers productivity, and support design portability between different FPGA vendors and parts. JITA approach takes advantage of partial reconfiguration technology to compose custom accelerators on the fly by using pre-built bitstreams. Since this work is extending the JITA approach, this chapter is dedicated to provide background description about the JITA.

3.2 JITA Approach

The JITA approach aims to increase the FPGAs' programmers productivity by moving the synthesis, place and route processes out of their path through composing pre-synthesized small bitstreams to form full custom accelerators. Under traditional FPGA design flows the programming patterns are combined into a single object, and then object is synthesized. Each time the functionality in the source code is changed to create a new object, it must be re-synthesized. This keeps synthesis, place and route in the development path of the programmer. The JITA approach differs in that individual programming patterns are synthesized at the same time they are coded as part of creating a Domain Specific Language (DSL). Software prototypes for each programming patterns are placed in a library and made available to the application programmers. This allows programmers to work with the software prototypes within a DSL without having to repeat synthesis.

To achieve the goal, the JITA approach provides a prototype system that includes an overlay to virtualize the FPGA resources, a new compilation flow, and a runtime interpretation.

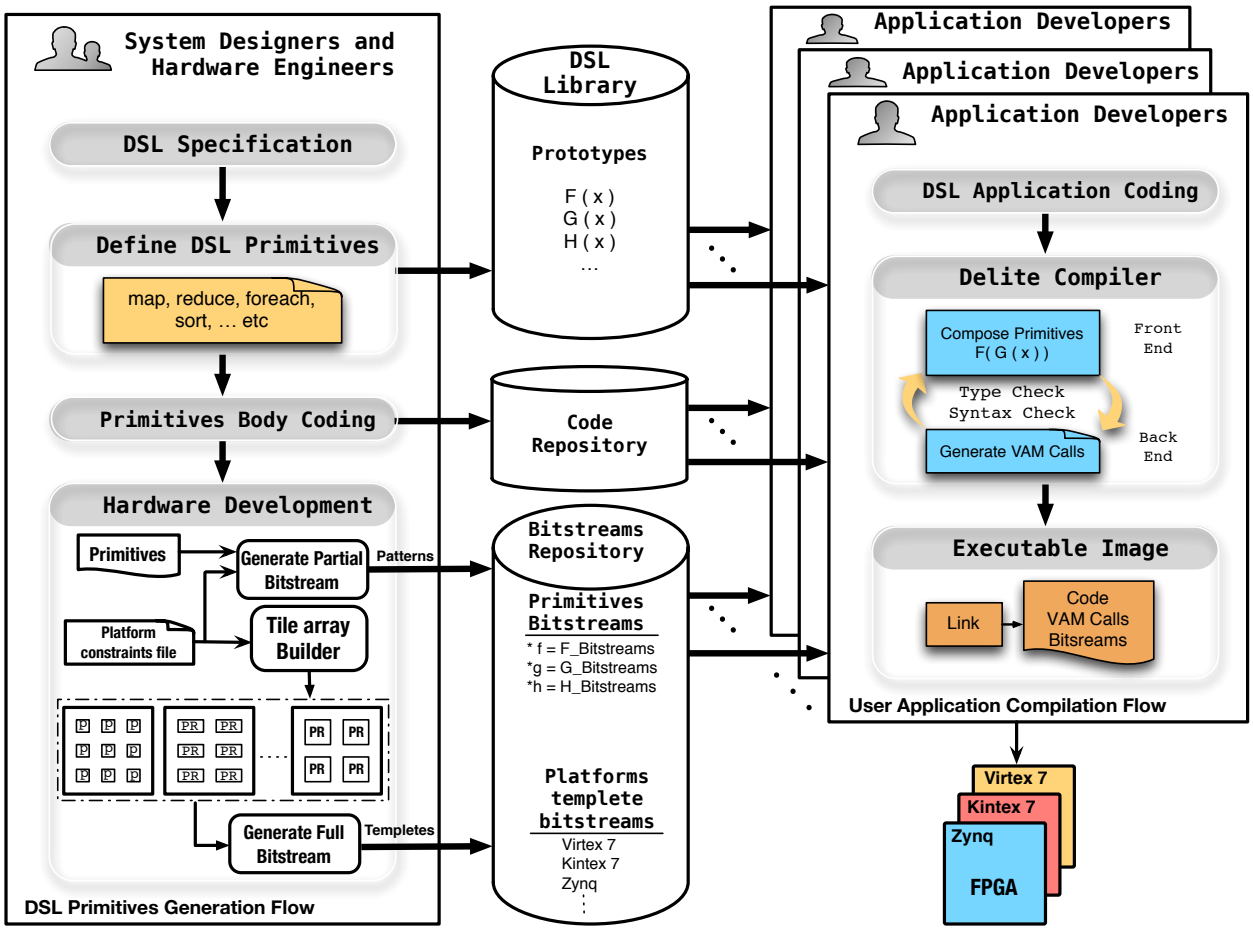


Figure 3.1: Design Approach.

3.3 Compilation Flow

The JITA uses the Delite Framework to create a new DSL. This is then compiled into for the overlay. Figure 3.1 shows how Delite supports two classes of users; system programmers and hardware engineers that create the DSL use the flow on the left, and application programmers use the flow on the right. Creating a standard DSL on the left involves defining and coding the programming patterns. The body of each primitive defined for the DSL is passed through an HLS generator to produce bitstreams. The modularity of the flow allows any HLS generator to be plugged in. The Vivado HLS tools were used for this step. Prototypes for the DSL primitives as well as their bitstream representations are placed in a library. Application programmers pull

prototypes from the library just as if the DSL was created for traditional software implementation.

After the application programmer forms the complete accelerator functionality, they invoke the Delite compiler with the ”-vam_calls” flag set. This invokes the JITA backend generator to produce a series of interpreter instructions that represent data and control flow information that will be used by a run time interpreter to build and control the accelerator at run time. These instructions, called Virtual Accelerator Machine (VAM) language are shown in Table 3.1. Figure 3.2 shows the interpreter instructions (pseudo code) output by the compiler for creating an accelerator to compute an inner product. The inner product was created by composing the REDUCE and VMUL primitives defined within the DSL. The compiler can also be run without the ”-vam_calls” flag to generate standard software executables for running on a standard processor and during debugging.

3.4 JITA Overlay

The JITA overlay was pre-formed programmable components built on top of an FPGAs lookup tables and flip-flops. The overlay was occupied with tile array and Black RAMs. The overlay includes a nearest neighbor programmable word width interconnect similar to traditional CGRA type overlays. Different from traditional CGRA overlays, the JITA overlay exposes the lookup tables and flip flops of the FPGA as partially reconfigurable tiles instead of abstracting them into programmable computational units. This combination of pre-formed interconnects and partial reconfiguration regions allows the bitstreams for the programming patterns to be downloaded at run time into the intermediate fabric. Figure 3.3 shows the structure of the hybrid overlay. The basic structure is a 2D array of partial reconfiguration tiles and programmable switches that are connected as a nearest neighbor interconnect network.

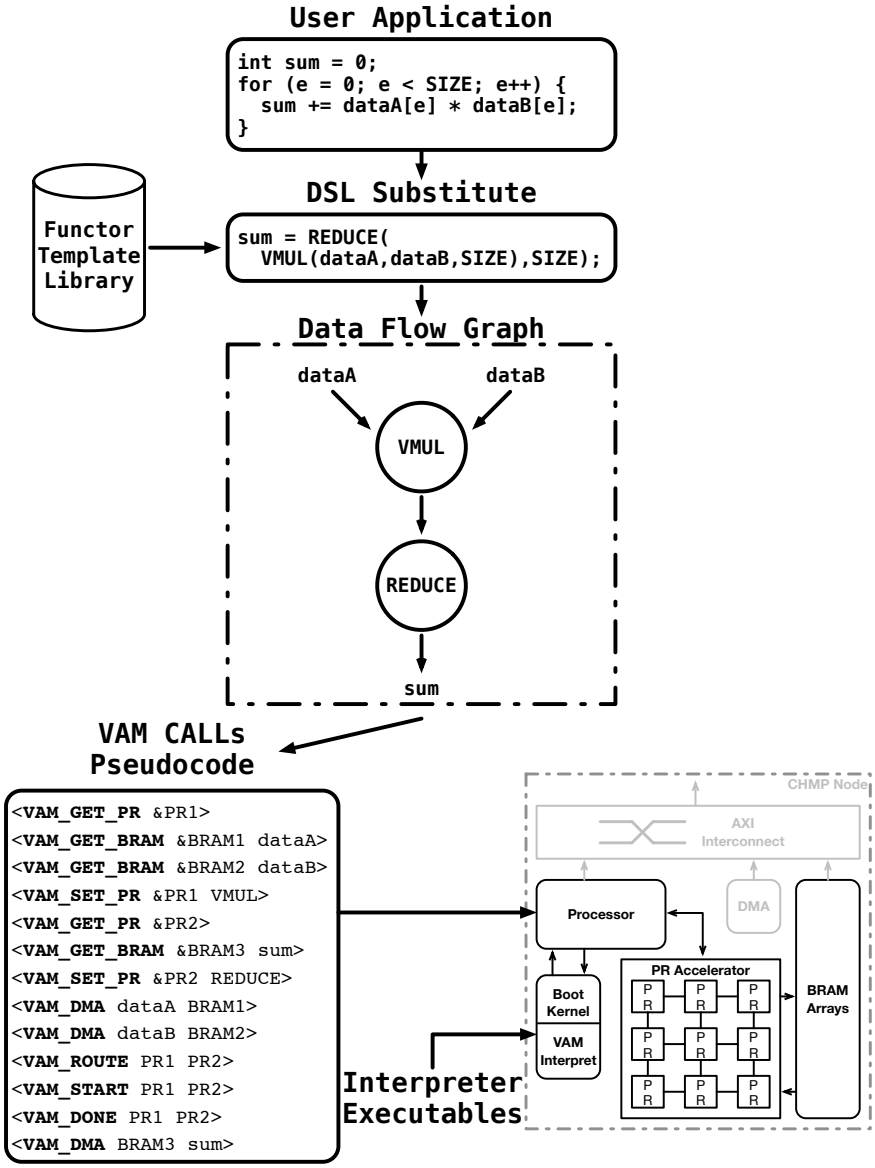


Figure 3.2: Compiler Flow and VAM Call Generation.

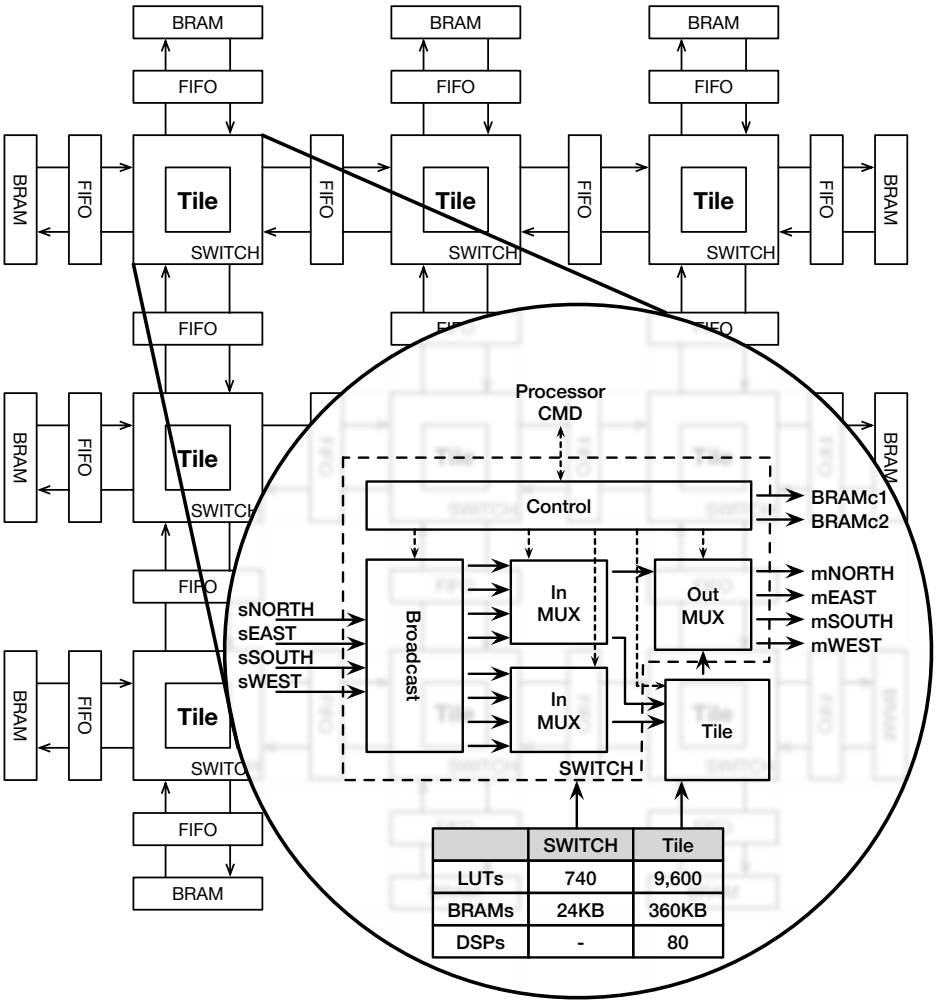


Figure 3.3: 3×3 Tile Array and Interconnect Network.

3.4.1 PR Tiles

The configurable 3×3 array, shown in Figure 3.3, was constructed of partial reconfiguration tiles sized at 9,600 LUTS, 360KB BRAM, and 80 DSPs. This PR regions was sized to hold the largest bitstream generated from the test DSL. The exact size of tiles is variable and can be set when the DSL is created. The number of the tiles is derived based on the size of the tiles and the number of resources available on a target FPGA logic family. At runtime, the PR tiles will be populated with functors' partial bitstreams which behave as computational units. By swapping different functors in the PR regions, a new accelerator will be formed. The input and output data for each PR region are controlled by tiles' programmable switches.

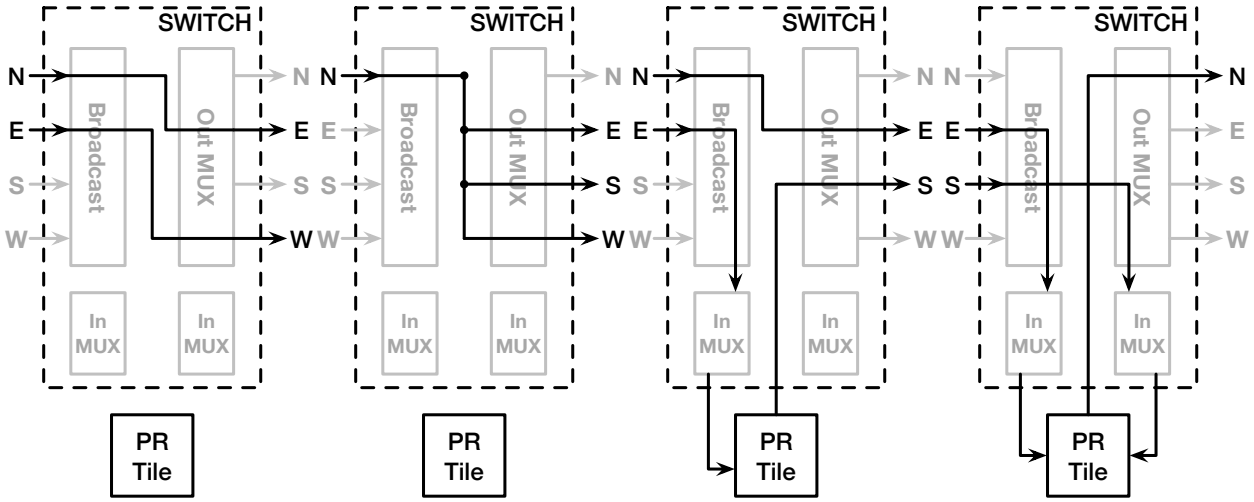


Figure 3.4: Switch Routing.

3.4.2 Programmable Switch

Figure 3.4 provides an exploded view of a switch. Figure 3.4 shows the types of routing patterns that can be programmed into switch. The routing patterns were defined to enable each switch to direct inputs and outputs through the tile, as well as serving as a pass through for routes between distant tiles.

Routes can be set statically or dynamically. Dynamic settings used for allowing the switch to support different time varying routing needs when multiple accelerators are resident within the overlay. Each switch serves as a pass through for one accelerator, and then source and synch data for a tile that is part of a different accelerator.

3.4.3 Local Memory

The boundary cells in the overlay include connections to blocks of local memories (BRAMs). These BRAMS can be used as addressable local memories or as FIFO data buffers for streaming data. Block data transfers use DMA (not shown) between the BRAMs and Global DRAM memory. The BRAMs are placed within the global address map of the system, allowing any processor or bus master device to transfer data into and out of a local memory. The BRAMS have buffer full/empty handshaking signals that are connected through the switches to enable processing to be dynamically triggered.

3.5 Run Time Interpreter

The JITA has a run time mechanism to interpret the function calls generated by the compiler to compose accelerators. Using the interpreter allows the data flow graph information generated by the back end of the DSL compiler to remain portable, similar to portable Java Byte Code. Table 3.1 shows the instructions the compiler produces to represent the data and control flow graphs. The output of the backend generator called Virtual Accelerator Machine (VAM) language. Just as a Java Virtual Machine (JVM) provides the run time mechanisms needed to implement the policies defined by the Java Byte Code, the VAM run time interpreter provides the run time executables specific to a particular organization of partially reconfigurable slots.

The interpreter allows the same output from the compiler to remain portable and used over different configurations of reconfigurable slots, and logic families. The run time interpreter, also,

provides the separation between policy and mechanism to enable the same data flow graph to be mapped and run on any configuration of reconfigurable.

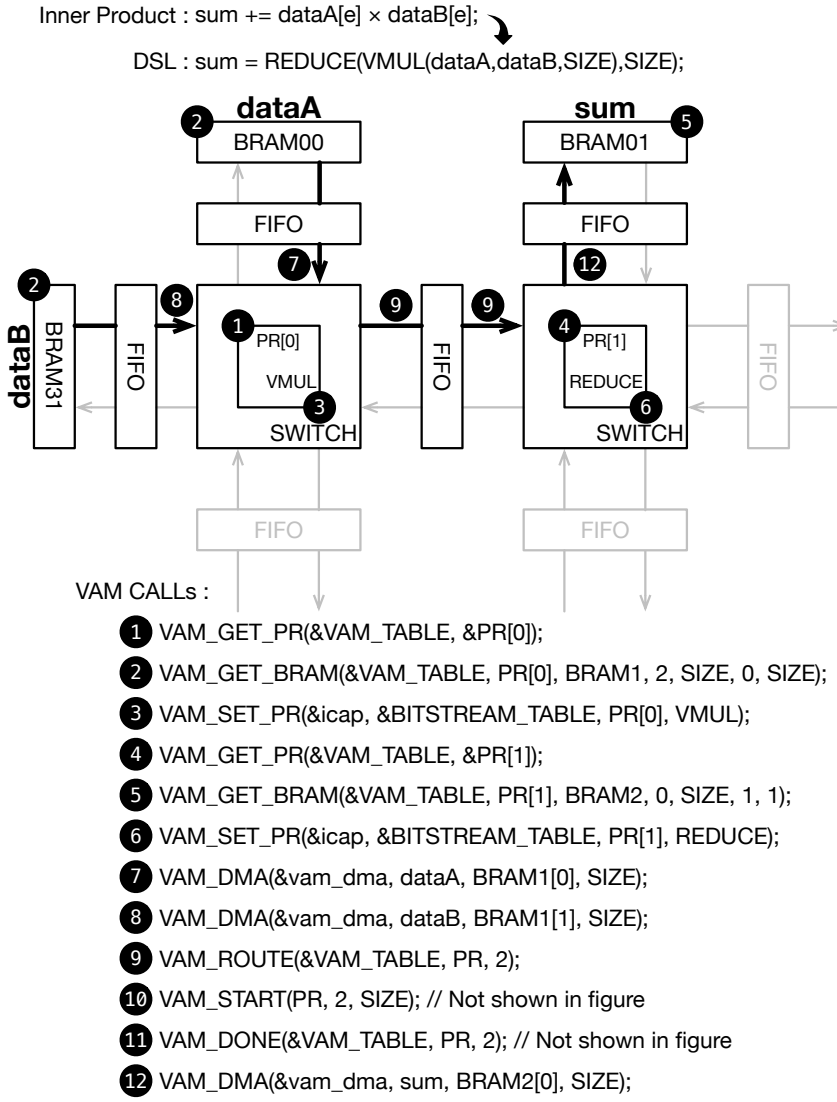


Figure 3.5: VAM Run Time Interpreter Configuration Steps.

The interpreter calls listed in Table 3.1 and shown in the example in Figure 3.2 make no commitment on how partially reconfigurable regions should be sized, configured or organized within a programmable interconnect network. As an example the *VAM_GET_PR* instruction in Figure 3.2 is a request to the run time system to get a free partially reconfigurable slot. The *VAM_SET_PR* instruction directs the run time system to load a slot with a particular bitstream, and

the *VAM_ROUTE* instruction requests the run time system to connect a data path between slots. How these commands are implemented are clearly dependent on how each FPGA is configured. The approach defines a run time interpreter to perform these platform specific operations. In this fashion the run time interpreter provides the same operation as a Java Virtual Machine with the platform independent VAM calls produced by the compiler taking the place of portable Java Byte Codes.

For example, Figure 3.7 shows how the run time interpreter constructs the accelerator shown in Figure 3.2 into the 3×3 tile array of partially reconfigurable slots shown in Figure 3.3.

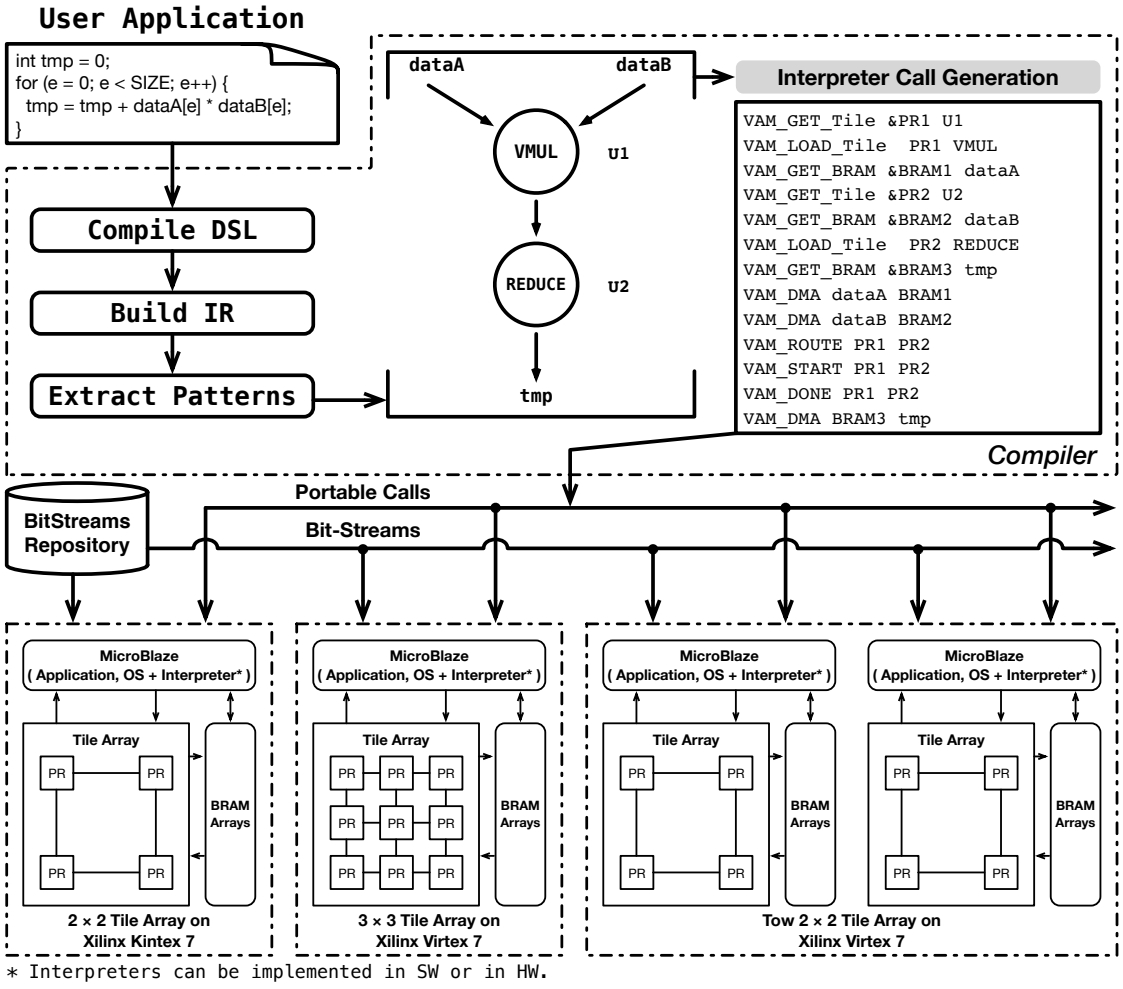


Figure 3.6: Design Portability with JIT.

Functor Placement and Loading The VAM interpreter manages the tile array by main-

Table 3.1: VAM Calls

Type	Name	Semantics	Description
PR Region Operations	VAM_GET_PR	*bool VAM_GET_PR (vam_table_t *VAM_TABLE, int *nPR);*	Requesting a free PR slot.
	VAM_GET_BRAM	*bool VAM_GET_BRAM (vam_table_t *VAM_TABLE, int nPR, u32 BRAMList, int nIN, int InSize, int nOUT, int OutSize);*	Requesting free BRAMs.
	VAM_SET_PR	*bool VAM_SET_PR (XHwIcap icap, vam_Bitstream_table_t *BITSTREAM_TABLE, int nPR, int nFunctor);*	Reconfiguring PR region.
Datapath Operations	VAM_DMA	*bool VAM_DMA (XAxiCdma *InstancePtr, u32 SrcAddr, u32 DstAddr, int Byte_Length);*	Starting DMA from the *SrcAddr* to *DstAddr* based on the *Byte_Length*.
	VAM_ROUTE	*bool VAM_ROUTE (vam_table_t *VAM_TABLE, int PR[], int nPR);*	Routing the nearest neighbor 2-D switch based on the data and control path.
Control Operations	VAM_START	*bool VAM_START (int PR[], int nPR, int len);*	Launching the accelerator in *PR* region
	VAM_DONE	*bool VAM_DONE (int PR[], int nPR, int len);*	Stalling until the accelerator in PR region is done.

taining a list of free tiles in a queue (the VAM_TABLE). For each *VAM_GET* (steps 1, 4) the interpreter pops a free tile from the queue. The tiles returned for two consecutive *VAM_GET* calls are not required to be adjacent within the array. Functor bitstreams are then loaded into free tiles using *VAM_SET* (steps 3, 6). The VAM interpreter performs this operation by DMA'ing the bitstream resident in DRAM into the ICAP port of the FPGA. The VAM interpreter manages input and output buffers for the accelerator in a similar fashion to tiles. For each input variable the *VAM_GET_BRAM* (steps 2, 5) returns a list of available local BRAM to be used as an input buffer.

Functor Routing After the VAM interpreter transfers the bitstreams into the tiles, the data paths

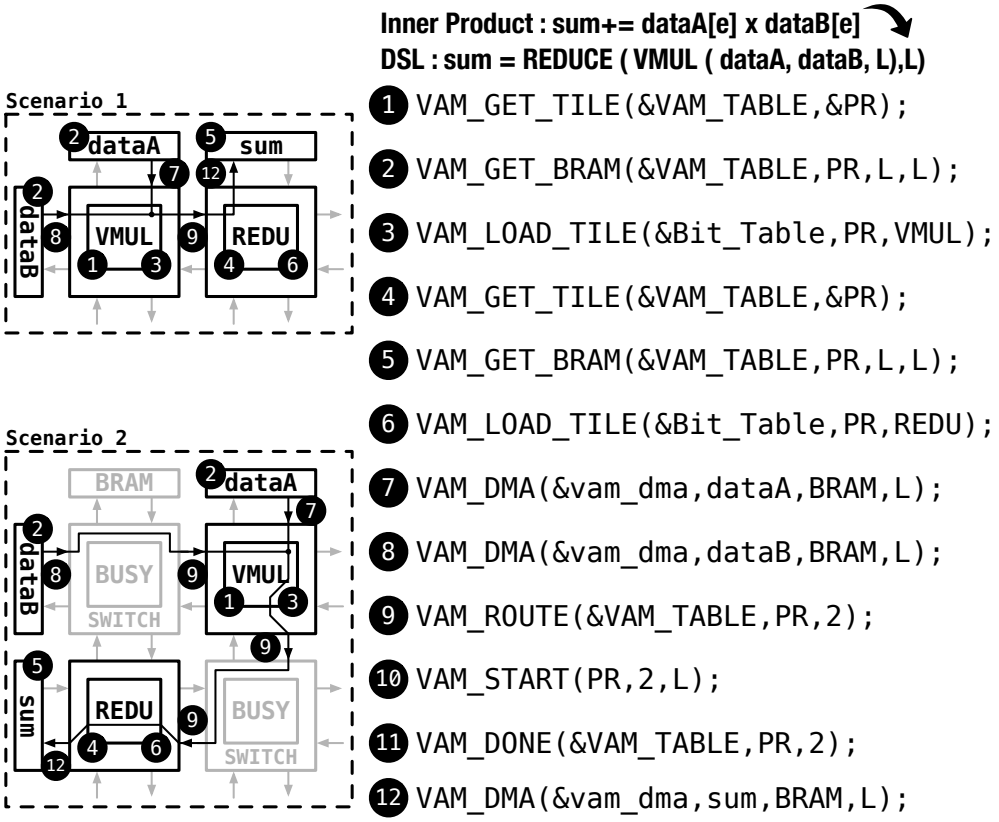


Figure 3.7: VAM Run Time Interpreter Configuration Steps For Inner Product.

represented by *VAM_ROUTE* (step 9) can be set.For this prototype, the VAM interpreter implements a simplified version of maze-routing algorithm [36]. This simple routing algorithm allows the VAM to host multiple accelerators within 3×3 tile array.

Data Transfer The VAM interpreter transfers input data from DRAM into the local input

buffer BRAMS using *VAM_DMA* (steps 7, 8). The outputs of the accelerator are transferred from the output BRAM buffer back into DRAM using *VAM_DMA* (step 12).

Control Operations The *VAM_START* (step 10) initiates the execution of the array. The *VAM_DONE* (step 11) returns status from the accelerator. The VAM run time interpreter prototype was written in 800 lines of C and runs on the MicroBlaze.

3.6 Summary

The JITA approach presents a way of allowing software developers to use standard software development tools and reach software levels of productivity when creating hardware accelerators. It can eliminate synthesis from the path of FPGAs' software applications developers. The JITA presents a prototype system which includes a configurable overlay and a compilation flow as well as a run-time interpreter. The overlay represents an intermediate fabric between design applications and low level FPGA hardware resources. The overlay consists of a number of tiles connected by a 2D mesh interconnect, and local block memories. Each tile is equipped with a partially reconfigurable region that can be populated with functors bitstreams. The PR region behave like computational units which can be replaced at runtime. An accelerator can be formed by downloading its functors partial bitstreams into tiles' PR and configuring the interconnect network to control data flow. The overlay is able to compose multiple accelerators at the same time when there are enough free tiles.

In JITA, Domain Specific languages (DSLs) are the platform for the software developers to express their applications. The approach presented the DSLs compilation flow and how the function calls (VAM) are created to construct an accelerator. The JITA separates the hardware design flow and creating partial bitstreams from the application compilation flow. The hardware design is produced one time by hardware designers while different applications can be created and implemented several times utilizing the provided hardware.

Additionally, the JITA runtime interpreter can compose accelerators by interpreting the com-

piled function calls. The interpreter keeps track of free tiles in order to manage composing multiple accelerators at the same time. The runtime system downloads partial bitsreams to the specified tiles and configures tiles interconnects.

While the presented JITA approach allows application developers to build accelerators on FPGAs, it suffers from the following limitations:

Bitstream Library Size The JITA requires that all variants of programming patterns within a DSL must be synthesized. Otherwise it will fail to compose accelerators that include non pre-synthesized patterns. For instance, the *sumIf* pattern in Figure 3.8 has two different versions (lines 14 & 18), each one has different functionality. We can imagine how many possible operations that can be included in this particular pattern. In practice, DSLs should have software programming flexibility to express a wide range of operations which leads to a large number of pattern versions. That means we need to pre-synthesize all the pattern variations in order to cover all the possible DSL expressions. This would require a large bitstream library.

Conditional Operations The JITA approach cannot handle conditional operations when they appear between patterns. For example, the code in Figure 3.9 has a condition between the two patterns (lines 5 &16). These kinds of conditions prevent composing these patterns as part of an accelerator. When we do not have a pre-synthesized circuit to handle the condition, then the overlay would fail to fully implement this application. The overlay structure expects patterns with a continuous data stream. In the given example, the expression of the condition can be combined with the second *map* (line 16) as an inside condition. However, the new *map* pattern should be pre-synthesized and available in the bitstream library in order to be composed with the other *map* (line 5).

One of the ways to resolve these problems is implementing non synthesized and un-composable patterns using software run by the host processor and other patterns implemented on the overlay. However, this would create high data transfer overhead and increase the complexity of the runtime system.

This work is propsoing a new overlay structure to overcome these limitations including pre-synthesized operators and adding a mechanism to resolve conditional operation problem.

```
object Example7Interpreter extends OptiMLApplicationInterpreter with Example7
trait Example7 extends OptiMLApplication {
  def main() = {
    val simpleSeries = sum(0, 100) { i => i } // sum(0,1,2,3,...99)
    println("simpleSeries: " + simpleSeries)

    val m = DenseMatrix.rand(10,100)
    // sum first 10 rows of m
    val rowSum = sumRows(0,10) { i => m(i) }
    println("rowSum:")
    rowSum.pprint

    // sum(0,2,4,8...98)
    val conditionalSeries = sumIf(0,100)(i => i % 2 == 0) { i => i }
    println("conditionalSeries: " + conditionalSeries)

    // conditional sum over rows of a matrix
    val conditionalRowSum = sumRowsIf(0,10)(i => m(i).min > .01) { i => m(i) }
    println("conditionalRowSum:")
    conditionalRowSum.pprint
  }
}
```

Figure 3.8: All variant patterns must be pre-synthesized. Modfied source from OptiML[58].

```
object Example21Interpreter extends OptiMLApplicationInterpreter with Example21
trait Example21 extends OptiMLApplication {
  def main() = {
    val V1 = DenseVector.rand(1000) // immutable vector initialized to random values
    val V2 = V1.map(e=> e*1000) // normalize values to be between from 0 to 1000    8
    var i = 0
    while (i < V2.length) {
      if (i % 10 == 0) {
         V3(i) = 1
      }
      eles {
         V3(i) = V2(i)/2
      }
         i += 1
    }
    val V4 = V3.map(e => if (e != 1) e * -1)
  }
}
```

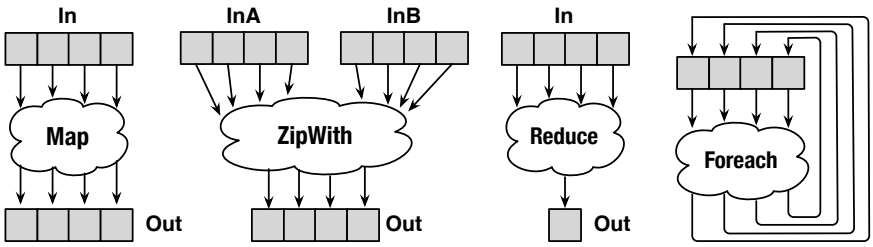
Figure 3.9: Composing conditional operations problem. Modfied source from OptiML[58].

Chapter 4

Proposed Solution

4.1 Introduction

The aim of this work is to move the synthesis process out of the FPGAs programmers' path, increase productivity, and remove the barrier of hardware skill design requirements on programming FPGAs. I believe this can be achieved by the use of pre-build programmable/reconfigurable overlays. By using overlays, we can split programming FPGAs into two phases: hardware design phase and software application phase. In the hardware design phase, the programmable/reconfigurable overlay hardware architecture is created by FPGAs hardware experts. The software application phase, on the other hand, starts when the overlay hardware platform is ready to use. The created overlay would go one time through the hardware design phase; while it will be used by software programmers many times without the need of re-synthesis. This way helps move the synthesis process from application developers' path and increase productivity. Xilinx FPGAs provide Partial Reconfiguration (PR) technique which allows replacing modules at run time by swapping in and out partial bitstreams. The PR technique provides very useful pre-built sub-circuits which can be used at run time to make changes in the circuit functionality. This feature serves our goal of removing synthesis process out of the way of programmers. Thus, our designed overlay, URUK, is equipped with multiple PR regions and a pre-synthesized bitstream library. This chapter is covering the URUK overlay architecture and the system design flow; while the software compilation is covered in the next chapter.



Pattern	Example
Map	In.map { e => e + 1 }
ZipWith	InA.zipWith(InB) { (eA , eB) => eA + eB }
Reduce	In.reduce { (e1 , e2) => e1 + e2 }
Foreach	InA.foreach { e => if (e > 0) inB(e) = true}

Figure 4.1: DSL parallel programming patterns.

4.2 Hardware Design Flow

The hardware design flow consists of two parts. First, creating the overlay static logic shown in the right side of Figure 4.2. In our design, the creation of the overlay static logic is scripted and parametrized to produce overlays with different settings. For instance, the number of tiles can be specified to 2×2 or 3×2 or any other dimensions depending on the target FPGA size. Besides, the PR region size parameters specifying the PR regions size for each tile. The tiles' PR region sizes may have the same or different depending on the design strategy. Once the parameters are set, the overlay static logic design will be created. Three overlay examples are given on the right side of the graph. The Second part which is on the left side of Figure 4.2. This side handles creating partial bitstreams for the DSL parallel programming patterns (i.e. map, reduce, zipwith, etc.) and basic computational operations (i.e. add, sub, mul, log, etc.). The flow starts from the DSL specified primitives (patterns, operators). Then, designers hard code the primitives on C language using one Vivado HLS template, shown on Figure **??**. The interface template maintains common input and output ports for all synthesized primitives. As a result, all PR regions and primitives will have the same I/O interface which allows many primitives synthesized for the same PR region. The coded primitives will be synthesized using Vivado HLS. A checkpoint for each primitive will be created. Then, the checkpoints will be passed to the next stage which is the system synthesis. In the system

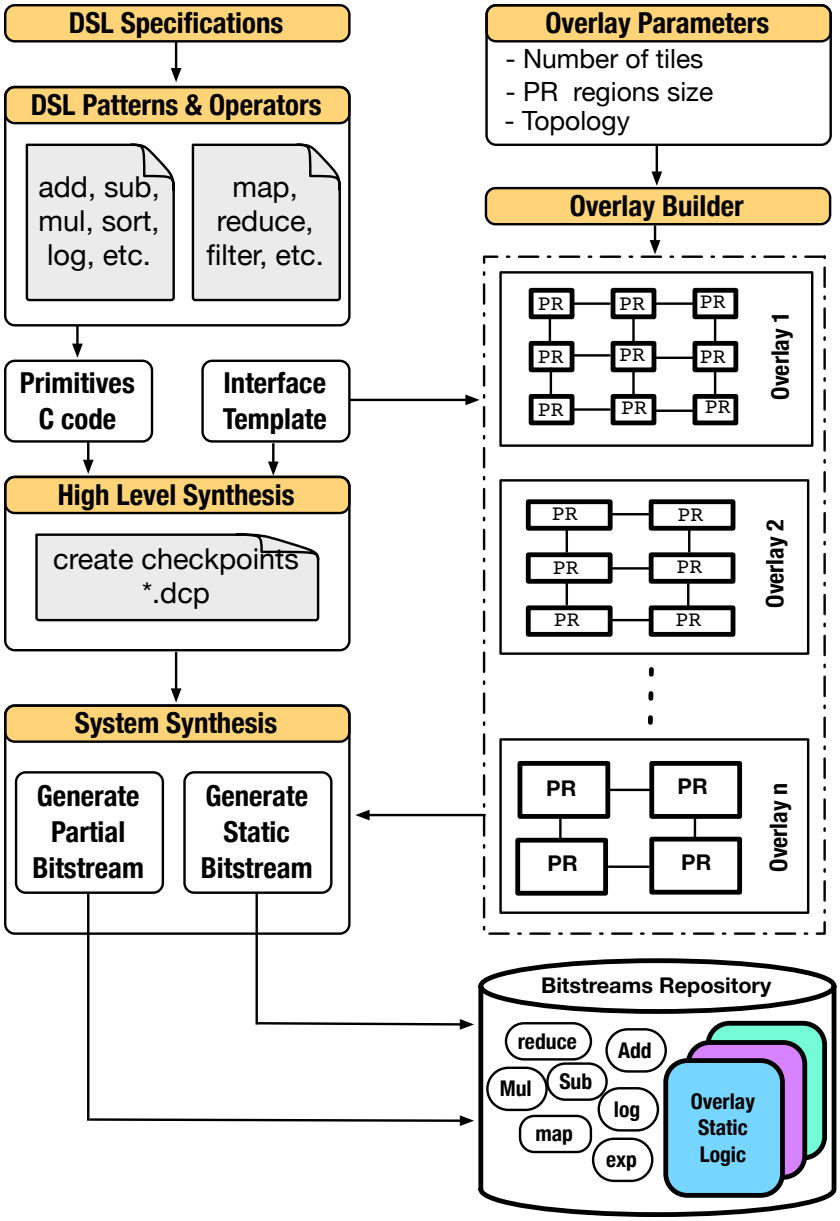


Figure 4.2: Hardware design flow for the overlay static logic and partial bitstreams.

synthesis, the first primitive checkpoint will be assigned to all fitting PR regions in the static logic. Then, partial synthesis begins and generates the partial bitstream for that primitive. This process will be repeated for other primitives. By the end, both static and partial bitstreams will be stored in the bitstream repository to be used later in the DSL application compilation, covered in the next chapter.

```
void Pattern_Name( volatile Type   *Input1,
                   volatile Type   *Input2,
                   volatile Type   *Output,
                   volatile int_1   Start,
                   volatile int_1   Done
                 ){

    //Type can be defined as integer or float
    //int_1 is a single bit type
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE axis port=Input1
    #pragma HLS INTERFACE axis port=Input2
    #pragma HLS INTERFACE axis port=Output

    #pragma HLS STREAM variable=Output depth=16 dim=1
    #pragma HLS STREAM variable=Input2 depth=16 dim=1
    #pragma HLS STREAM variable=Input1 depth=16 dim=1

    if(Start == 1b'1){
        //insert the pattern/operator function here.
        //set the Done signal to 1 when the operation finishes.
    }

}
```

Figure 4.3: Patterns/Operators HLS template.

4.3 URUK Architecture

URUK overlay is structured to compose a wide variety of accelerators using pre-synthesized operators and parallel patterns benefiting from the partial reconfiguration technology. In URUK overlay, the candidate function for acceleration is partitioned into fundamental operators(add, sub, mul, div, and, or, not, and xor) and standard parallel patterns shown on Figure 4.1. These operators and patterns are pre-synthesized and stored into a bitstream library. During system setup time, these small partial bitstreams will be downloaded from the library and assigned to the specified tiles' PR regions. After filling the PR regions, the interconnect network will be configured to route and control the communication data between PR regions in word width level. That gives the URUK overlay an advantage of constructing different accelerators with different functionalities from the same standard bitstreams. What makes URUK overlay different from other PR-Based overlays [] is that the logic of an accelerator is partitioned and distributed among multiple PR regions instead of being employed in one PR region.

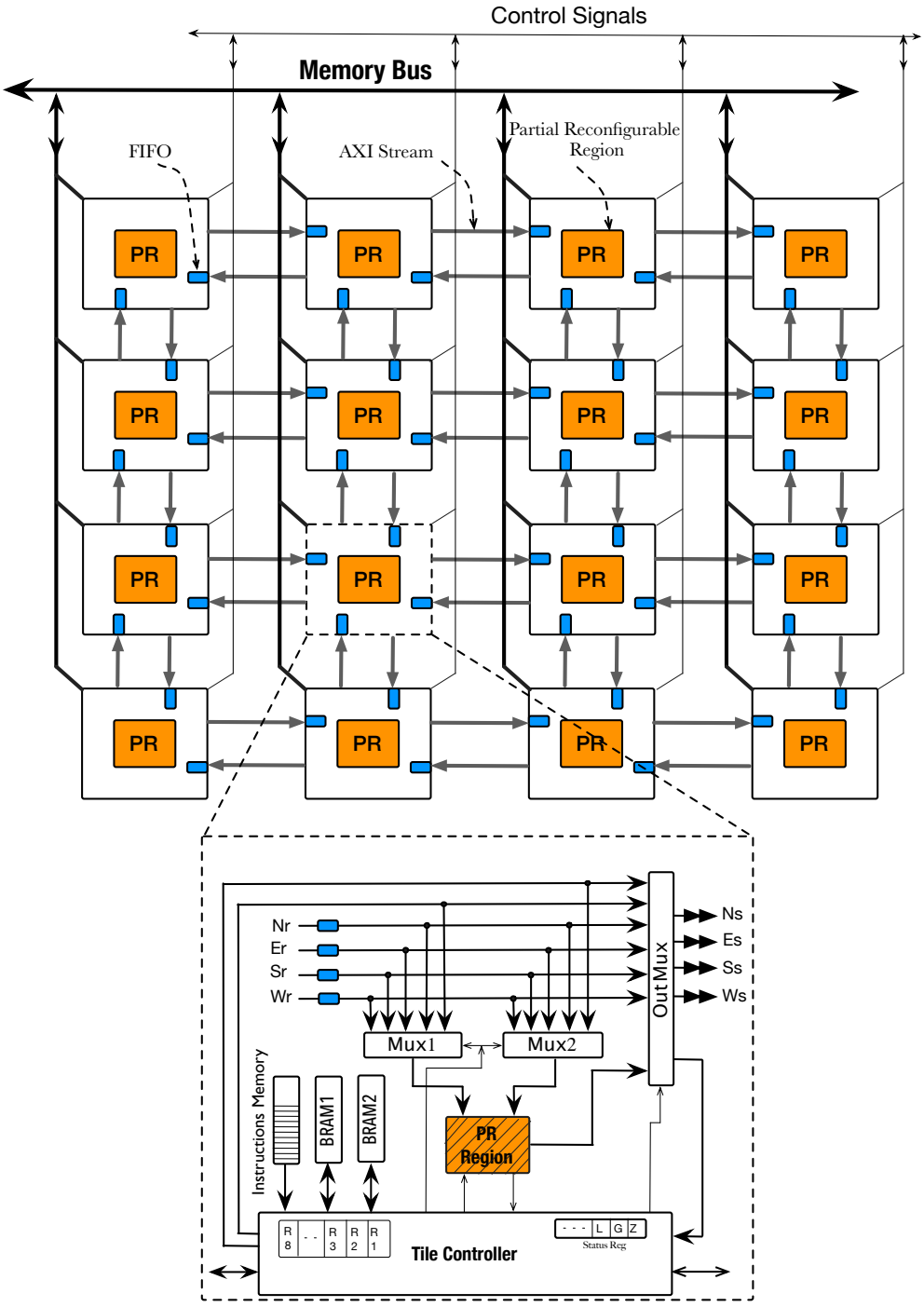


Figure 4.4: 2D URUK Overlay Structure.

Figure 4.4 shows the basic structure of the URUK overlay which has a 3×4 tile matrix. Tiles are connected with each other by a configurable word width 2D mesh interconnect. The interconnect interfaces are simple AXI Stream with FIFOs to maintain synchronization between tiles. Besides, each tile has a link to the memory interface bus to provide the host processor and the DMA engine a direct access to the tiles' local memories. Additionally, there are some other signals (not shown in the graph) between the processor and tiles to issue tiles start execution and check done signal when tiles finish execution.

4.3.1 Tile Structure

Tiles are constructed to have a PR region that holds the main computational logic, word width interconnect switches to direct the data in and out of the tile, two local data BRAMs , one BRAM for instructions, and a controller that controls data movement and computational operations. Tiles are programmable with a special instructions set (see Section4.3.6). The tile components are described in following sections.

4.3.2 PR Regions

At setup time, PR regions can be populated with one of the computational operators or parallel patterns partial bitstreams which can be dynamically replaced to change the functionality. The wrapper of the PR region has two AXI Stream input ports, one AXI Stream output port, start signal, and done signal. Figure 4.3 shows the Vivado-HLS template for a synthesized pattern or operator that will be mapped to PR regions. Based on the tile instructions, the input ports receive data either from a local BRAM or neighbor tiles, and likewise the output port sends data to a local BRAM or other tiles.

The size of the PR regions should be set by the size of pre-synthesized operators and parallel patterns. In URUK overlay, we have two strategies to set the size of the PR regions. First, we can set all the PR regions to one size which the size of the largest pattern/operator partial bitstream in

the library. This choice provides high flexibility and allows any operator or pattern to be mapped to any tile. However, it costs more resources and suffers from internal resources fragmentation when small operators are mapped to a large PR region. The Second strategy is to set the PR regions to different sizes. Few PR regions are set to a large size while the rest are small. This strategy can reduce the internal fragmentation and provide better resource utilization. Contrarily, it reduces the flexibility of mapping any operator or parallel pattern to any tile. In addition, it may increase the communication cost when two dependent operators cannot be mapped to close tiles due to the size fit. Since URUK overlay is automated and parametrized during design time, it provides the two options to set the sizes of PR regions.

4.3.3 Configurable Switches:

Tiles are equipped with programmable switches to route data internally to/from local BRAMs and computational logic in the PR region as well as externally with neighbor tiles. As shown the Figure 4.4 there are two 32 bit multiplexers to direct the data that is either coming from neighbor tiles or local BRAMs to the logic in the PR region. The tile also has one output switch to bypass data to near tiles or stores the data in the local BRAMs. The controller sets these multiplexers based on the execution instructions. The size of input and output switches depend on the tile type. There are three types of tiles with different numbers of interconnects based on their positions in the 2D mesh overlay.

Type A This type of tiles are normally located in the corners of the overlay tile matrix. These tiles have two interconnect input ports, two interconnect output ports, and one port to pass data internally. Figure 4.5(a) shows the structure of type A tiles. The output switch has three multiplexers, two of them to pass data out to the neighbor tiles and one to direct the data internally.

Type B This type is located in the surrounding tiles except the corners. As shown in the Figure 4.5(b), type B has three interconnect input ports, three interconnect output ports, and one to pass data inside. The PR input multiplexers (Mux1 and Mux2) are larger than the PR multiplexers

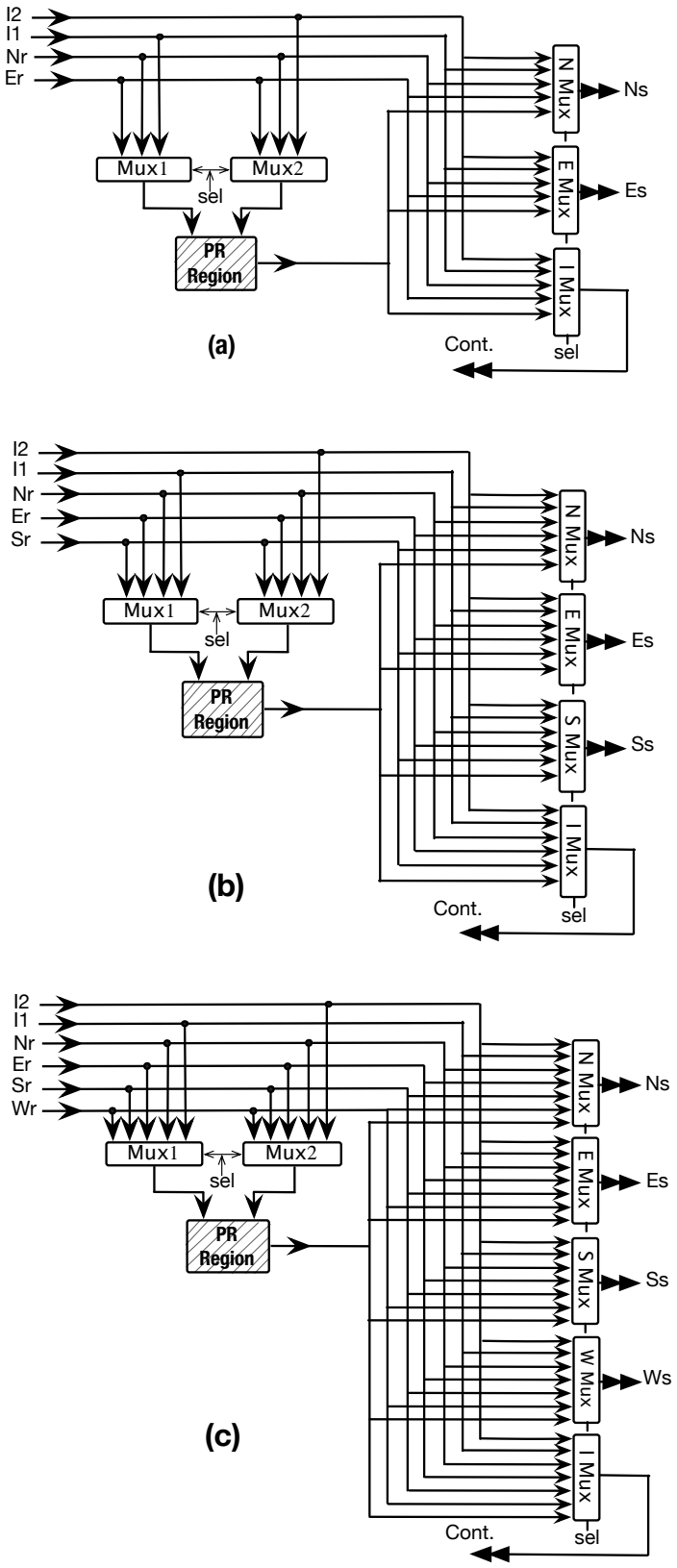


Figure 4.5: Three tile interconnect types.

on type A by one extra input each.

Type C Type C are the inside tiles which require four input ports and four output ports to be fully interconnected with the neighbor tiles. This type has larger multiplexers and use more hardware resources compared to the other two types. Figure 4.5(c) shows the type C structure.

4.3.4 Memory Interface

Each tile is equipped with three dual ports BRAMs, two 64Kbytes each for data and one 8Kbytes for instructions. All BRAMs are connected to memory bus system to allow the host processor and the DMA engine to access them. In addition to the tile interconnect network, the DMA engine can transfer large data between tiles. This is useful when tiles are not neighboring each other, and the data is large. The tile controller, on the other hand, can read from all the three BRAMs and write into the two data BRAMs. The instructions memory holds the executable binaries that is consumed by the tile controller in order to control the data movements and execution.

4.3.5 Tile Controller

The controller fetches and interprets the commands from the instruction memory to control data movement and operations. The controller can issue commands to move data between local BRAMs, internal registers, input and output communication ports. It also selects the input data to the computational logic in the PR region as well as directs the output data from the PR region to local BRAMs, internal registers, or to the output communication ports. Additionally, it can stream a large segment of data from local BRAMs to near tiles. Streaming allows the controller to issue vector operations in addition to the scalar operations.

To highlight here, the controller is designed to only manage the coming and going data to/from the logic in the PR region. In other words, it does not have control over the internal logic of the PR region. This way of design provides high computational capability compared to standard

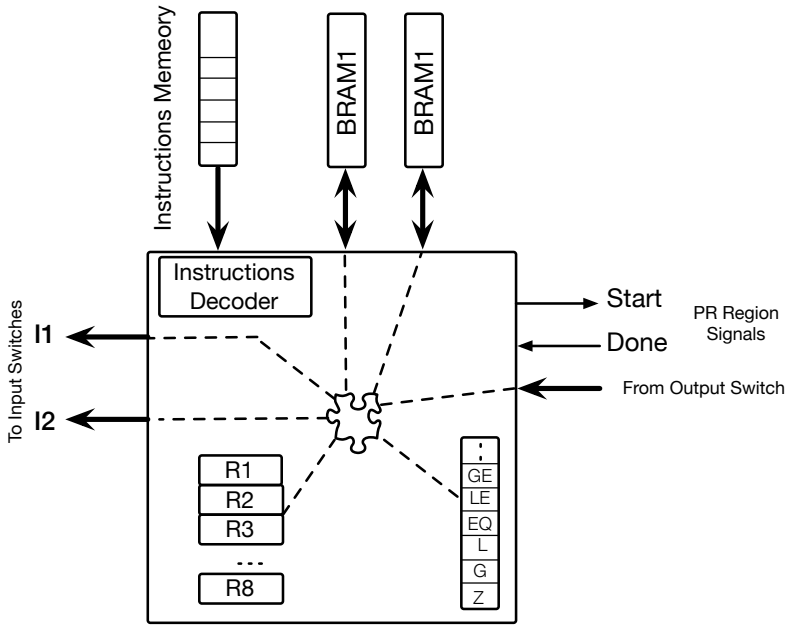


Figure 4.6: Tile Controller.

ALUs. For Instance, when an ALU supports the four basic operations (Add, Subtract, Multiply, and Divide), the ALU will have at least four different commands to represent these operations. In addition a full logic for these operators should exist as part of the ALU. If we want to add some more computational capabilities to the ALU, we need to add more logic, more instructions, and more decoding. However, when we want to add more computational operators or functions to the URUK overlay, we do not need to make changes to the tile controller, interface, neither adding new instructions. All we need is to have a pre-synthesized bitstreams for the new computational operator or pattern swapped in the PR region. This is one of the big advantages that makes the implementation and the compilation more flexible and easier.

4.3.6 Tile Instruction Sets

Special instruction set for URUK is provided to manage data transfer and implement operations. As shown in Figure 4.7, URUK overlay provides 7 instructions categorized into three types: operational instruction*(opr)*, data movement instructions *(mov ,and movi)*, and branching instructions

Category	Instr.	Format & Example	Sources		Destinations	
			Name	Code	Name	Code
Operational	**opr**	opcode S1 S2 D Length **31-28** | **27-24** | **23-20** | **19-16** | **15-0** **0001** **opr** *S1, S2, D, Length* *i.e. opr Nr, Er, Bram1[index], 1024*	**non**	0000	**STR**	0000
			Nr	0001	**Ns**	0001
			Er	0010	**Es**	0010
Data Movement	**mov**	opcode S D Length **31-28** | **27-24** | **23-20** | **19-0** **0010** **mov** *S, D, Length* *i.e. mov Er, Ws, 256*	**Sr**	0011	**Ss**	0011
			Wr	0100	**Ws**	0100
			Bram1	0101	**Bram1**	0101
	movi	opcode D Value **31-28** | **27-24** | **23-0** **0011** **movi** *D, value* *i.e. movi Ws, 0xFFFFFE*	**Bram2**	0110	**Bram2**	0110
			R1	0111	**R1**	0111
			R2	1000	**R2**	1000
Conditional Branches	**jnz** **jgt** **jlt**	opcode Location **31-28** | **27-0** **0100** jnz **0101** jgt **0111** jlt *i.e. jnz 0x2bf* *conditional branches read the flags from the status register	**R3**	1001	**R3**	1001
			R4	1010	**R4**	1010
			R5	1011	**R5**	1011
			R6	1100	**R6**	1100
Unconditional Branches	**jump**	opcode Location **31-28** | **27-0** **1000** jump *i.e. jump 0x2bf*	**R7**	1101	**R7**	1101
			R8	1110	**R8**	1110

Figure 4.7: URUK instructions set and operations code

(jnz, jgt, jlt, and jump). The *"opr"* instruction has the format : *opr source1, source2, destination, length (i.e. opr Nr,Bram1[index], Es,1024)*. The special thing about *"opr"* command is, it does not specify the type of operation such as add, sub, mul, etc. It works as a general command for computation which depends on what computational operand or parallel pattern logic is downloaded to the tile PR region. Based on the *"opr"* command argument, the controller will configure the input and the output switches and also will set the start signal high and wait for the done signal to execute the next instruction. The *"mov"* command can transfer a single, or a stream of data, between registers, local memories, and input and output communication ports. It has the format: *mov source1, destination, length (i.e. mov Bram2[index], Ws,256)*. One important note here about the length in both *"opr"* and *"mov"*, it will be set explicitly when memories are included in the operations or data transactions. Otherwise, the length part will be set to *"0000"*. The reason for that is when the data size is unknown at compilation time as in the example **??**, the length will be set to *"0000"* which indicates that the computation or data transmission will continue until the tile controller receives end code *"0xFFFFFE"* from the transmitter. The command,"movi", is used explicitly for sending the end code to the receiving tile. The *"movi"* is formatted as follows : *movi destination, value (i.e. movi Ns, 0xFFFFFE)*. The URUK overlay is spared with branching instructions to overcome the main JITA approach and allow changing the execution order. The conditional jumps such as Jump Non Zero *(JNZ)*, Jump if Greater Than *(JGT)*, and Jump if Less Than *(JLT)* are made to check the status register *(STR)* and jump to the specified location if the flag is set. Also, the unconditional jump *(JUMP)* is provided to change the execution sequence when it is needed.

Operational Instruction: Since the main computational logic for each tile is based on the downloaded logic into the tile PR region, the tile controller does not need to specify and select the type of the operation (i.e. add, mul, log, map, etc). As a result, the responsibility of the controller is to direct the input data to the logic in the PR region, issues start signal to the PR logic to begin execution, and retrieves the output results from the PR region. Therefore, URUK uses one operational instruction *"opr"* to preform different computations depend on the downloaded logic in the PR region. This reduces the instruction decoder size and complexity. The *"opr"* is

Table 4.1: Conditional branching instructions.

Instruction	Flag	Description
jz	Z = 1	Jump if the zero flag is set.
jnz	Z = 0	Jump if the zero flag is not set.
jgt	G = 1	Jump if the greater than flag is set.
jlt	Z = 0 & G = 0	Jump if the zero flag and greater than flag are not set.
jge	G = 1 ‖ Z = 1	Jump if the greater than flag or Zero flag is set.
jle	G = 0 ‖ Z = 1	Jump if the grater than flag is not set and the zero flag is set.

formatted as follows: *opr source1, source2, destination, length.* The source1 and source2 can be any general register, data memory, or interconnect input as listed on Table 4.7. The destination, also, can be one of the listed destinations on the mentioned table. The length specifies the number of data elements. Based on the specified sources, destination, and length, the controller configures the input and output switches, sets the *Start* signal to "1", and waits for the *Done* signal to execute the next instruction. When memory is used as an operand in the *"opr"*, the start address should be explicitly specified. At run time, the address will be incremented until it reaches the upper bound specified by the length operand.

When the data size is unknown at compilation time, the length will be set to zero in the *"opr"* instruction. The controller will interpret that and put the execution into a while loop with a condition of receiving the end code *"0xFFFFFE"* from the on of the source inputs.

Data Movement: The overlay provides the instruction (*" mov"*) to move data between memories, registers, and communications interconnects. The *"mov"* instruction is formatted to specify data source, destination, and length. When the source is one of the input interconnect ports, and the length is unknown at compilation time, the length will be set to zero. If the length is zero, the controller will continue to read input ports until it receives the end-code,*"0xFFFFFE"*, from the transmitter.

Another move instruction, *"movi"*, is provided to immediately set registers and output interconnect ports to a specific value. The *"movi"* instruction is formatted to define the destination and the immediate value. The destination can be any of the listed destinations on Table 4.7. However,

the output interconnect ports are mostly used with this instruction to set the ”end-code” when the length is unknown (i.e. *”movi Ns, 0xFFFFFE”*).

Conditional Branching To handle conditional operations, URUK overlay provides six instructions as shown on Table 4.1. The conditional branching instructions checks the status register flags and changes the sequence of execution by changing the Program Counter (PC) to the specified address when the condition is true.

Unconditional Branching The overlay, also, provides the *”jump”* to change the execution sequence unconditionally. This instruction adds more flexibility for the overlay programmability.

4.4 Design Automation

The whole design flow is scripted using TCL script to automate the creation of the overlay with multiple settings as well as the bitstream library. The overlay script is parametrized to set the dimension of the tile matrix and the size of PR regions. The HLS script creates the checkpoints for the synthesizable parallel patterns and computational operators using Vivado HLS. Both the output of the overlay and the HLS scripts will used by the top level PR flow script to create the overlay static logic bitstream and the partial bitstream. The script shortens the design development time and reduces possible design errors.

Chapter 5

URUK Compilation Flow

5.1 Introduction

This chapter provides a guideline for the compilation process when targeting URUK overlay. The overlay is designed to support Domain Specific Languages (DSLs) that include parallel programming patterns. The DSL can be written in any software programming language,and it is not necessary to be a hardware DSL such as FSMLang [1]. Figure 5.1 shows the compilation flow of a DSL application targeting URUK overlay. The DSL compiler is responsible for extracting the parallel programming patterns and computational operators from the user source code and generating Data Flow Graphs(DFGs). Each node in the generated DFGs should be a parallel pattern(i.e. map, reduce, zipwith, filter, etc) or a computational operator. Further, the DSL compiler should have a backend generator to create function calls for mapping the DFG patterns and operators to the tile matrix as well as creating binaries for each tile to control data movements based on the DFG. The compiler design and development are out of the scope of this work.

When the DSL compiler creates the DFG from the source code, it will represent the parallel patterns in the graph nodes that already have bitstreams in the library. Further, if the compiler did not find a matchable bitstream for a specific pattern, then the compiler would break down that pattern into its fundamental operators and inserts them in the graph nodes. Using patterns is more efficient in tile utilizations and performance than using pre-synthesized operators as the results show in Chapter6. Therefore, using patterns should be a priority for the compiler if they are available in the bitstream library. The overlay supports pre-synthesized operators as an alternative if a specific pattern is not pre-synthesized.

After creating the DFG, the DSL compiler will search through the nodes and their connec-

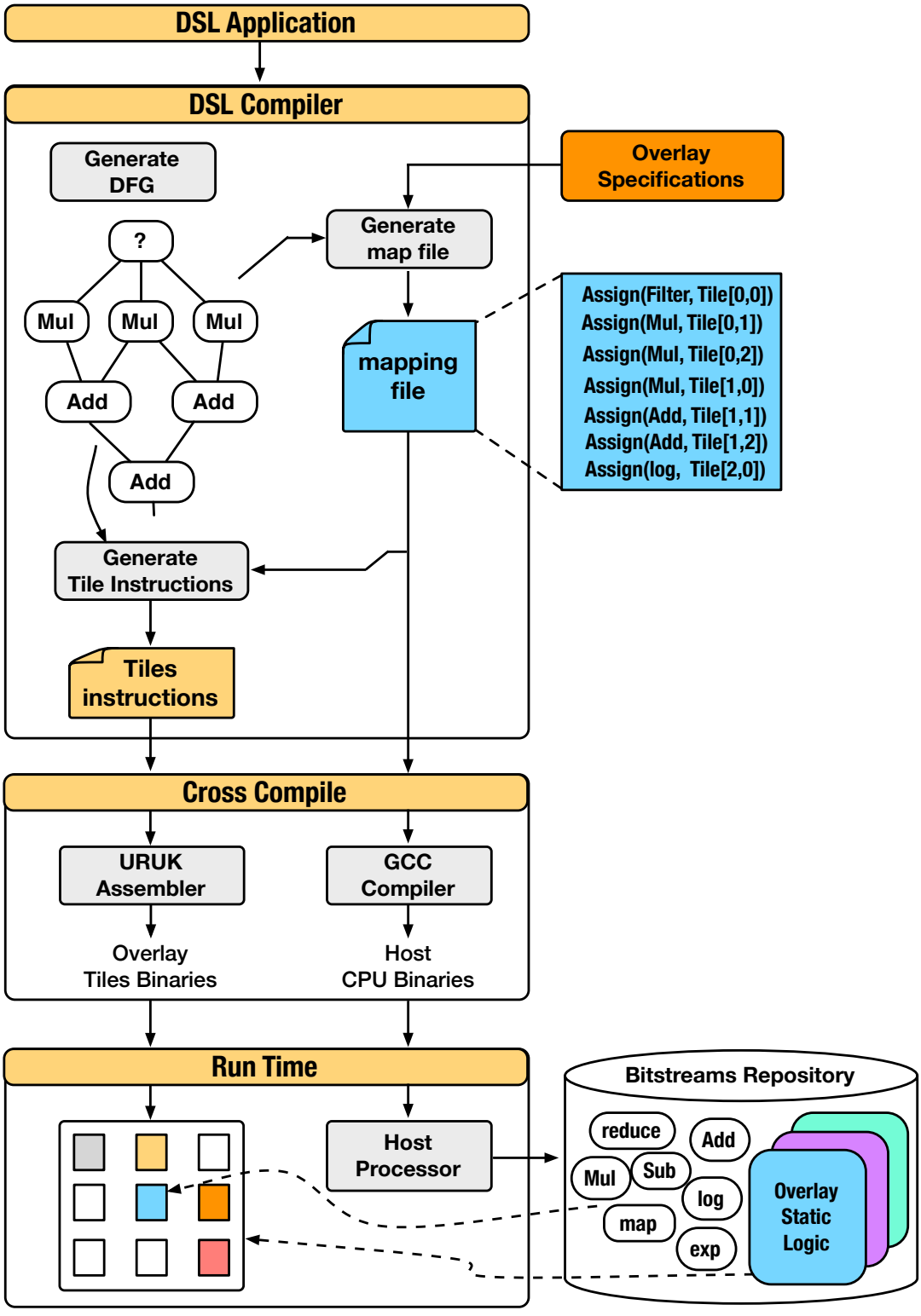


Figure 5.1: URUK compilation flow.

tions to create a *map file* as well as *tile instructions*. The *map file* will contain a sequence of function calls to assign and place partial bitstreams of the DFG nodes (patterns, operators) into the chosen Tiles' PR regions. The overlay allows the compiler to assign nodes to the target tiles offline or leave the assignment flexible for the runtime system which should keep track of the tiles availability in this case.

When the compiler assigns nodes to the tile matrix offline, it should consider reducing communication overhead through placing dependent nodes as close as possible in the matrix. Transmitting data through tile interconnects between two distant tiles keeps other tiles in between busy by passing data. The interconnect latency also will be incremented. Moreover, the compiler would create extra instructions for those intermediate tiles for passing data. Therefore, it is more efficient to place dependent tiles near each other.

After creating the *map file*, the DSL compiler should generate *tile instructions* which will be downloaded into tile's instruction memories at setup time in order to control data movement and tile operations. Each tile will have its own instructions. Additionally, the *tile instructions* should be generated based on the links between DFG node and the tile position in the matrix. The tile instruction sets are presented and explained previously in Chapter 4.

The generated *tile instructions* and *map file* will be cross compiled to generate executable binaries. The *tile instructions* will be translated into executable binaries using *URUK Assembler*, which is built purpose. The assembler is written in Python. It reads the *tile instructions*, translates them into binaries, then writes the binaries into a C header file to be downloaded later into the specified tile's instruction memory.

The main body code of the *map file* function calls is in C language. Thus, it will be cross compiled using a "gcc compiler" to generate executable binaries for the host processor. When executing the function calls, the host processor will download the specified partial bitstreams into the their target tiles.

5.2 URUK Parallelism

Within URUK overlay, parallelism can be achieved by instantiating additional copies of the application throughout additional tiles. This is like unrolling a loop which translates temporal iterations into spacial parallelism. Each copy will have the same DFG patterns of the original one. Besides, The source data should be divided among them be processed in parallel, then the results should be collected back from the tiles that hold the results. This is similar to unrolling loops by a factor in HLS. However, unrolling loops should be done compilation which include synthesis, place and route; while unrolling in the URUK can be done during application compilation which does not require repeating synthesis process.

5.3 Conditional Operations

Conditionals were a barrier that prevents composing patterns in the original JITA. Some DFGs have conditional branching which was a limitation in the original JITA approach. Within URUK overlay, branches are handled by swapping a comparator bitstream into a tile PR region, supplying the inputs, and feeding the PR region's output to the status register within the same tile. Based on the status of the flag, the next command will change the direction of data, or change the execution sequence by modifying the Program Counter (PC) and jump to a specified address.

5.4 Domain Specific Langauges

This section provides a brief introduction about possible Domain Specific Languages (DSLs) that can serve as developing applications as well as compiling for the overlay. DSLs (eg. Python, Snort, HTML) are common within software development. DSLs promote the use of languages tuned for the needs of specific application domains. Once created and tuned, the language promotes increased programmer productivity through appropriate abstractions and heavy reuse. DSLs are

also being considered to generate accelerators within FPGAs. How DSLs are currently being used with FPGAs can be found in [8, 11, 28]. A DSLs ability to define reusable programming patterns is advantageous to moving the use of CAD tools and synthesis from application programmers development flows.

Delite framework [12] can be modified and used to compile for the URUK overlay for several reasons. First, Delite facilitates the definition and construction of a DSL language and includes the generation of the compiler for the new language. Delite also is built in a modular fashion to allow the insertion of unique domain specific optimizations to be included into the compiler flow. All DSLs then take advantage of the built in traditional lower level instruction optimizations, such as common subexpression elimination, loop fusion, etc. Importantly, Delite's modular structure allows new backends to be easily added. These features can make the job easy of integrating the *map file* and *tile instructions* generators in a new backend.

5.5 Data Flow Graph

Since the overlay enables utilizing PR regions with pre-synthesized programming patterns as well as pre-synthesized computational operators, the compiler can create Data Flow Graphs (DFGs) based on patterns, operators, or mixes between them.

5.5.1 Pattern Based

In general, functional languages (i.e. Python, Scala, etc) support programming patterns such as: *map*, *reduce*, *zipwith*, *filter*, etc. Also, these patterns are provided within DSLs (i.e OptiML). In software programming, patterns are linked to implement a specific function. For instance, by linking the *zipwith* and the *reduce* pattens, we can implement matrix multiply. If these patterns are pre-synthesized, then their bitstreams can be downloaded to two overlay tiles and linked together by tile interconnects. In URUK Pattern Based (UPB), each node in the DFG represents a pro-

grammable pattern. Therefore, the DFG generator should search through the user source code for patterns and their data flow as well as searching the bitstream library for matchable bitstream. If the DFG generator does not find a matchable bitstream for a specific pattern, the generator will replace it by its basic operations and expand the DFG nodes.

The number of nodes in the pattern based DFGs are equal or less than the number of nodes in the operator based DFGs for the same application. Using patterns are advantageous due to their fine-grind place and route. In contrast, when patterns are not pre-synthesized, they should be replaced by operators. In some cases, the performance will decrease due to the expansion of the original pattern among tiles, a tile for each operation. The communication latency increases by the number of tiles because it adds around 2 clock cycles to every transmitted data element.

In contrast, large prgramming patterns require big PR regions which may lead to inefficient FPGA resource utilizations. The overlay is designed to keep balance between performance and FPGA resource utilizations which is discussed in Chapter6.

5.5.2 Operator Based

In the original JITA approach, all variant programming patterns must be pre-synthesized. In practice, this requires a large bitstream library to cover all the DSL patterns diversity. URUK overlay overcomes this problem by supporting pre-synthesized computational operators when some patterns are not pre-synthesized. In URUK Operator Based (UOB), patterns and other scalar operations are represented in the generated DFG nodes on their basic operators. The UOB utilizes more tiles, but makes composing accelerators, using pre-synthesized primitives, flexible and possible. With few types of pre-synthesized operators, different accelerators can be composed. For evaluation purposes, all DFG nodes represent operators when the UOB is used. the next two examples illustrate how the UOB data flow graphs are mapped into the overlay.

5.6 Example 1

This example illustrates how pattern based and operator based data flow graphs are represented and placed on the overlay. Additionally, the example shows the created function calls and tiles' instructions to compose and implement the accelerator. Figure 5.2 displays a source code, written in OptiML syntax, which conditionally sum matrix rows. The function is represented in DFGs shown in Figure 5.3. The DFG on the left side will be created when the *sumRowIf* is pre-synthesized and available in the bitstream library. If it is not pre-synthesized, the DFG generator would create the DFG on the right side in which the *sumRowIf* is replaced by its internal basic operations (e.g. min, compare, and reduce).

Then, the DFGs are placed separately on a 2×2 overlay as shown in Figure 5.4. The example assumes that we do not have a pre-synthesized *random* circuit, and the random numbers will be generated on the host processor and transferred to *Tile_00* local memory. The DFG in the left side, Figure 5.3 (a), requires two tiles, one multiply and one for the *sumRowIf*. While, the DFG in the right side requires four tiles as shown in Figure 5.4 (b).

The example, also, presents how conditionals are implemented within the overlay as shown in Figure 5.4 (b), *Tile_10*. The output of the comparator on *Tile_11* is sent to *Tile_10* and written in the Status Register (STR). Then, the next instruction in *Tile_10* will check the *gth* flag. Based on the flag value, the controller will either implement reduce or flush the data buffer and read the next row from *Tile_00*.

The function calls are shown in the bottom of the tile matrix in Figure 5.4. The function calls will be executed by the host processor to download the partial bitstreams at runtime. Additionally, the generated tile instructions are shown in Figures 5.5, 5.6.

```
object Example7Interpreter extends OptiMLApplicationInterpreter with Example7
trait Example7 extends OptiMLApplication {
  def main() = {
    val m = DenseMatrix.rand(4,100)
     m = m * 100
    // conditional sum over Vector
    val conditionalSum = sumRowIf(0,4)(i => m(i).min > 0) { i => m(i) }
    println("conditionalSumRow:")
    conditionalSumRow.pprint
  }
}
```

Figure 5.2: A DSL source code.

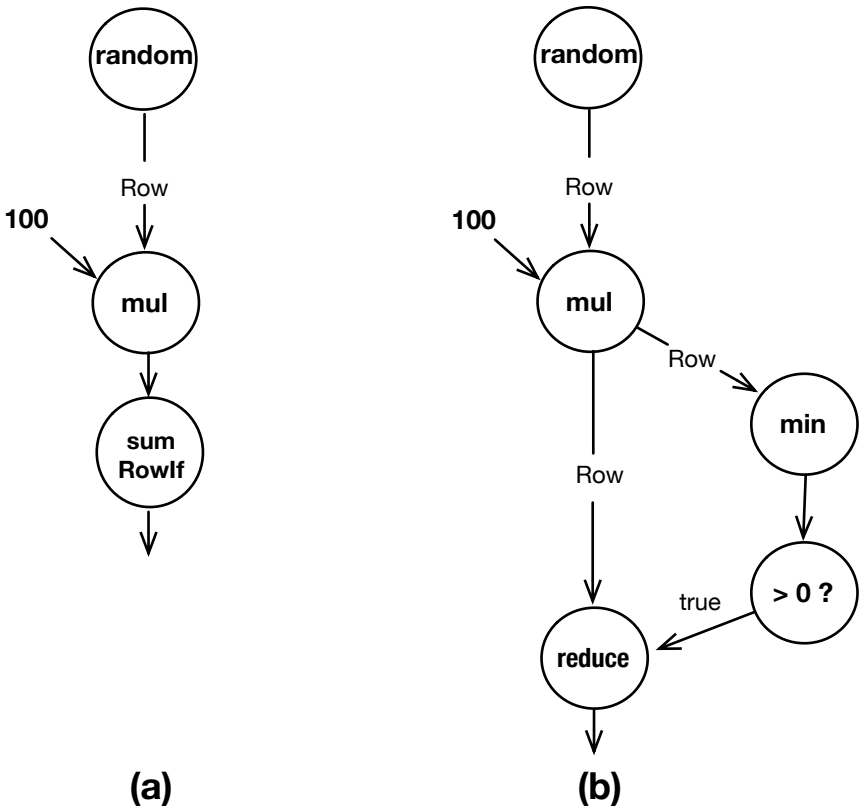


Figure 5.3: Data Flow Graphs (DFGs) for the code in Figure 5.7

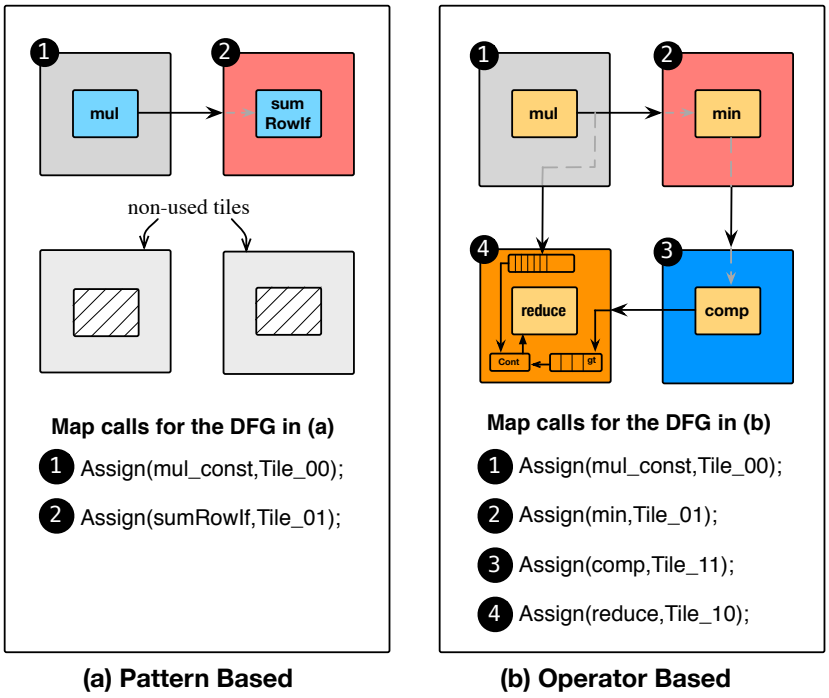


Figure 5.4: Place&Route DFGs in Figure 5.8 on 2×2 URUK overlay.

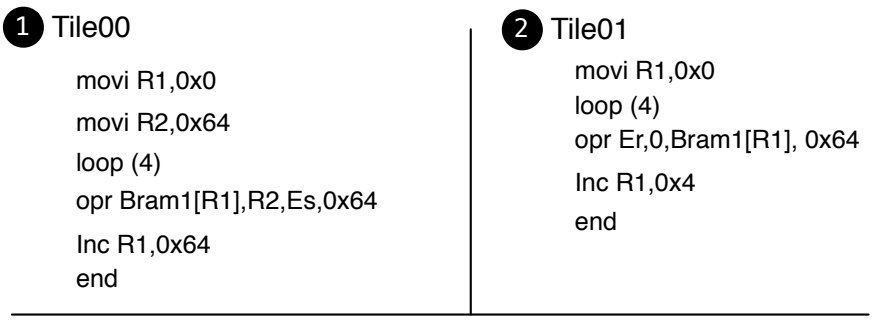


Figure 5.5: Instructions and executable binaries for the two placement examples in Figure 5.9

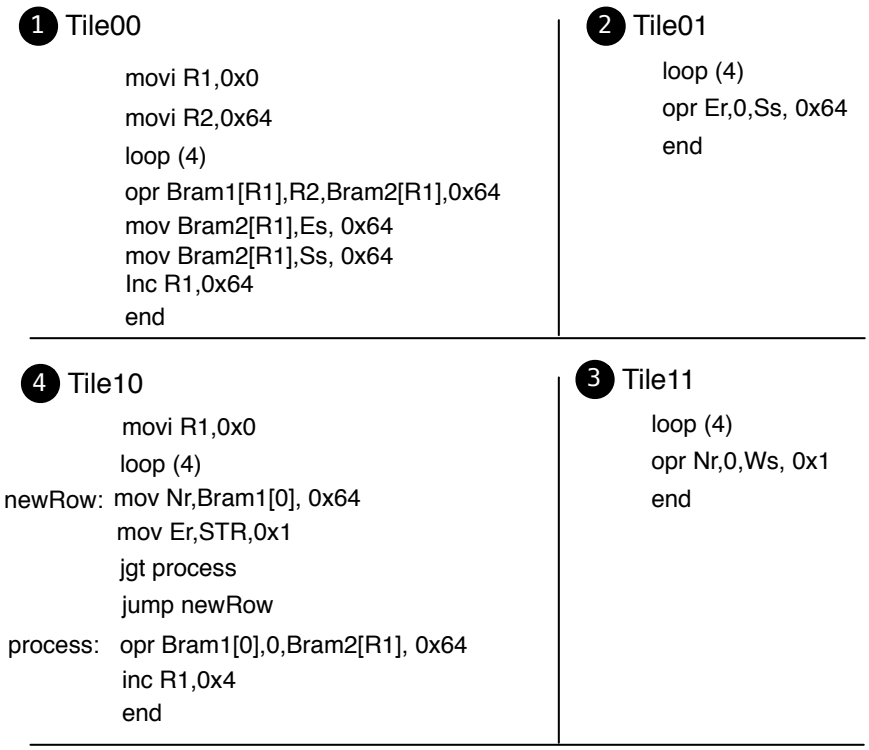


Figure 5.6: Instructions and executable binaries for the two placement examples in Figure 5.9

5.7 Example 2

In this section, we provide an example that shows how to compile a source code from OptiML, a domain specific language for machine learning built by PPL on top of Delite framework [59], targeting the URUK overlay. The source code in Figure 5.7 has some programming patterns such as: random, filter, map, reduce and other operations like multiply and log. The compiler would generate a DFG as shown in Figure 5.8 in which each node represents a pattern or an operator. When the compiler finds a pattern in the source code, it will look for a matchable bitstream in the pre-synthesized library. If it did not find one, then it will break down the pattern into its basic operators and represent them in the DFG nodes. As illustrated in Figure 5.8, when the compiler finds a map pattern that squares elements multiplied by a constant, it generates the DFG in (a). If not, it generates the DFG in (b) through replacing the map pattern by two nodes, a node for square and the other for multiply by constant.

```
object Example1Interpreter extends OptiMLApplicationInterpreter with Example1
trait Example1 extends OptiMLApplication {
  def main() = {
    val v = DenseVector.rand(1000)

    // filter selects all the elements matching a predicate
    // map constructs a new vector by applying a function to each element
    val v2 = (v*1000).filter(e => e < 500).map(e=>e*e*random[Double])

   // reduce produces a scalar by successively applying a function to pairs
   val logmin = v2.reduce((a,b) => if (log(a) < log(b)) a else b)
  }
}
```

Figure 5.7: OptiML Example .

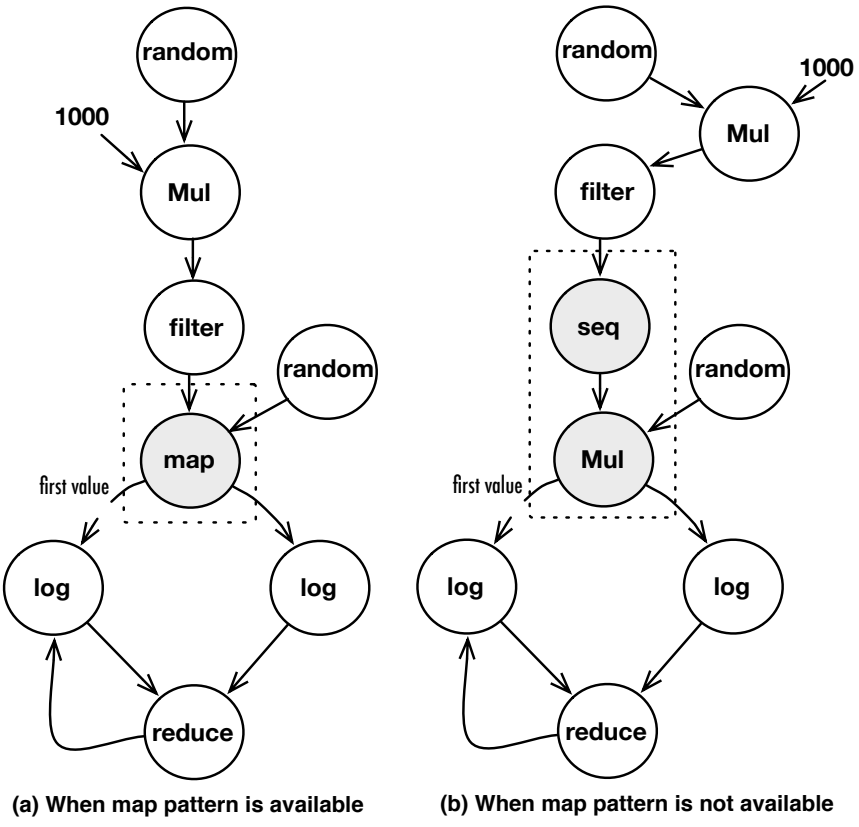


Figure 5.8: Data Flow Graph for the code in Figure 5.7

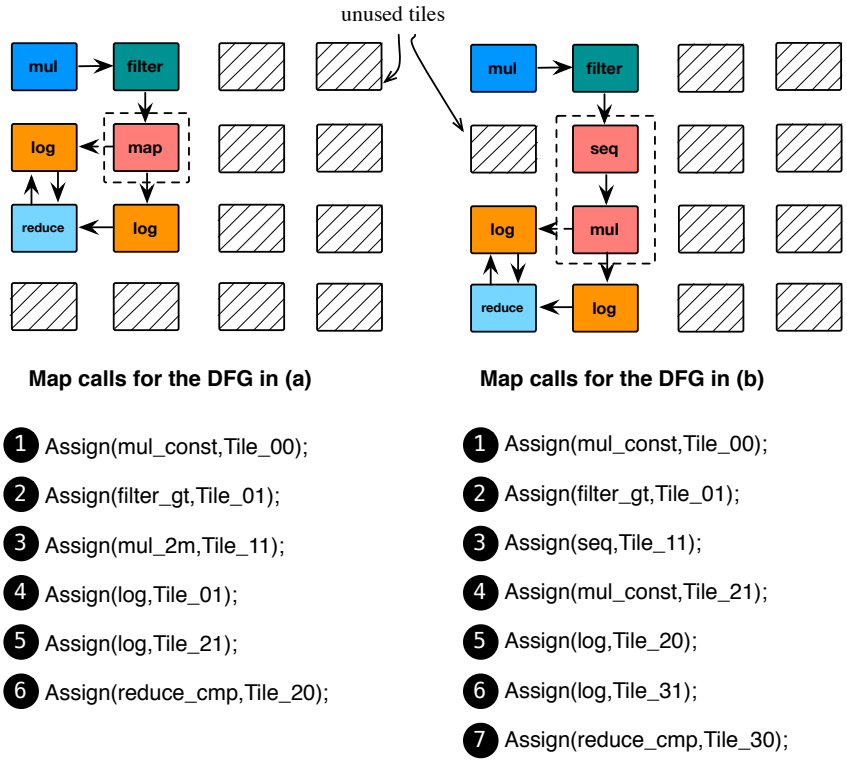


Figure 5.9: Place&Route DFGs in Figure 5.8 on 4×4 URUK overlay.

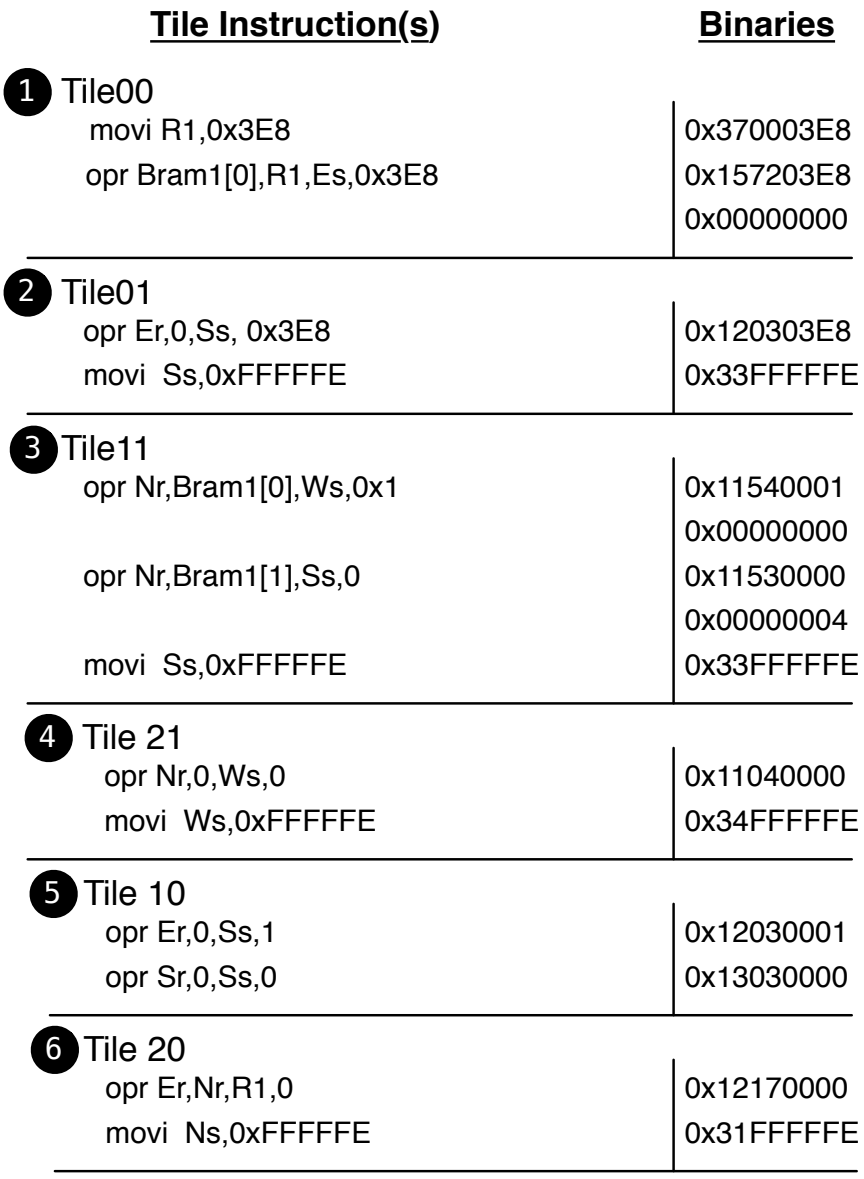


Figure 5.10: Instructions and executable binaries for the pattern based placement in Figure 5.9 (a).

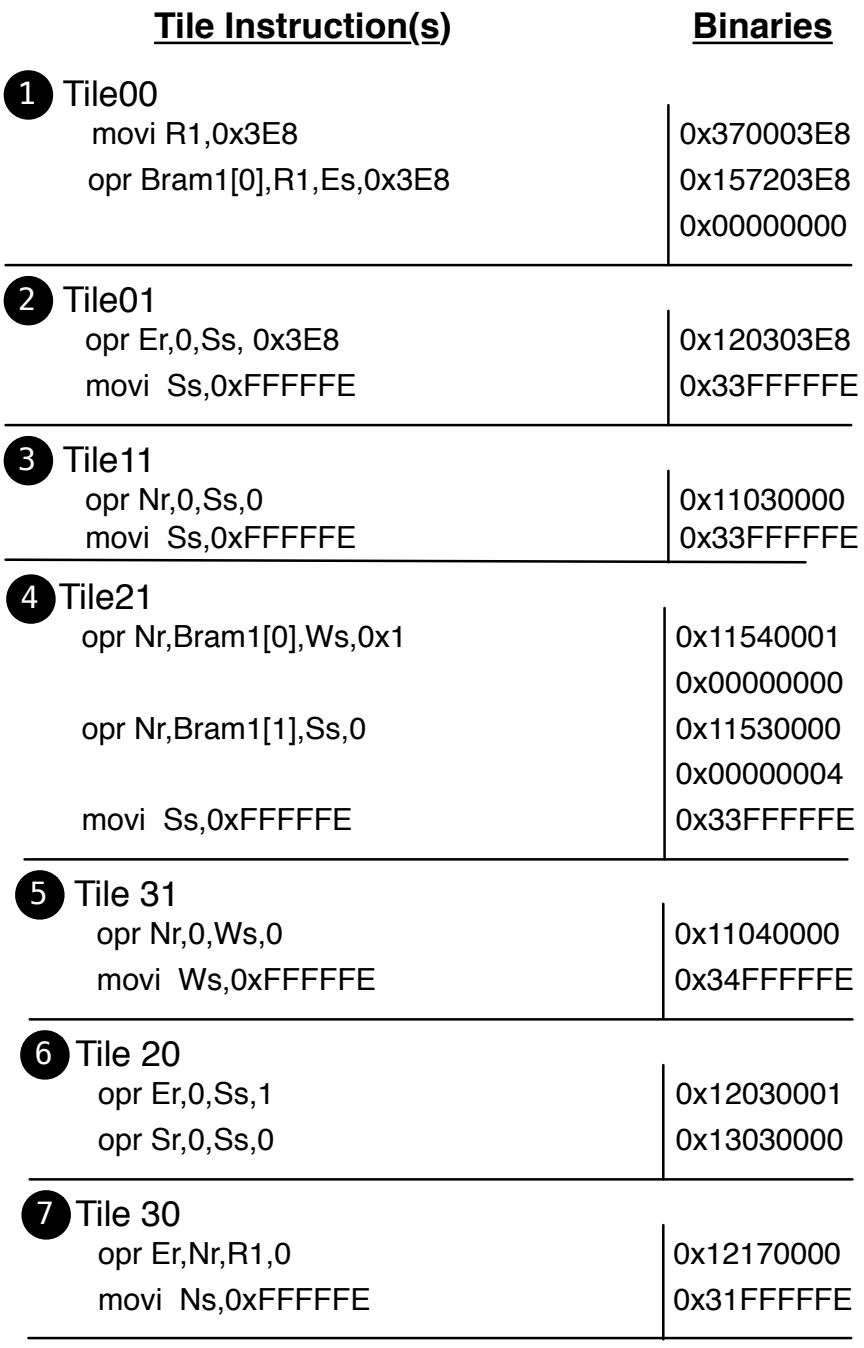


Figure 5.11: Instructions and executable binaries for the operator based placement in Figure 5.9 (b).

Previously mentioned that the URUK overlay comes up with different settings such as the number of tiles and PR region size. Therefore, the target overlay settings should be known during compilation. In this example, we are using a target overlay with 4×4 tile matrix and same size settings for all PR regions (the size of the largest pattern).

Once the DFG created, the DSL complier *map file* generator will search where to place nodes within the tile matrix and create a *map file*. The *map file* will have function calls that will be used by the during accelerator setup time to download the nodes' partial bitstreams to the specified tiles as shown in Figure 5.9. From the example, we can see that the DSL compiler has several solutions to place and route nodes within the given target overlay. Since URUK overlay allows more than one accelerator to be implemented within the overlay at the same time, it would be efficient to reduce the *"dead"* tiles such as the tile in the second row and first column ,"tile_10", on Figure 5.9(b). Even though *"dead"* tiles can be used in other accelerators that work concurrently on the overlay, but they will degrade the performance of neighbor tiles by keeping them busy bypassing data. Also, they will make code generation for neighbor tiles more difficult by including extra *"mov"* instructions to bypass data .

It is clear that the search space for the place and route within URUK overlay is extremely less than the search space on fine-grind FPGAs. This makes a significant difference in the compilation time between using the overlay and the traditional FPGA applications development tools.

In addition to the *map file*, URUK generator will create instructions for each tile to control data flow between tiles. Figure 5.11 shows the created instructions and their binaries for each tile in both cases which are represented on Figure 5.9. The tile controller will ignore the source operand in the *"opr"* instruction when it is set to zero because in some patterns the computational logic requires only one source operand. Further, when the operand *length* is set zero in both *"opr"* and *"mov"* instructions, the tile controller will keep executing the same instruction in a while loop until it receives the end code,*"0xFFFFFE"* from the sending tile. This is important when the data size is unknown during compilation.

Chapter 6

Evaluation

This chapter presents an evaluation of the thesis' claims and questions put earlier in Section 1.2. The newly designed overlay, URUK, is programmable and able to compose pre-synthesized programming patterns as well as pre-synthesized computational operators. Additionally, the thesis claims the new overlay can handle conditional operations. Thus, we chose benchmark functions that include patterns and operators with conditionals to evaluate these claims. Next, the benchmark functions are implemented on the overlay using patterns then using operators to measure the impact of replacing patterns by operators on resource utilizations and performance.

High Level Synthesis (HLS) has been considered as a robust compilation technology to increase productivity. Hence, the HLS was chosen to compare against URUK's performance, resource utilizations, and productivity. Further, a software version for each benchmark function is implemented on MicroBlaze to be the common factor in the speedup of both the HLS and URUK. Moreover, the optimization of both HLS and URUK is presented to assess the hardware skills' requirement to gain more speed up. The last section discusses the flexibility of URUK overlay for being dynamic and compared against a static overlay from three different perspectives : area, routing data, and achieving parallelism.

6.1 Benchmark:

Table 6.1 shows the used benchmark functions to evaluate the approach. The benchmark includes vector addition, vector multiplications, matrix multiply, and four other functions with Compound Patterns (CP). The CP functions are chosen to show the differences between the use of pre-synthesized patterns and pre-synthesized computational operators as well as to examine condi-

Table 6.1: Synthetic Benchmark Functions

Name	Formula	Patterns	Operators
VADD	$\vec{v_c} = \vec{v_a} + \vec{v_a}$	zipwith	add
VMUL	$\vec{v_c} = \vec{v_a} \cdot \vec{v_a}$	zipwith	mul
Matrix Multiply	$M_c = M_a \times M_b$	zipwith , reduce	mul , add
CP1*	$cost = \sum_{i=0}^{n} (y_{pred} - y_{real})^2$	zipwith, map, reduce	sub,sqr,add
CP2†	$f(x_i) = \begin{cases} \sum_{i=0}^{n} \sqrt{2x_i - 1}, & \text{if } x_i > 0 \\ null, & \text{otherwise} \end{cases}$	map, reduce	$>$,sqrt,mul,sub
CP3‡	$f(x_i) = \frac{2x_i}{1+\lvert x_i \rvert}$	map	mul,div,add,abs
CP4§	$f(x_i) = \begin{cases} x_i^2 + 2x_i + c, & \text{if } x_i > 0 \\ null, & \text{otherwise} \end{cases}$	filter, map	$>$,sqr,add,mul,add

*V3 =V1 - V2; V4 = V3.map(e=> e^2).reduce((a,b)=>a+b).
†V2 = V1.map(e=> if (e >0) sqrt(2*e -1) else 0).reduce((a,b)=>a+b).
‡V2 = V1.map(e=> 2*e/(1+abs(e))).
§V2 = V1.filter(e=> e>0).map(e=> e^2 +2*x+ const).

tional operations. Besides, the table presents the patterns and operators involved in each function.

6.2 HLS Implementation

For comparison purposes, we created a full hardware for each function on Table 6.1 using Vivado HLS. The created HLS accelerators were not optimized with directives and pragmas in the first implementation because our study is targeting software developers with little background on FPGAs hardware design. In the second implementation, we used an optimized HLS source code for matrix multiply to compare the optimization difficulty and benefits of both HLS and URUK. Table 6.6 shows the resource utilization of the synthesized accelerators in term of BRAMs, DSPs, FFs, and LUTs. From the table, it is obvious that the optimized matrix multiply is utilizing more resources compared to the non-optimized one. All accelerators were synthesized for 100MHz frequency.

Table 6.2: Resource Utilizations of HLS Full Accelerators on Vertix7

Function	**BRAMs**	**DSPs**	**FFs**	**LUTs**
Vector Add	0	0	300	427
Vector Multiply	0	4	268	128
Matrix Multiply (unoptimized)	0	12	689	585
Matrix Multiply (optimized)	0	28	1046	1393
CP1	0	4	203	224
CP2	0	0	1636	3367
CP3	0	0	489	554
CP4	0	4	267	203

Table 6.3: Resource utilizations of Computational Operators on Virtex7

Operator	**BRAMs**	**DSPs**	**FFs**	**LUTs**
Add	0	0	3	48
Sub	0	0	3	48
Mul	0	4	6	15
Div	0	0	293	336
Logf	0	13	572	1380
Expf	0	7	400	1697
sqrt	0	0	1440	3144
fmax, fmin	0	0	578	2240
abs	0	0	3	91
comp (>, <, ==, >= ,<=)	0	0	8	81

6.3 URUK Implementation

For prototyping, we created an URUK overlay with 3×3 tiles matrix as well as a bitstream library which includes pre-synthesized patterns and computational operators. Tables 6.5& 6.3 show the resource utilizations of the pre-synthesized patterns and operators respectively. From Table 6.5, the pattern *map{sqrt(2x-1)}* requires 1506 FFs and 3180 LUTs, which is the highest among other patterns and operators. Therefore, the PR regions, as shown on Table 6.4, are set to fit the largest used patterns. Table 6.4 also displays the total tile size on Vertix7 including the PR region and other logics.

Table 6.4: Tile's Resource utilization on Virtex7

Part	BRAMs	DSPs	FFs	LUTs
Tile logic *	45	0	921	2072
PR Region	0	18	1506	3180
Total	45	18	2427	5252

*Including three interconnect switches, five 256 streaming FIFOs, two 64Kbytes data BRAMs, one 8Kbytes instruction BRAM, and the controller logic.

Table 6.5: Resource utilizations of Programming Patterns on Virtex7

Pattern	BRAMs	DSPs	FFs	LUTs
map{e=> e*e}	0	4	73	17
map{(a,b) => if (a>b) a else b}	0	0	41	59
map{e => sqrt(2*x -1)}	0	0	1506	3180
map{e=> 2*e/(1+abs(e))}	0	0	360	444
map{e=> e*e + 2*e + const.}	0	4	138	53
A.zipwith(B){(eA,eB) => eA + eB}	0	0	39	49
A.zipwith(B){(eA,eB) => eA * eB}	0	4	40	16
A.zipwith(B){(eA,eB) => eA - eB}	0	0	39	49
reduce{(a,b) => a*b}	0	0	90	104
filter{e=> if(e>0) e }	0	0	37	28
filter{e=> if(e<0) e }	0	0	37	28
filter{e=> if(e==const.) e }	0	0	37	28
filter{e=> if(e>=const.) e }	0	0	37	28
filter{e=> if(e<=const.) e }	0	0	37	28

To study the differences between the use of pre-synthesized programming patterns and pre-synthesized computational operators within URUK overlay, we implemented the Compound Patterns (CP) functions on the benchmark table based on patterns as well as operators. Table 6.6 presents the resource utilizations in term of tiles for the benchmark functions in both pattern's based and operator's based implementations. The performance section will discuss how that will impact the execution time and the overall speed up.

Table 6.6: Tile Utilizations of Benchmark Functions on URUK Overlay.

Function	**Pattern Based**	**Operator Based**
Vector Add	1 Tile [zipwith]	1 Tile [add]
Vector Multiply	1 Tile [zipwith]	1 Tile[mul]
Matrix Multiply	2 Tiles [zipwith, reduce]	2 Tiles [mul, add]
CP1	3 Tiles [zipwith, map, reduce]	3 Tiles [sub, sqr, add]
CP2	2 Tiles [map, reduce]	4 Tiles [greater than, sqrt, mul, sub]
CP3	1 Tile [map]	4 Tiles [mul, div, add, abs]
CP4	2 Tiles [filter, map]	5 Tiles [greater than, sqr, add, mul, add]

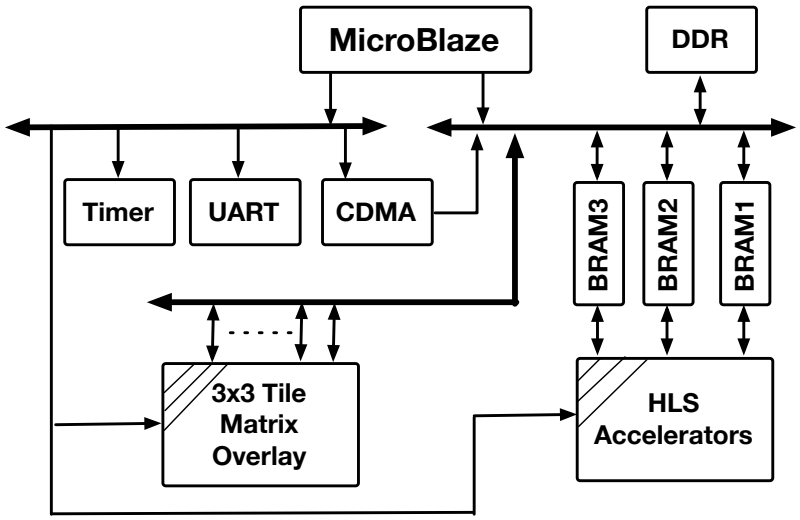


Figure 6.1: Prototype System.

6.4 Prototyping System

We built a system that includes a MicroBlaze, a Central Direct Memory Access (CDMA), a timer , a 3×3 URUK overlay, and custom HLS full accelerators for each function in the chosen benchmark. In this work, the HLS accelerators' performance represents the baseline to compare against URUK performance. Figure 6.1 displays the prototype system. The figure shows only one HLS accelerator box connected to three BRAMs; however, in the actual system, every HLS accelerator is connected independently to three BRAMs, 64 Kbytes each. The system is built on Vertix7 using Vivado 2015.4 and synthesized at 100MHz frequency. The MicroBlaze is configured with the

following features:

- *256 Kbytes for data and 256 kbytes for instructions*
- *Data cache and instruction cache are enabled with 16 kbytes and 8 line length each*
- *Extended floating point unit is enabled*
- *32 bit integer multiplier is enabled*
- *Integer divider is enabled*

The MicroBlaze plays the role of the host processor in the system to set the CDMA in order to transfer data from the DDR to tiles' local memories as well as the HLS accelerators memories. At the accelerator setup time, the MicroBlaze downloads the partial bitstreams to the specified tiles. Additionally, the MicroBlaze issues a start signal to tiles matrix and HLS accelerators to begin executing. Then, it waits until they finish to calculate the execution time and print the output results. The MicroBlaze is also used to run the software version of the testing functions in order to calculate the speed ups of the overlay and the HLS accelerators. In this study, the MicroBlaze's execution time is not the baseline target to evaluate the overlay performance. However, it represents the common factor in the speed up measurement.

The CDMA can transfer data between the DDR and the local BRAMs as well as between local BRAMs themselves. The tiles memory interface allows the CDAM to transfer data between them. It is crucial to transfer data between tiles by using the CDMA instead of using the communication interconnect when the data size is large. In the system, the CDMA is mastered by the MicroBlaze.

The timer is used to time the execution time for both the overlay and the HLS accelerators. Before measuring the actual execution time, the time calibration is calculated to be subtracted later from the total execution time for relatively accurate measurements. The 32-bit AXI timer IP is used in the implemented system. To print out the results, the AXI UART Lite is used.

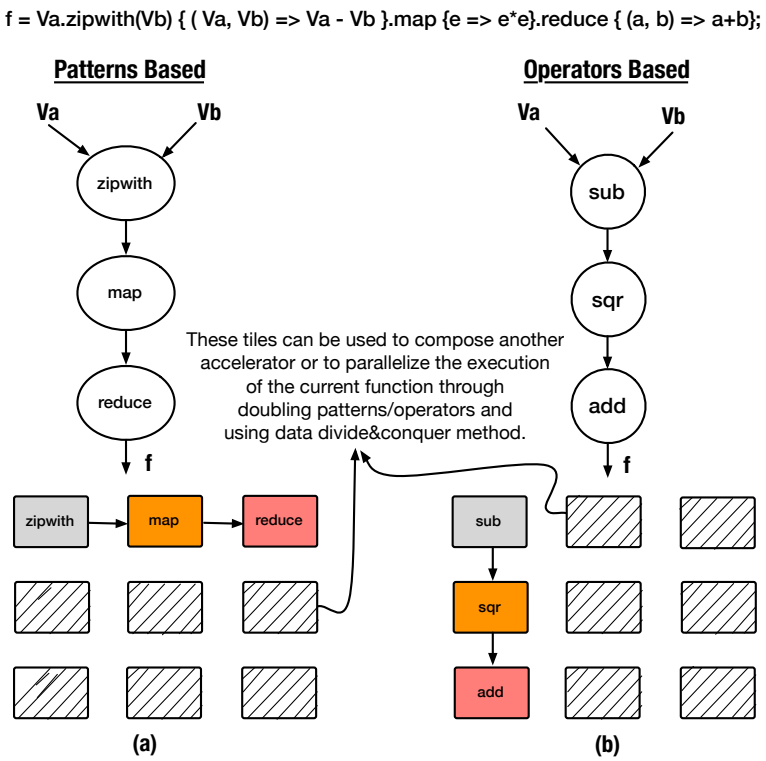


Figure 6.2: The DFG of CP1 function based on pre-synthesized (a) Patterns and (b) Operators, and the placement on 3×3 overlay.

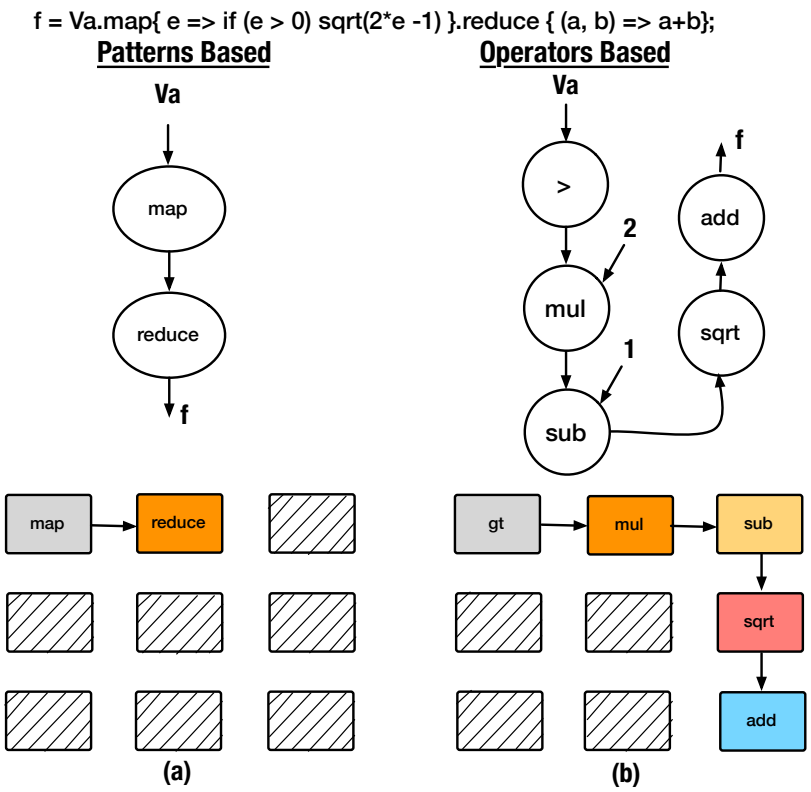


Figure 6.3: The DFG of CP2 function based on pre-synthesized (a) Patterns and (b) Operators, and the placement on 3×3 overlay.

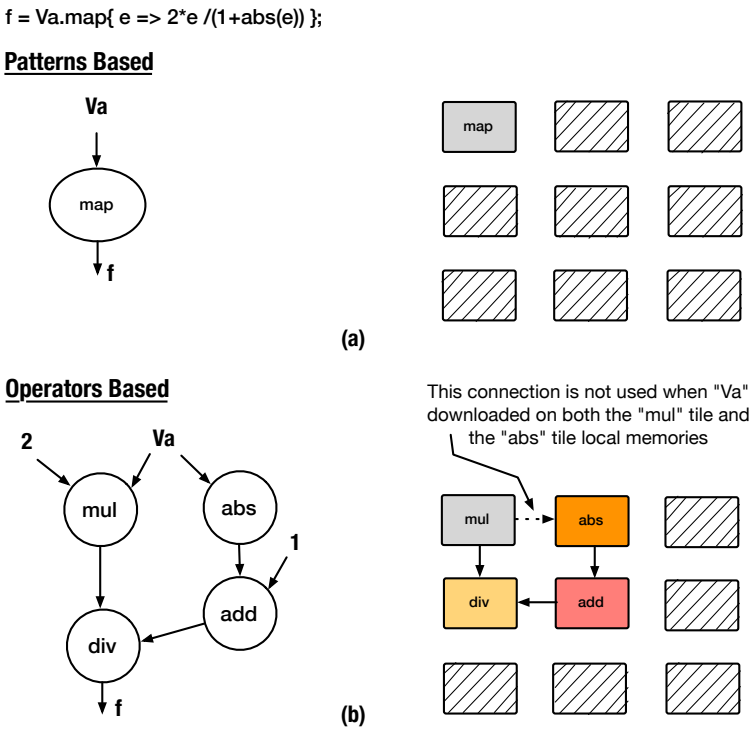


Figure 6.4: The DFG of CP3 function based on pre-synthesized (a) Patterns and (b) Operators, and the placement on 3×3 overlay.

6.5 Performance Evaluation

In addition to the HLS full accelerators and URUK overlay implementations, a software application for each function in the benchmark is created and run on the MicroBlaze. The software execution time will represent the common factor on the speed up formula of HLS accelerators and URUK. In this work, the speed up of the HLS accelerators represents the baseline in our comparisons. In URUK , the benchmark functions are implemented in two methods; pattern based and operator based. Initially, this section evaluates the performance of the pattern based URUK implementation and compares it with non-optimized HLS as well as the MicroBlaze. Additionally, it discusses the cost and the benefits of optimizing the execution of both URUK overlay and HLS. Next, it presents the performance of the optimized implementations. Then, a comparison between pattern based and operator based performance is provided to evaluate the advantages and disadvantages of using each one of them.

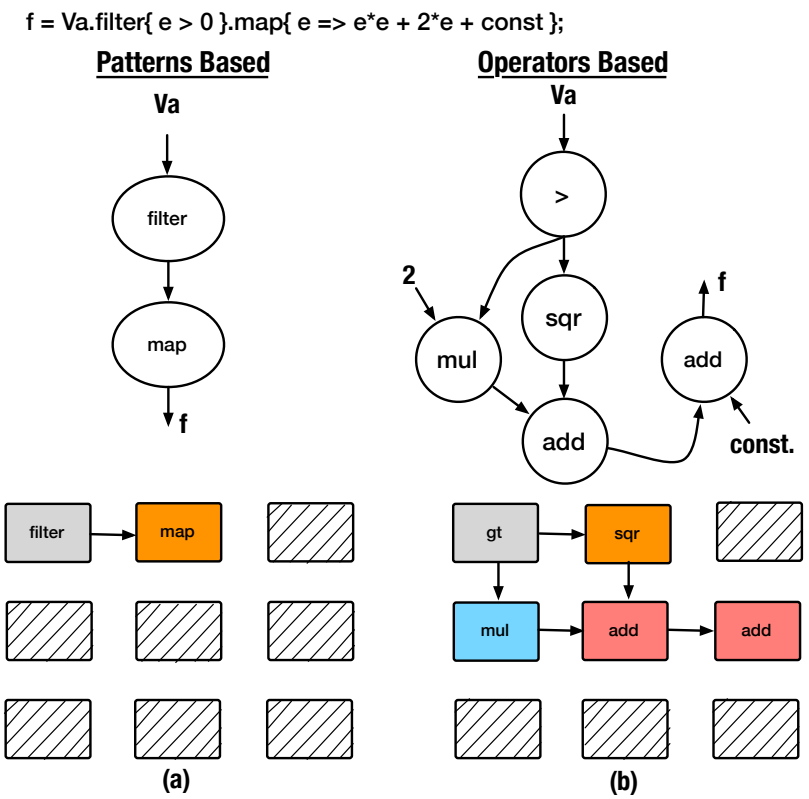
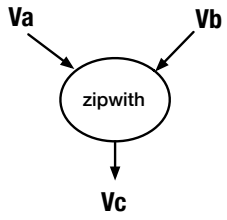
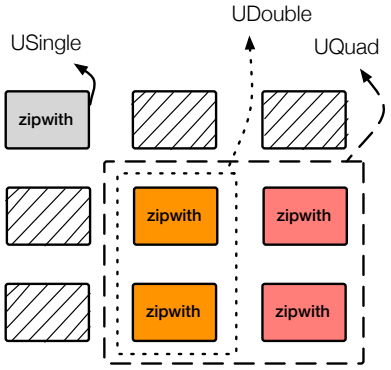


Figure 6.5: The DFG of CP1 function based on pre-synthesized (a) Patterns and (b) Operators, and the placement on 3×3 overlay.

Vector Add (VADD)

Vc = Va.zipwith(Vb) { (Va, Vb) => Va + Vb };

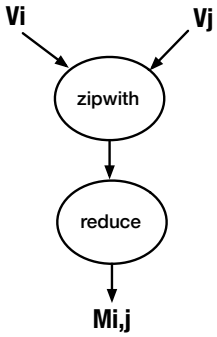


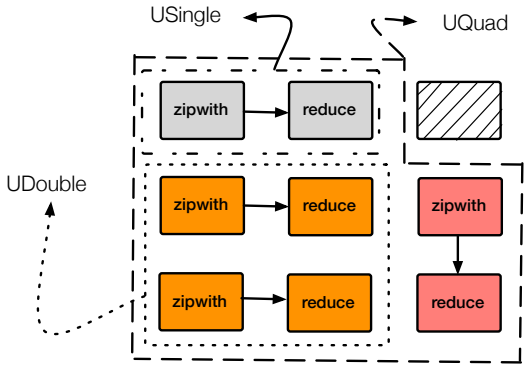


(a)

Matrix Multiply (MM)

Mi,j = Vi.zipwith(Vj) { (Vi, Vj) => Vi * Vj} .reduce {(a,b) => a+b)};





(b)

Composing Five Different Modules on One 3x3 Overlay

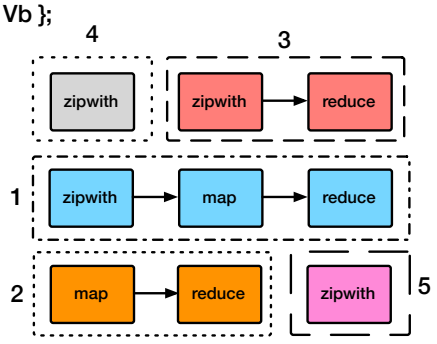
1 : f = Va.zipwith(Vb) { (Va, Vb) => Va - Vb }.map {e => e*e}.reduce { (a, b) => a+b};

2 : f = Va.map{ e => if (e > 0) sqrt(2*e -1) }.reduce { (a, b) => a+b};

3 : Mi,j = Vi.zipwith(Vj) { (Vi, Vj) => Vi * Vj} .reduce {(a,b) => a+b)};

4 : Vc = Va.zipwith(Vb) { (Va, Vb) => Va + Vb };

5 : Vc = Va.zipwith(Vb) { (Va, Vb) => Va * Vb };



(c)

Figure 6.6: Vector Addition and Matrix Multiplication DFGs, Pattern Based. Also, the three mapping methods, USingle, UDouble, and UQuad.

6.5.1 Pattern Based

In the URUK Pattern Based (UPB) implementation , a DFG for each function is created. Every node in the DFG represents a pattern. During the overlay setup time, patterns are mapped to the tiles matrix through downloading their partial bitstreams. Figures 6.6,6.2,6.3,6.4, and 6.5 show the pattern based DFGs and how they are mapped on the overlay.

Within URUK overlay, parallelism can be achieved by increasing the number of computational components (tiles) through mapping the same DFG patterns into multiple tiles. For instance, the vector addition function is represented in ”zipwith” pattern, which requires only one tile. However, it can be parallelized by downloading the ”zipwith” bitstream into multiple tiles and dividing the vectors' data among them, then gathering their results. The overlay allows doubling patterns as long as there are free available tiles. This feature provides an easy way to parallelize the execution and increase performance without the need of repeating synthesis, place and route. In contrast, the speed up does not increase linearly with the increase of used patterns. In this work, three mapping ways are used as follows:

- *USingle: mapping the DFG pattern(s) on the overlay tiles without doubling*
- *UDouble: mapping copies of the DFG pattern(s) on the overlay tiles*
- *UQuad: mapping four copies of the DFG pattern(s) on the overlay tiles*

Figure 6.6 presents mapping DFG patterns using USingle, UDouble, and UQuad. Figure 6.7 displays the execution time (ns) of the 7 benchmark applications using software versions on MicroBlaze, HLS full accelerators(non-optimized), and the three ways of pattern based URUK overlay implementations. The USingle achieved equivalent or less performance than the HLS implementation of the tested applications.

By comparing the execution time of the *USingle*, *UDouble*, and *UQuad*, we see that the speed up is not dropping linearly with the increase of computational components (tiles). The

nonlinear relationship is a result of the data transfer overhead which increments by increasing the number of tiles. For instance, when the *USingle* implements the *VADD* application which normally occupies one tile, the DMA makes two data transactions to the two tile's local BRAMs and one additional transaction to gather the results back to the main memory. However, when *UDouble* is used to implement the same application, it will occupy two tiles which require four DMA data transactions to the tiles' local BRAMs and two DMA transactions to move the results back to the main memory. With every DMA transaction, there is a constant DMA setup time which accumulates to be significant when multiple transactions of small data size occur. The DMA transactions increase with the number of utilized tiles. The CDMA engine takes around 177ns to transfer 4Kbytes (1K elements), 189ns to transfer 8Kbytes, and 209ns to transfer 16Kbytes.

The chart also shows that the benefit of using *UQuad* is proportional to the ratio between the processing time and the data transfer time (marked black in the chart) of the implemented application. For example, the *UQuad* implementation of CP2 achieved more than two times the speed up of the *USingle* implementation of the same application because the processing time is dominant and significant in that function compared to DMA time. The *UQuad* of *VADD* did not show much of a difference from *UDouble* in the execution time even though the processing time in the *UQuad* was decreased. That is because the DMA time in the *UQuad* was increased nearly by the same amount of the processing time decrease. The *UQuad* impact appears in the functions that have considerable processing time such as CP2, CP3 and matrix multiply.

The speed up of the implementations in Figure 6.7 is calculated based on the following formula:

$$Speedup = \frac{AcceleratorExecutionTime}{MicroBlazeExecutionTime}$$

Here, the *Accelerator* refers to the HLS accelerator as well as the URUK accelerators.
Figures 6.8, 6.9, and **??** display the calculated speed up of the HLS and URUK implementations. The speed up charts are related to the execution time in Figure 6.7. Since the computations of

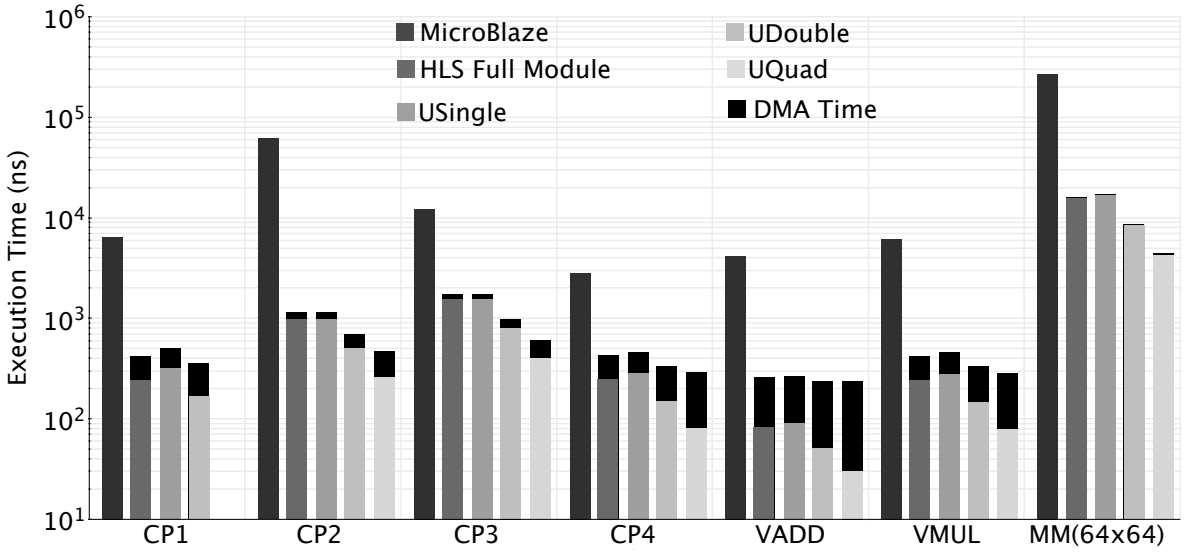


Figure 6.7: The execution time of the 7 benchmark functions. Only CP1 function was not implemented in UQuad due to the limited number of tiles. The HLS accelerators were not optimized. 4K data elements(32-bit integers) are used.

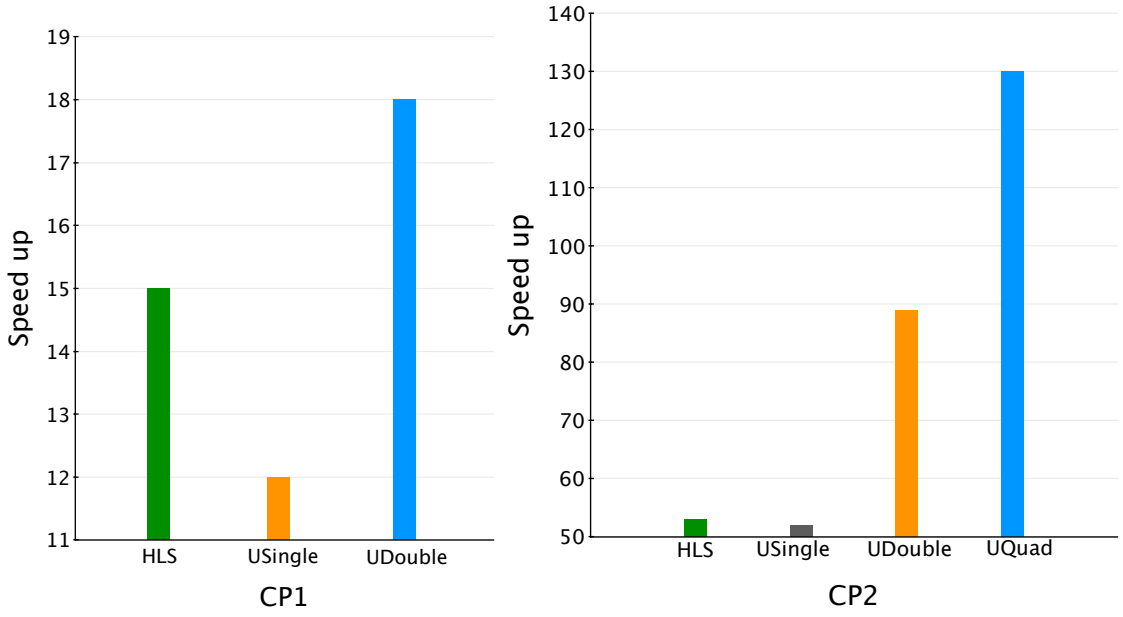
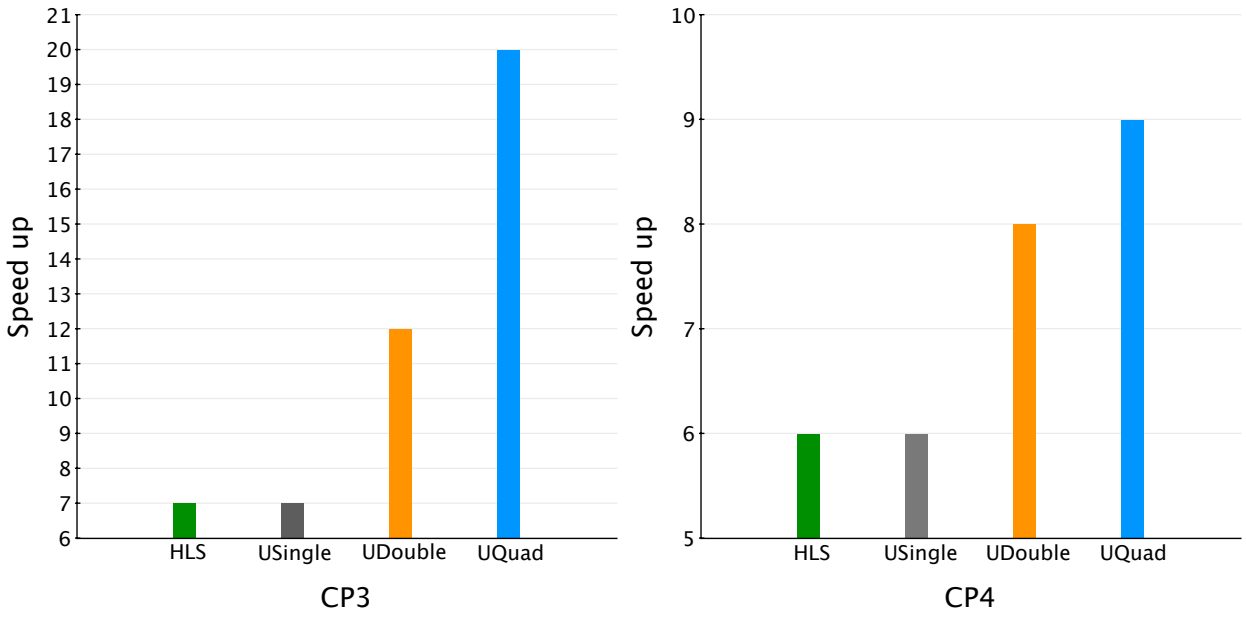
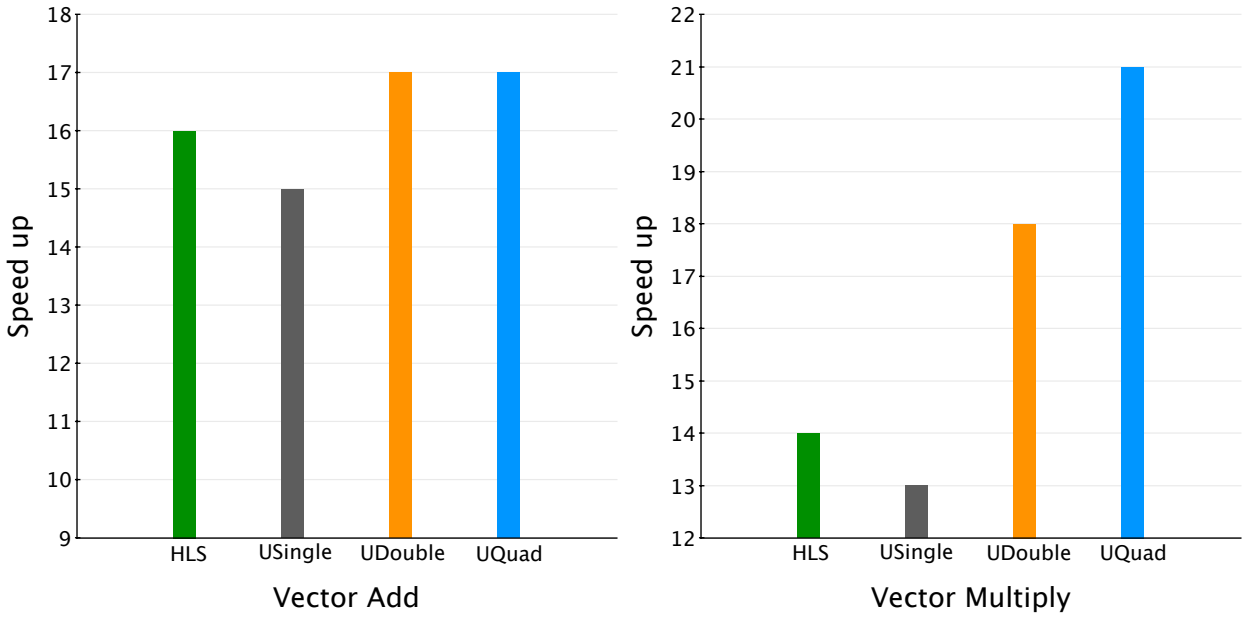


Figure 6.8: The speed ups of the HLS and URUK (USingle, UDouble, and UQuad) implementations over software versions of (a) CP1 and (b) CP2 functions.



(a) (b)

Figure 6.9: The speed ups of the HLS and URUK (USingle, UDouble, and UQuad) implementations over software versions of (a) CP3 and (b) CP4 functions.



(a) (b)

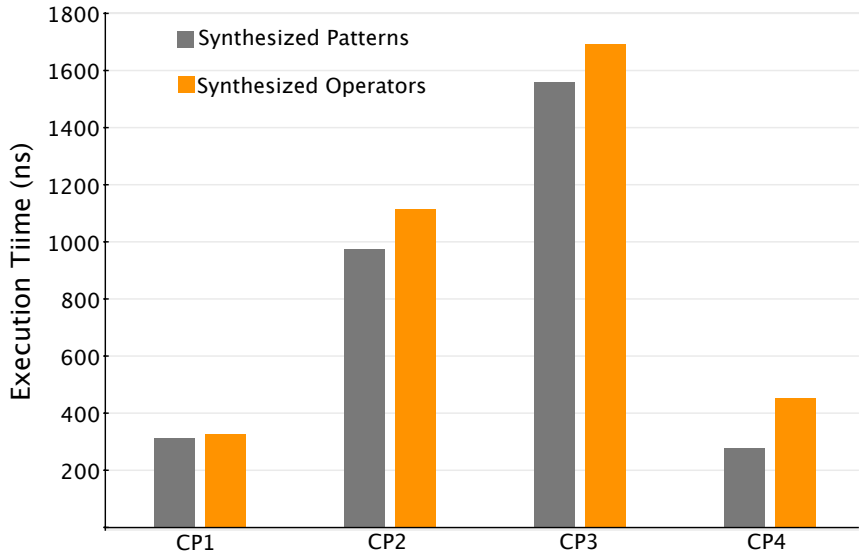
Figure 6.10: The speed ups of the HLS and URUK (USingle, UDouble, and UQuad) implementations over software versions of (a) Vector add and (b) Vector multiply functions.

CP2 , CP3 and *VMUL* are intensive compared to other functions in the benchmark, doubling the computational components (tiles), which increases the speed up by nearly two times the *USingle* implementation. The DMA time is not significant in these cases.

6.5.2 Operator Based

As discussed earlier, using pre-synthesized patterns requires multiple versions for each pattern to cover variant functionalities. For instance, the *map* pattern on Table 6.5 has several versions to implement various operations. In order to implement *map* with a new functionality, the *map* pattern should be synthesized and added to the library. Practically, this requires a big bitstream library in order to make the overlay capable of composing wide variety of accelerators. To overcome this problem, URUK overlay supports composing accelerators using pre-synthesized computational operators. In this section, we evaluate the URUK Operator Based (UOB) performance compared to the URUK Pattern Based (UPB) implementation.

Figures 6.2, 6.3, 6.4, and 6.5 show the Data Flow Graph (DFG) and tile utilizations of the UOB and UPB implementations for the CP1, CP2, CP3, and CP4 functions respectively. In the UOP, the overlay composes accelerators using pre-synthesized operators instead op patterns. Figure 6.11 presents the execution time (ns) and the speedups of both UOB and UPB implementations for the benchmark functions, CP1, CP2, CP3, and CP4. In this experiment, 4K data elements (32-bit integers) size is used. The chart demonstrates that using patterns provides better performance than using operators. In fact, patterns utilize fewer tiles than operators which leads to less communication overhead. Additionally, patterns can have many operations which are all placed in one PR region and routed internally in bit level, which optimizes the circuit and reduces latency. As a result, patterns can achieve more speed up than operators. However, the UOB provides more flexibility to implement a wide range of applications with few pre-synthesized operators.



	CP1		CP2		CP3		CP4	
	Pattrens	Operators	Pattrens	Operators	Pattrens	Operators	Pattrens	Operators
Speed up	12x	12x	53x	47x	7x	6x	6x	4x

Figure 6.11: Compares the execution time and speed up of UPB and UOB implementations using 4K data elements(32-bit integers).

6.5.3 Optimization

To compare the three methods of URUK overlay implementations with an optimized HLS accelerator, we created an optimized version of the matrix multiply using Vivado HLS. Pipeline directive was used in the optimized version. Figure 6.12 shows the 64x64 matrix multiply execution time (ns) using a software version on MicroBlaze, HLS (unoptimized), HLS (optimized), and the three ways of URUK overlay(*USingle*, *UDouble*, *UQuad*). The figure also displays the speed up of each method. The chart presents that the optimized version of HLS achieved around 96x speed up which is higher than the other implementations. The URUK *Quad* is similar to unrolling the inner loop by a factor of 4 in Vivado HLS. The *UQuad* achieved around 60x speed up, which is less than the HLS optimized version speed up. However, optimizing an HLS source code requires some hardware skills as well as repeating the synthesis, place, and route process, which are time consuming. In contrast, enhancing an application performance within the overlay through doubling

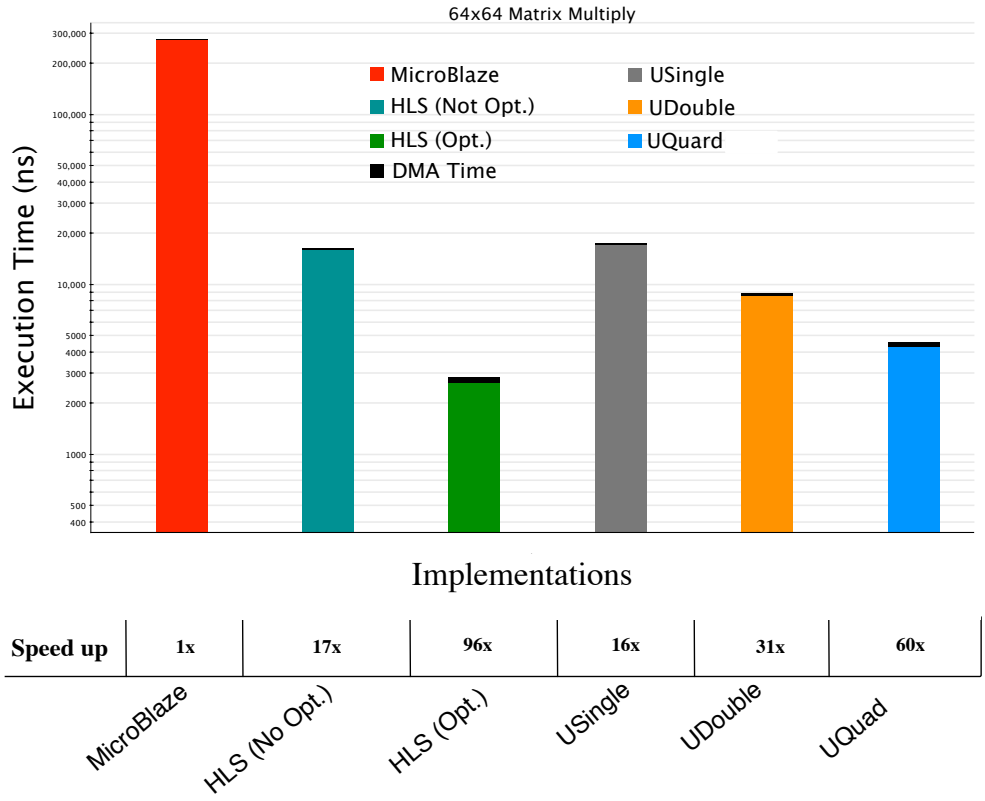


Figure 6.12: The execution time of the optimized and non-optimized HLS of 64x64 matrix multiply also the speed up compared to MicroBlaze and the three methods of URUK overlay implementations using 4K data elements(32-bit integers).

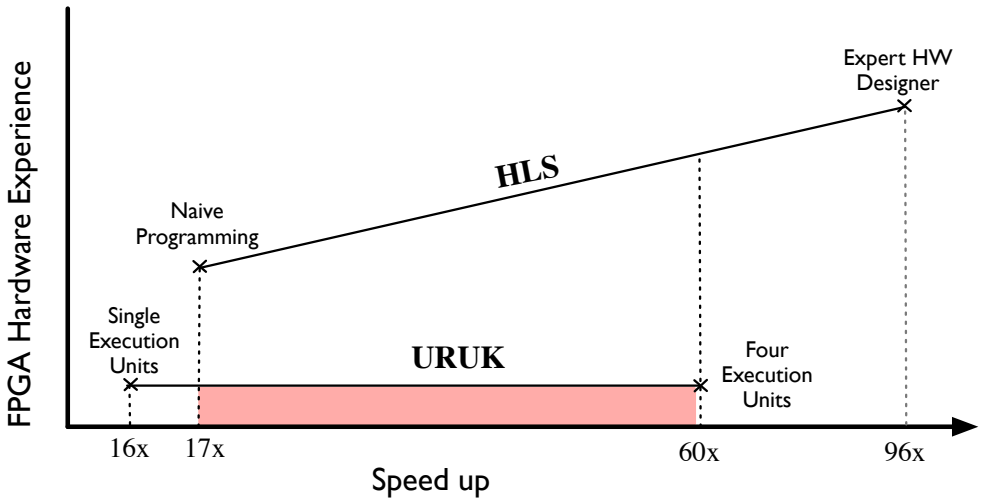


Figure 6.13: The required design experience on FPGA to optimize the 64x64 matrix multiply, in Figure 6.12, on both HLS and URUK to gain speed up.

tiles can be achieved without repeating synthesis and without hardware design skills. It only costs downloading more bitstreams and making minor changes in the system application level to divide and conquer data. This cost nothing compared to repeating synthesis.

Figure 6.13 shows a relation between design experience level and gaining speed up. Essentially, with little experience on FPGAs, one can optimize the implemented application on URUK to increase performance. From the optimized matrix multiply example, the programmer can gain 44 more speed up by only increasing the number of computational units to 4 and dividing the processing data among them. However, even a trivial implementation on HLS requires FPGA design skills to integrate the created accelerator into the system, and background knowledge on how to drive it. Moreover, optimizing a design on HLS requires more design skills and low level hardware details. On the other hand, the optimized designs on HLS gains more speed up, which presents tradeoffs between productivity and performance.

6.6 Productivity

To quantify the productivity, URUK is compared against HLS, which is currently considered the FPGAs' highest productive tools. Figure 6.14 shows the compilation steps of a design in Vivado HLS and URUK including the estimated time for each step. The left side of the figure shows the design compilation steps of Vivado HLS. The shown time may vary based on the design size. For instance, Vivado HLS takes around 30 seconds to generate a matrix multiply IP. Then Vivado System takes a round 16 minutes to synthesis, place, route, and generate the bitstream for the system, which includes the IP of matrix multiply with a MicroBlaze. When the system is large, Vivado may take hours to place and route the design. This is the most extensive step in the system design. Further, the time here is not including the designer work time on building the system as well as integrating the IP into the system. This by itself requires high level of experience on system design, interfacing, and connections. After generating the bitstream, the design should be exported to the System Development Kit (SDK) in order to implement the application. In the SDK,

applications go through a normal software compilation process.

In the HLS design compilation flow, the real challenge is that any minor changes within the design source code requires repeating the whole flow processes including the Vivado system process. In fact, the HLS increases the designers' productivity only for the part where they represents their algorithms in a high level language instead of using one of the hardware description languages. However, the generated IP from the HLS must be integrated into the system and synthesized within the Vivado system, which is the most time consuming stage in the CAD tool flow. Therefor, to achieve equivalent productivity to normal software, we moved this stage from design compilation in URUK by using pre-synthesized parallel-data patterns and computational operators.

The right side of Figure 6.14 presents URUK compilation steps of a design. Since the overlay is constructed as a coarse-grind, the place and route search space is very small compared to the fine-grind. Thus, the DSL compilation time is expected to be fast, a few seconds. The generated function calls and tile binaries are executed directly by the host processor, the MicroBlaze in this work. When the MicroBlaze executes the function calls, it downloads the partial bitstreams to the specified tiles. The download time depends on the size of PR regions and the FPGA. The total download time vary between 2 seconds to milliseconds. It took around 2.37ms to download and compose the partial bitstreams of the matrix multiplay patterns.

In summary, URUK can enable software programmers without any hardware skills to create hardware accelerators at productivity levels consistent with software development and compilation.

6.7 Dynamic vs. Static

Since URUK is using Partial Reconfiguration (PR) technique, it can assign (place) computational operators dynamically. This feature provides high flexibility in placing operators and routing data between them. To evaluate the cost and benefits of this feature, we designed a static overlay, which has a similar structure to the original URUK overlay except no PR regions. In the static overlay,

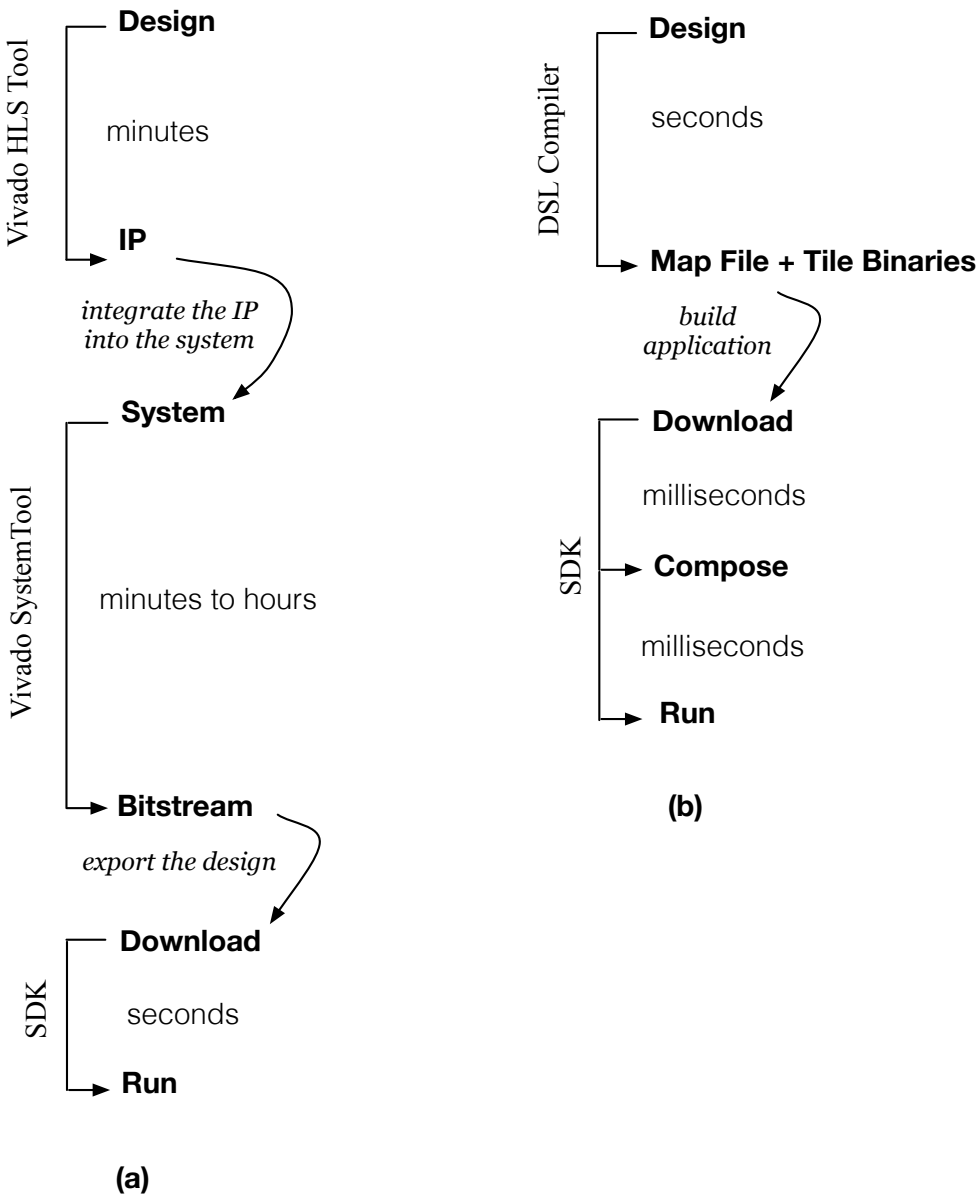


Figure 6.14: (a) The HLS design steps including estimated time; (b) URUK compilation steps including estimated time.

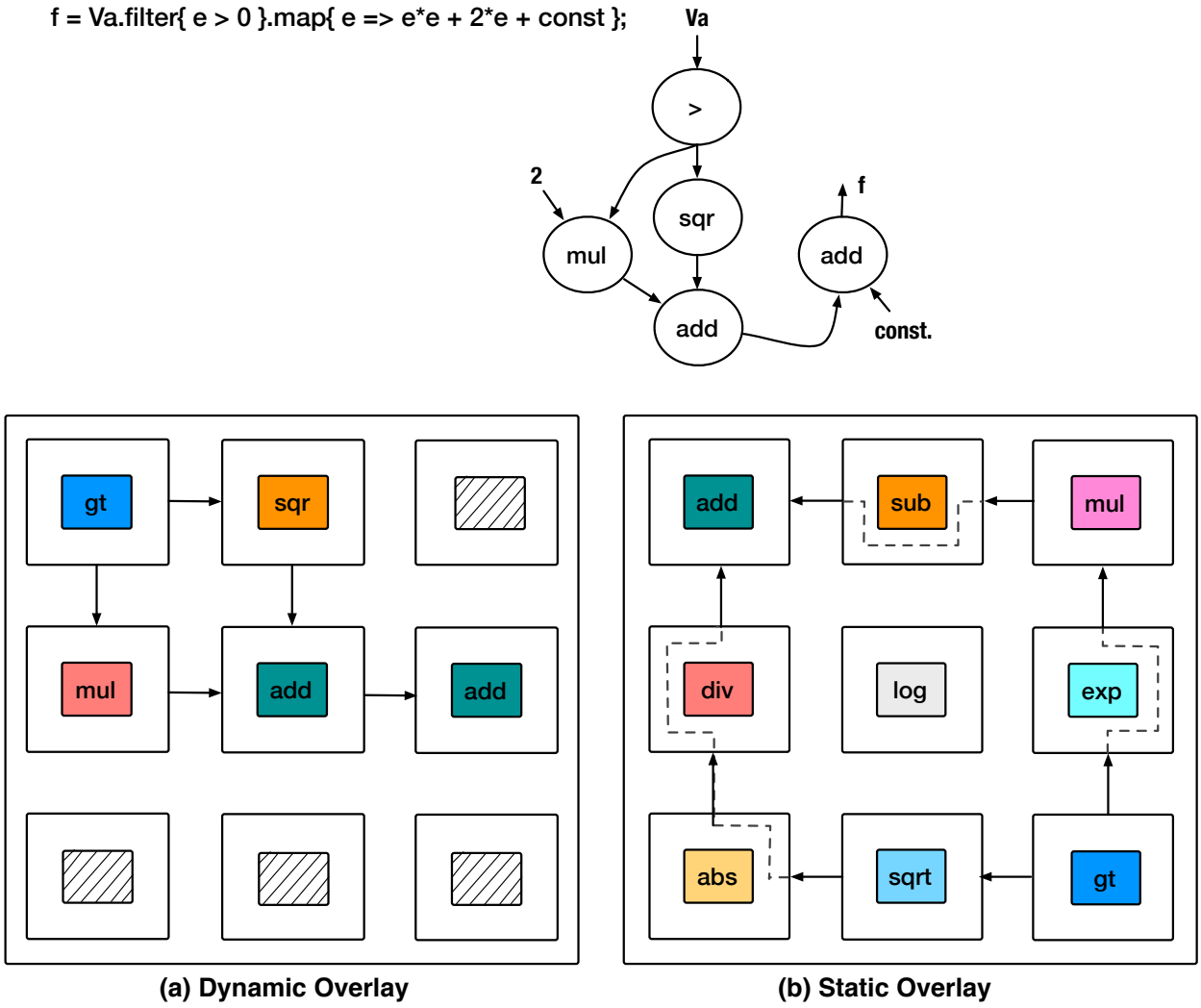


Figure 6.15: Dynamic vs. static overlays

instead of using PR regions, each tile has an non-replaceable computational operator, synthesized as part overlay static logic. URUK is compared against the static overlay in three different parts as follows:

6.7.1 Area

Tiles in the static overlay are naturally different in size because every tile is statically holding a different operator(small or large). In contrast, tiles in the URUK overlay have the same size when PR regions are set to the largest operator. Thus, resources internal fragmentations appear in the URUK overlay tiles when small operators are placed in large PR regions. As a result, the overall area usage of URUK overlay is higher than the static one. This is a cost of being dynamic. Further optimizations can be made within URUK overlay to reduce the resources internal fragmentations such as setting PR regions to variant sizes.

6.7.2 Routing Data

Figure 6.15 presents example of implementing CP4 function on a 3×3 URUK overlay and a a 3×3 static overlay. Both overlays can route data between tiles using same configurations. However, URUK has the flexibility of placing operators as close as possible tiles at accelerator's setup time to reduce communication overhead. While, the static overlay does not have this flexibility at setup time, and the communication overhead depends on how operators are distributed among tiles at design time. Each tile interconnect in both overlays has a latency of 2 clock cycles to transfer a data element. In this study, operators are distributed arbitrarily in the static overlay, which can be optimized with better organizations. Therefor, the shown results are not standard in this case of the static overlay.

Figure 6.15 (a) shows that dependent operators are placed near each other without data hoping throughout tiles. In contrast, the static organization of operators on the right side of the figure forces the data to be bypassed though other tiles in order to reach the destination tile. For example, the

output of square root (sqrt) tile has to pass through two other tiles to get to the destination, add tile. Likewise, the output of the comparator (gt) has to hop one tile to get to the multiply (mul) tile.

The execution time of URUK using USingle is around 452ns; while the execution time of the static overlay is around 764ns for the same function, CP4. The noticeable difference between the performance of URUK and the static overlay is due to two reasons. First, the four hops in the static overlay are making incremental latency. Second, the *add* tile handles two separate add operations without taking the advantage of streaming. Contrarily, URUK uses two *add* tiles to benefit from streaming.

6.7.3 Parallelism

As shown previously, parallelism can be achieved by instantiating additional copies of the application throughout additional tiles as presented in using *UDouble* and *UQuad*. In contrast, It is complicated to be achieved in the static overlay for some reasons. First, the number of tiles should be increased by the number of additional copies of the application during design time. Since tiles' operators are irreplaceable, the overlay size should be increased by the number of copies. Second, performance will not be guaranteed because it depends on the operators organization among tiles.

Chapter 7

Conclusion

7.1 Summary

To summarize, this thesis has investigated the following questions:

Can URUK eliminate the challenges that result from composing pre-synthesized parallel patterns while still preserving all the productivity benefits of the original JIT approach? URUK eliminated the challenges, all variant parallel programming patterns must be pre-synthesized, by composing the fundamental pre-synthesized operators of the non synthesized patterns. The presented solution achieved not only the same productivity benefits but also more flexibility in constructing a wide variety of accelerators. The results show also that this solution is valid with some sacrifice in tile utilizations and performance.

Can URUK allow conditionals to be composed with the synthesized programming patterns without generating multiple bitstreams for each case? The URUK overlay structure provided a mechanism with instruction sets to handle conditional operations, which are used to prevent (in some cases) composing patterns in the original JITA approach. Three of the test benchmark functions (CP1, CP2, and CP4) include conditional operation to validate the solution.

How much time does it take to construct an accelerator using the new compilation flow targeting URUK compared to Vivado HLS? Constructing an accelerator URUK takes a few seconds due to the extremely small search space of tiles' place and route compared to the HLS flow, which has a large find-grind search space. The system synthesis, place and route are moved out of the programmers' path. Therefore, URUK can enable software programmers without any hardware

skills to create hardware accelerators at productivity levels consistent with software development and compilation.

How will performance and resource utilization be effected compared to full custom designed modules using Vivado HLS as well as the original JIT approach? The results show that URUK can achieve equivalent or higher performance when the HLS design is not optimized. Contrarily, the HLS provides higher performance, around 36x, when it is optimized by an expert designer. The HLS also presents better resource utilization in all examples. However, this is the penalty of achieving software level productivity.

What are the costs and benefits of considering Partial Reconfiguration techniques as part of the overlay dynamic system? URUK is compared against a static overlay to evaluate the cost and benefits of using Partial Reconfiguration techniques. The static overlay occupies less area than URUK, which is dynamic. On the other hand, URUK presented high flexibility in placing dependent operators as close as possible and routing data between them. The static overlay was limited by the operators' organization at the overlay design time. Additionally, parallelism can be easily implemented with the URUK overlay; while parallelism is costly and hard to be achieved in the static one.

7.2 Future Work

The presented overlay architecture and the compilation flow hold potential for additional constructions and several other optimizations, which can be considered in the future. First, a DSL compiler need to be developed by integrating a backend generator into the Delite Framework or similar platform. This stage includes several critical steps to enhance the overlay performance and utilizations. For instance, integrating a search algorithm for efficient place and route.

Second, the diminution of the overlay can be increased and distributed across multiple FP-

GAs for large scale computations. This will allow achieving high parallelism and optimizing the performance especially when the computational task is extensive, and the data set is very large.

Third, to optimize the resource utilization of the overlay, tiles' PR regions can be set to variant sizes. This will reduce the resource's internal fragmentations. Further, since the overlay is dynamic, it is crucial to explore other interconnect topologies, which may provide more efficient overlay structure.

References

[1] Jason Agron. Domain-Specific Language for HW/SW Co-design for FPGAs. In *DSL*, volume 5658 of *Lecture Notes in Computer Science*, pages 262–284. Springer, 2009.

[2] Zeyad Aklah and David Andrews. *A Flexible Multilayer Perceptron Co-processor for FPGAs*, pages 427–434. Springer International Publishing, Cham, 2015.

[3] Zeyad Aklah, Sen Ma, and David Andrews. A dynamic overlay supporting just-in-time assembly to construct customized hardware accelerators. *CoRR*, abs/1603.01187, 2016.

[4] Alterra. Nios II Processor Reference Handbook. http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf. Last accessed May 3, 2017.

[5] F. Anjam, M. Nadeem, and S. Wong. A vliw softcore processor with dynamically adjustable issue-slots. In *2010 International Conference on Field-Programmable Technology*, pages 393–398, Dec 2010.

[6] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM.

[7] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[8] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[9] A. Becker, S. Sirowy, and F. Vahid. Just-in-time compilation for fpga processor cores. In *Electronic System Level Synthesis Conference (ESLsyn), 2011*, pages 1–6, June 2011.

[10] A. Brant and G.G.F. Lemieux. Zuma: An open fpga overlay architecture. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 93–96, April 2012.

[11] Gordon J. Brebner and Weirong Jiang. High-speed packet processing using reconfigurable computing. *IEEE Micro*, 34(1):8–18, 2014.

[12] K.J. Brown, A.K. Sujeeth, Hyouk Joong Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 89–100, Oct 2011.

[13] Stuart Byma, J.Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 109–116, May 2014.

[14] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2):24:1–24:27, September 2013.

[15] D. Capalija and T. S. Abdelrahman. Towards synthesis-free jit compilation to commodity fpgas. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 202–205, May 2011.

[16] D. Capalija and T.S. Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8, Sept 2013.

[17] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell. idea: A dsp block based fpga soft processor. In *2012 International Conference on Field-Programmable Technology*, pages 151–158, Dec 2012.

[18] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. 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, April 2011.

[19] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Charm: A composable heterogeneous accelerator-rich microprocessor. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 379–384, New York, NY, USA, 2012. ACM.

[20] J. Coole and G. Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 13–22, Oct 2010.

[21] J. Coole and G. Stitt. Adjustable-cost overlays for runtime compilation. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 21–24, May 2015.

[22] CRA. Revitalizing Computer Architecture Research.

[23] R. Dimond, O. Mencer, and W. Luk. Custard - a customisable threaded fpga soft processor and tools. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 1–6, Aug 2005.

[24] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM.

[25] Carl Ebeling, Carl Ebeling, Darren C. Cronquist, Darren C. Cronquist, Paul Franklin, Paul Franklin, Chris Fisher, and Chris Fisher. Rapid - a configurable computing architecture for compute-intensive applications, 1996.

[26] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual*

international symposium on Computer architecture, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[27] Daniel D. Gajski, Jianwen Zhu, Rainer Dmer, Andreas Gerstlauer, and Shuqing Zhao. *SPECC: Specification Language and Methodology*. Springer US, 1997.

[28] N. George, Hyoukjoong Lee, D. Novo, T. Rompf, K.J. Brown, A.K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[29] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. Piperench: a reconfigurable architecture and compiler. *Computer*, 33(4):70–77, Apr 2000.

[30] V. Govindaraju, Chen-Han Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and Changkyu Kim. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *Micro, IEEE*, 32(5):38–51, Sept 2012.

[31] http://www.bls.gov/ooh/Computer-and Information-Technology/. Occupational Outlook Handbook.

[32] T. Kranenburg and R. van Leuken. Mb-lite: A robust, light-weight soft-core implementation of the microblaze architecture. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 997–1000, March 2010.

[33] David Ku and Giovanni DeMicheli. Hardwarec – a language for hardware design (version 2.0). Technical report, Stanford, CA, USA, 1990.

[34] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, 2(2):135–253, February 2008.

[35] Charles Eric LaForest and John Gregory Steffan. Octavo: An fpga-centric processor family. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 219–228, New York, NY, USA, 2012. ACM.

[36] C.Y. Lee. An algorithm for path connections and its applications. *Electronic Computers, IRE Transactions on*, EC-10(3):346–365, Sept 1961.

[37] C. Liu, H. C. Ng, and H. K. H. So. Quickdough: A rapid fpga loop accelerator design framework using soft cgra overlay. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 56–63, Dec 2015.

[38] C. Liu, C. L. Yu, and H. K. H. So. A soft coarse-grained reconfigurable array based high-level synthesis methodology: Promoting design productivity and exploring extreme fpga frequency. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 228–228, April 2013.

[39] Xiaobin Liu. Energy Efficient Exploration of Coarse-Grain Reconfigurable Architecture With Emerging . Master's thesis, University of Massachusetts Amherst, USA, 2015.

[40] R. Lysecky, F. Vahid, and S. D. X. Tan. A study of the scalability of on-chip routing for just-in-time fpga compilation. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 57–62, April 2005.

[41] Roman Lysecky, Kris Miller, Frank Vahid, and Kees Vissers. Firm-core virtual fpga for just-in-time fpga compilation (abstract only). In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05, pages 271–271, New York, NY, USA, 2005. ACM.

[42] Roman Lysecky, Frank Vahid, and Sheldon X.-D. Tan. Dynamic fpga routing for just-in-time fpga compilation. In *Proceedings of the 41st Annual Design Automation Conference*, DAC '04, pages 954–959, New York, NY, USA, 2004. ACM.

[43] S. Ma, Z. Aklah, and D. Andrews. A run time interpretation approach for creating custom accelerators. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2015.

[44] S. Ma, Z. Aklah, and D. Andrews. Run time interpretation for creating custom accelerators. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 900–905, March 2016.

[45] Sen Ma, Zeyad Aklah, and David Andrews. Just in time assembly of accelerators. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 173–178, New York, NY, USA, 2016. ACM.

[46] Alan Marshall, Tony Stansfield, Igor Kostarnov, Jean Vuillemin, and Brad Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays*, FPGA '99, pages 135–143, New York, NY, USA, 1999. ACM.

[47] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE Proceedings - Computers and Digital Techniques*, 150(5):255–61–, Sept 2003.

[48] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Dresc: a retargetable compiler for coarse-grained reconfigurable architectures. In *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.*, pages 166–173, Dec 2002.

[49] BingfengMei2003 Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix*, pages 61–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[50] Jiayuan Meng, Xingfu Wu, Vitali Morozov, Venkatram Vishwanath, Kalyan Kumaran, and Valerie Taylor. Skope: A framework for modeling and exploring workload behavior. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 6:1–6:10, New York, NY, USA, 2014. ACM.

[51] Michael Metzner, Jesus A. Lizarraga, and Christophe Bobda. *Architecture Virtualization for Run-Time Hardware Multithreading on Field Programmable Gate Arrays*, pages 167–178. Springer International Publishing, Cham, 2015.

[52] E. Mirsky and A. DeHon. Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, Apr 1996.

[53] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, May 2015.

[54] S. Shukla, N. W. Bergmann, and J. Becker. Quku: a two-level reconfigurable architecture. In *IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures (ISVLSI'06)*, pages 6 pp.–, March 2006.

[55] S. Shukla, N. W. Bergmann, and J. Becker. Quku: A fpga based flexible coarse grain architecture design paradigm using process networks. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–7, March 2007.

[56] H. Singh, Ming-Hau Lee, Guangming Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. C. Filho. Morphosys: a reconfigurable architecture for multimedia applications. In *Proceedings. XI Brazilian Symposium on Integrated Circuit Design (Cat. No.98EX216)*, pages 134–139, Sep 1998.

[57] Kyle L. Spafford and Jeffrey S. Vetter. Aspen: A domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 84:1–84:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[58] Stanford-PPL. OptiML. http://stanford-ppl.github.io/Delite/optiml/examples.html. Last accessed May 3, 2017.

[59] Arvind K. Sujeeth, Hyoukjoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. Optiml: an implicitly parallel domainspecific language for machine learning. In *in Proceedings of the 28th International Conference on Machine Learning, ser. ICML*, 2011.

[60] Farooq U, Marrakchi Z, and Mehrez H. *FPGA architectures: An overview. In: Tree-based Heterogeneous FPGA Architectures*, pages 7–48. Springer, New York, 2012.

[61] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, pages 127–134, May 2010.

[62] Lu Wan, Chen Dong, and Deming Chen. A coarse-grained reconfigurable architecture with compilation for high performance. *International Journal of Reconfigurable Computing*, pages 1–17, 2012.

[63] T. Wiersema, A. Bockhorn, and M. Platzner. Embedding fpga overlays into configurable systems-on-chip: Reconos meets zuma. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6, Dec 2014.

[64] J.G. Wingbermuehle, R.D. Chamberlain, and R.K. Cytron. Scalapipe: A streaming application generator. In *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on*, pages 44–53, July 2012.

[65] Xilinx. MicroBlaze Processor Reference Guide. http://bit.ly/1xFtH8q. Last accessed May 3, 2017.

[66] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors. In *CASES '08: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 61–70, New York, NY, USA, 2008. ACM.

[67] Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector Processing as a Soft-Core CPU Accelerator. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 222–232, New York, NY, USA, 2008. ACM.

[68] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. *AutoPilot: A Platform-Based ESL Synthesis System*, pages 99–112. Springer Netherlands, Dordrecht, 2008.