2019

Towards hardware as a reconfigurable, elastic, and specialized service

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

TOWARDS HARDWARE AS A RECONFIGURABLE, ELASTIC, AND SPECIALIZED SERVICE

by

AHMED SANAULLAH

B.S., Lahore University of Management Sciences, 2013
M.Sc., University of Nottingham, 2014

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2019

Approved by

First Reader

Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

Second Reader

Orran Krieger, PhD
Professor of Electrical and Computer Engineering

Third Reader

Ayse K. Coskun, PhD
Associate Professor of Electrical and Computer Engineering

Fourth Reader

Jonathan Appavoo, PhD
Associate Professor of Computer Science

Always a better way

Acknowledgments

To my advisor and mentors for their invaluable guidance and unwavering support, to my friends and colleagues for their help and encouragement, and to my family for being my pillars of strength: I am utterly grateful to all of you for being part of this incredible journey.

Ahmed Sanaullah
PhD Candidate
ECE Department

TOWARDS HARDWARE AS A RECONFIGURABLE, ELASTIC, AND SPECIALIZED SERVICE

AHMED SANAULLAH

Boston University, College of Engineering, 2019

Major Professor: Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

ABSTRACT

As modern Data Center workloads become increasingly complex, constrained, and critical, mainstream *CPU-centric* computing has had ever more difficulty in keeping pace. Future data centers are moving towards a more fluid and heterogeneous model, with computation and communication no longer localized to commodity CPUs and routers. Next generation *data-centric* Data Centers will *compute everywhere*, whether data is stationary (e.g. in memory) or on the move (e.g. in network). While deploying FPGAs in NICS, as co-processors, in the router, and in Bump-in-the-Wire configurations is a step towards implementing the data-centric model, it is only part of the overall solution. The other part is actually leveraging this reconfigurable hardware. For this to happen, two problems must be addressed: *code generation* and *deployment generation*. By code generation we mean transforming abstract representations of an algorithm into equivalent hardware. Deployment generation refers to the runtime support needed to facilitate the execution of this hardware on an FPGA.

Efforts at creating supporting tools in these two areas have thus far provided limited benefits. This is because the efforts are limited in one or more of the following ways: They i) do not provide fundamental solutions to a number of challenges, which

makes them useful only to a limited group of (mostly) hardware developers, ii) are constrained in their scope, or iii) are *ad hoc*, i.e., specific to a single usage context, FPGA vendor, or Data Center configuration. Moreover, efforts in these areas have largely been mutually exclusive, which results in incompatibility across development layers; this requires wrappers to be designed to make interfaces compatible. As a result there is significant complexity and effort required to code and deploy efficient custom hardware for FPGAs; effort that may be orders-of-magnitude greater than for analogous software environments.

The goal of this dissertation is to create a framework that enables reconfigurable logic in Data Centers to be targeted with the same level of effort as for a single CPU core. The underlying mechanism to this is a framework, which we refer to as *Hardware as a Reconfigurable, Elastic and Specialized Service* or HaaRNESS. In this dissertation, we address two of the core challenges of HaaRNESS: reducing the complexity of code generation by constraining High Level Synthesis (HLS) toolflows, and replacing ad hoc models of deployment generation by generalizing and formalizing what is needed for a hardware Operating System. These parts are unified by the back-end of HLS toolflows which link generated compute pipelines with the operating system, and provide appropriate APIs, wrappers, and software runtimes.

The contributions of this dissertation are the following: i) an empirically guided set of systematic transformations for generating high quality HLS code; ii) a framework for instrumenting HLS compiler to identify and remove optimization blockers; iii) a framework for RTL simulation and IP generation of HLS kernels for rapid turnaround; and iv) a framework for generalization and formalization of hardware operating systems to address the *ad hoc*'ness of existing deployment generation and ensure uniform structure and APIs.

Contents

List of Tables

List of Figures

List of Abbreviations

λ	.....	Multi Dimension Eigenvalues
A	.....	Transformation Vector
v	.....	Multi Dimension Eigenvectors
AES	.....	AdvancedEncryptionStandard
ALM	.....	Adaptive Logic Module
API	.....	Application Programming Interface
APP	.....	Application
ARP	.....	Address Resolution Protocol
ASIC	.....	Application Specific Integrated Circuit
AVMM	.....	Avalon Memory Mapped
AWS	.....	Amazon Web Services
BitW	.....	Bump in the Wire
BO-DFG	.....	Bitwise-Only Data Flow Graph
BRAM	.....	Block Random Access Memory
BSP	.....	Board Support Package
CAM	.....	Content Addressable Memory
CL	.....	Custom Logic
CPU	.....	Central Processing Unit
CRC	.....	Cyclic Redundancy Check
cuDNN	.....	CUDA Deep Neural Network Library
DDR	.....	Double Data Rate
DFG	.....	Data Flow Graph
DIP	.....	Dual In-Line Package
DMA	.....	Direct Memory Access
DNN	.....	Deep Neural Network
DRAM	.....	Dynamic Random Access Memory
DSL	.....	Domain Specific Language
DSP	.....	Digital Signal Processing
ER	.....	Elastic Router
FFT	.....	Fast Fourier Transform
FIFO	.....	First In First Out
FixOP	.....	Fixed Point Operation
FLOP	.....	Floating Point Operation
FOI	.....	Function of Interest

FP	.....	Floating Point
FPGA	.....	Field Programmable Gate Array
FST	.....	FPGA Specific Transformations
GPGPU	.....	General Purpose Graphics Processing Unit
GPU	.....	Graphics Processing Unit
GRU	.....	Gated Recurrent Unit
HaaRNESS	.....	Hardware as a Reconfigurable, Elastic and Specialized Service
HBM	.....	High Bandwidth Memory
HDL	.....	Hardware Description Language
HLL	.....	High Level Language
HLS	.....	High Level Synthesis
IB	.....	Input Bridge
IBP	.....	Intel Best Practices
ID	.....	Input Demultiplexer
ID	.....	Identification
II	.....	Initiation Interval
IP	.....	Intellectual Property
IP	.....	Internet Protocol
IR	.....	Intermediate Representation
JTAG	.....	Joint TestAction Group
LCM	.....	Least Common Multiple
LED	.....	Light Emitting Diode
LOC	.....	Lines of Code
LSTM	.....	Long Short-Term Memory
LSU	.....	Load Store Unit
LTL	.....	Lightweight Transport Layer
MAC	.....	Media AccessControl
MGT	.....	Management
MIMD	.....	Mulitple Instruction Multiple Data
MLAB	.....	Memory Logic Array Block
MLP	.....	Multi Layer Perceptron
MMM	.....	Matrix Matrix Multiplication
MPSoC	.....	Multi Processor System on Chip
MSB	.....	Most Significant Bit
MUX	.....	Multiplexer
MVM	.....	Matrix Vector Multiplication
NIC	.....	Network Interface Card
OpenCL	.....	Open Computing Language
OS	.....	Operating System
OSI	.....	Open Systems Interconnection

OUF	.....	Other User Functions
PA	.....	Payload Alignment
PCIe	.....	Peripheral Component Interconnect express
PCP	.....	Persistent Critical Path
PF	.....	Packet Formaters
PF	.....	Physical Function
PHY	.....	Physical Layer
PME	.....	Particle Mesh Ewald
PR	.....	Partial Reconfiguration
PTR	.....	Packet Transmitter Receiver
QPI	.....	Quick Path Interconnect
RAW	.....	Read After Write
Regex	.....	Regular Expression Match
ReLU	.....	Rectified Linear Unit
RHOS	.....	Reconfigurable Hardware Operating System
RTL	.....	Register Transfer Language
SAS	.....	Serial Attached SCSI
SAVI	.....	Smart Applications on Virtual Infrastructure
SDFG	.....	Stateful DataFlow multiGraph
SDK	.....	Software Development Kit
SHA	.....	Secure Hash Algorithm
SIMD	.....	Single Instruction Multiple Data
SL3	.....	SerialLite III
SM	.....	Streaming Multiprocessor
SOP	.....	Start of Packet
SpMV	.....	Sparse Matrix Dense Vector Multiplication
SSA	.....	Single Static Assignment
TCP	.....	Transmission Control Protocol
ToR	.....	Top of Rack
UART	.....	Universal Asynchronous Receiver/Transmitter
UCT	.....	Universal Code Transformation
UDP	.....	User Datagram Protocol
VCP	.....	Variable Critical Path
VFR	.....	Virtualized FPGA Resource

Chapter 1

Introduction

1.1 Motivation

The proliferation of digitization in all aspects of life, including healthcare, finance, education, and governance, has resulted in massive quantities of data being produced at unprecedented rates; everything we do generates some form of data, and everything we don't do does as well. Simultaneously, the requirements of processing complexity and performance have become increasingly stringent in order to maximize utility of this data. Simply put, we are being overwhelmed by the aggregate demand for compute resources to effectively process all this data. The supply of Data Centers has consequently grown and thousands of them have been deployed worldwide as shown in Figure 1·1. These Data Centers provide massive amounts of globally accessible compute and storage resources, connected over high speed commodity networks.

While it is possible that the aggregate compute capability available today is sufficient for us to process all data effectively, practical limitations mean that we are unable to do so. We are on the tail end of utilization of our compute resources, with an average of less than 1% of total computing cycles spent on doing useful work. This is shown in Figure 1·2. With a few hundred instructions possible per byte of data, only trivial operations can be performed if all data is to be processed. On the other hand, more complex processing would require a substantial fraction of data to be dropped in order to prevent buildup of a backlog.

The low utilization observed in Data Centers is because the evolution of Data

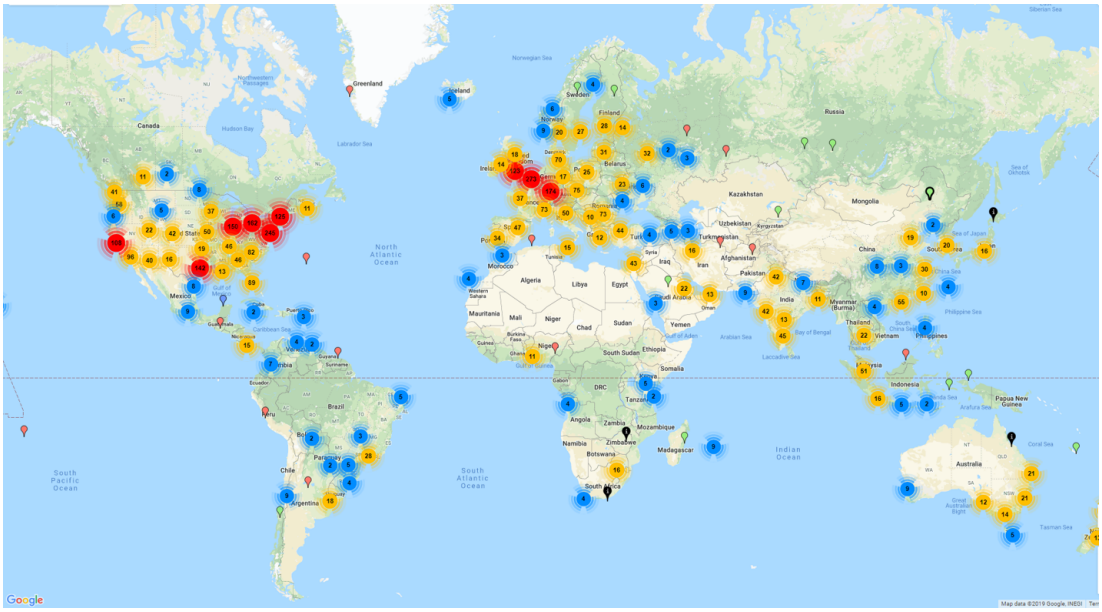


Figure 1·1: Global distribution of Data Centers (DataCenterResearch, 2019)

Center hardware has not been matched by a similar evolution of tools that target it. The traditional node hardware has changed from different types of resources being concentrated on different parts of the silicon, which resulted in high overhead for data movement, to now being implemented as multiple blocks distributed chip-wide; these blocks can be interleaved to achieve virtually infinite on-chip throughput as shown in Figure 1·3a. Similar to nodes, the high level structure of Data Centers has also changed as shown in Figure 1·3b. Instead of localizing compute to compute servers (CPU-centric), we now compute everywhere (Data-centric); in compute servers, in storage servers, and in the network. At all levels in the Data Center, however, tools have remained primitive i.e. they target traditional models rather than the ones implemented today. As a result, these tools are unable to leverage enhancements to the Data Center hardware, and a substantial fraction of available resources remains idle for a large number of cycles.

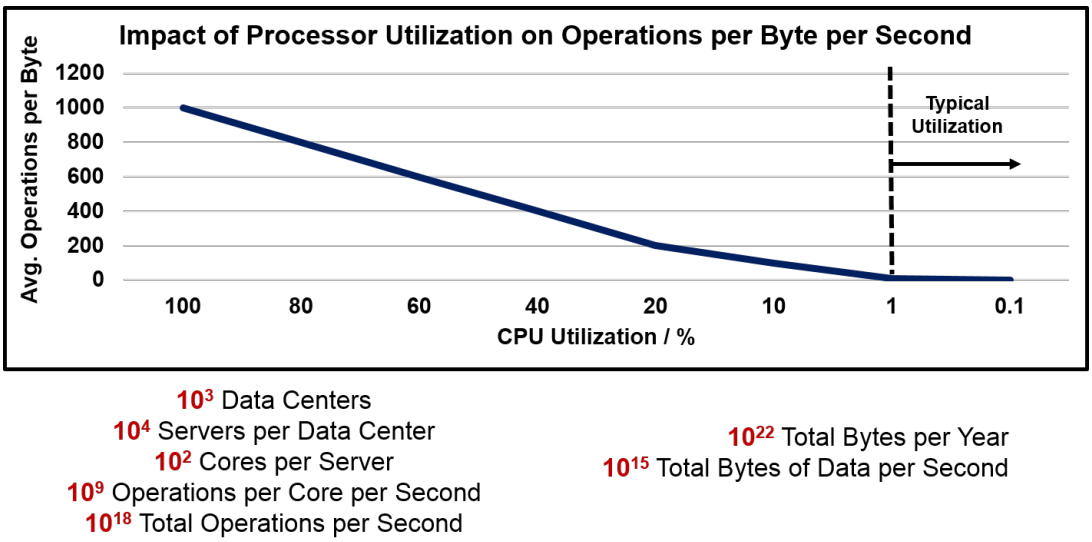


Figure 1·2: Low utilization of Data Centers - typically less than 1% (Cisco, 2017; Forbes, 2017)

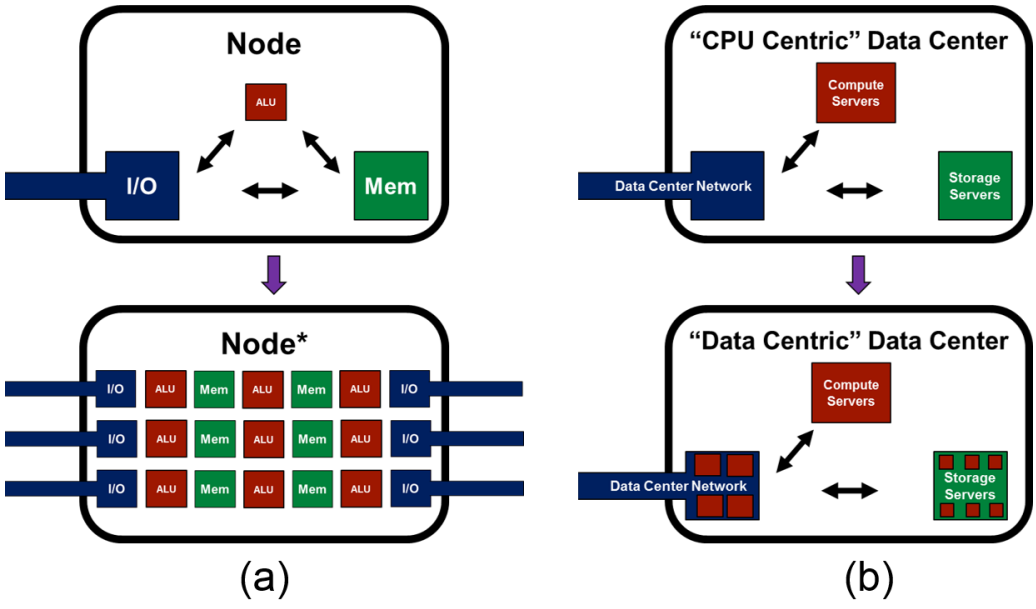


Figure 1·3: Hardware evolution of Data Centers and Data Center Nodes

1.2 Technology Domain of this Dissertation: FPGAs

One example of an FPGA enhanced Data Center is Microsoft's Azure cloud (Chung et al., 2018). Figure 1·4 shows the structure of a Catapult 2 node in the Azure cloud which leverages Bump-in-the-Wire (BitW) FPGAs. These BitW FPGAs are used by system administrators for a number of applications such as encryption, machine learning, database operations, SDN offload, Bing search, QoS, and web services. FPGAs were preferred over ASICs due to the diversity and dynamicity of these workloads; most offloads do not remain stable for long enough, or are only used by a small fraction of the servers.

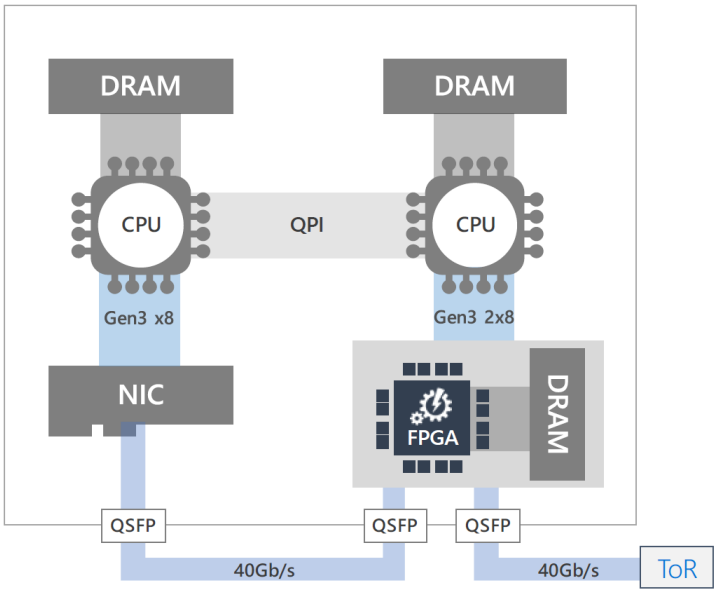


Figure 1·4: Microsoft Catapult 2 Node in the Azure Cloud

The reality is that FPGAs are rapidly becoming first class citizens in the Data Center. This because FPGAs: i) can achieve high throughput and low latency by implementing specialized architecture which eliminate a number of bottlenecks and overheads of general purpose computing, ii) consume low power and, by extension, have high power-performance, iii) have high speed interconnects and can tightly cou-

ple computation with communication to mask the latency of data movement, and iv) can be configured so that each design is tuned for individual use cases. As a result, our focus in this dissertation is the use of FPGAs in Data Centers.

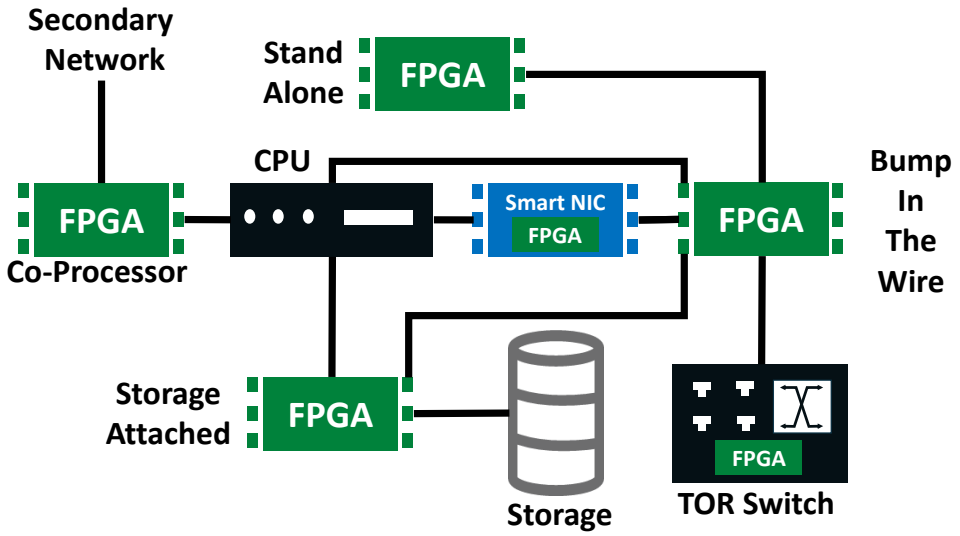


Figure 1·5: Potential configurations for deploying FPGAs in Data Center nodes

Figure 1·5 illustrates the different configurations in which FPGAs are being deployed in a Data Center. BitW FPGAs, such as Microsoft Catapult 2 in the Azure cloud, process all traffic between server and switch for performing application and system function acceleration. Co-Processor FPGAs, such as those available in (Alibaba, 2019; Amazon, 2019; Baidu, 2019; Chameleon, 2019; Huawei, 2019; Nimbix, 2019; OVH, 2019; OpenPOWER, 2019; Telekom, 2019; Tencent, 2019), provide a traditional accelerator configuration, similar to GPUs, with an optional back-end secondary network for direct accelerator-accelerator connectivity. Storage Attached FPGAs process data locally on storage servers to avoid memory copies to compute servers. Stand Alone FPGAs, such as the IBM cloudFPGA (Weerasinghe et al., 2015;

Weerasinghe et al., 2016a; Weerasinghe et al., 2016b), provide a pool of reconfigurable accelerators that can be programmed and interfaced directly over the network. Smart NICs, such as those by (Accolade, 2018; Broadcom, 2019; Endace, 2019; EthernityNetworks, 2019; MiTwell, 2019; Mellanox, 2018; Molex, 2018; Myricom, 2018; Napatech, 2018; Netcope, 2018; NewWaveDV, 2018; Portwell, 2018; Silicom, 2018; Solarflare, 2019; Vadatech, 2018), contain embedded FPGAs which perform custom packet processing alongside a NIC ASIC. These are typically deployed in high performance systems, such as those by (LANL, 2019). Finally, Network switches, such as those by (Arista, 2019), can also contain embedded FPGAs which operate on data as it moves through the Data Center network e.g. for compression, collectives.

1.3 Core Problem in This Dissertation

As shown in Figure 1·6, CPUs have traditionally had well defined programming models, e.g. abstractions and toolflows, and deployment support, e.g. Operating Systems and runtime. This enables CPUs to be easily targeted and high quality software to be developed without requiring significant prior expertise. On the other hand, FPGAs are predominantly spatial computers, which is where they derive high performance from. However, this is also the primary reason for FPGAs being difficult to program. They have traditionally lacked the clean, coherent, compatible, and consistent support found in CPUs. Herein lies the core problem that this dissertation is targeting; we cannot benefit from FPGAs if we cannot effectively code and deploy custom architectures.

In this dissertation, we "CPU-ize" FPGAs by addressing current limitations of code generation and deployment generation within the same framework. We use the phrase "CPU-ize" to denote an advancement to the status of CPU systems. This does not imply that this is a limit or complete (that everything needs to get done)

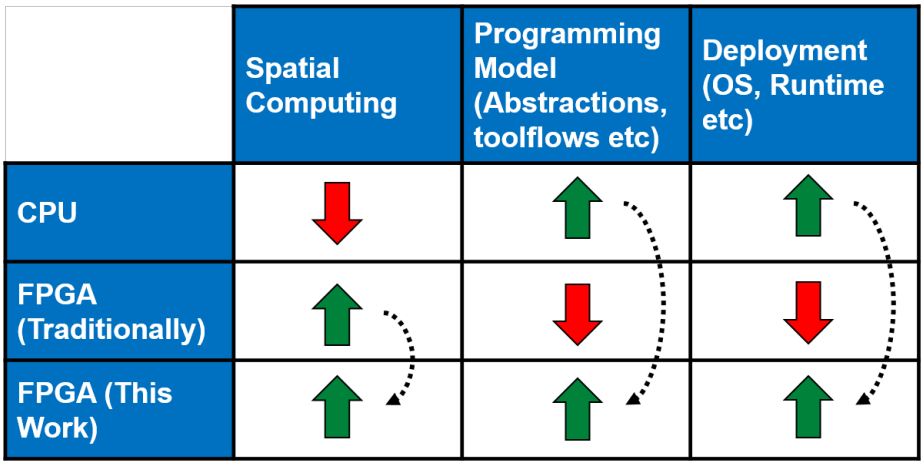


Figure 1·6: "CPU-ization" of FPGAs through effective and well defined programming models and deployment support

or completely the same. As shown in Figure 1·6, this work is aimed at building programming models and deployment support similar to what is available for CPUs. This is a good target because CPU support for code generation and deployment generation is both extremely well-understood and much more advanced than the state of the art for FPGA systems.

1.4 Major Parts of the Dissertation

As mentioned above, there are three major parts of this dissertation i.e. code generation, deployment generation, and the framework that links them together.

1.4.1 Framework: High Level Synthesis (HLS)

HLS refers to generation of hardware using high level languages (Intel, 2019; Xilinx, 2018; Nikhil, 2004; Microsemi, 2019; MathWorks, 2018; Maxeler, 2019; Mentor, 2019). The aggregate design space for these HLS tools has evolved in a predominantly rectilinear trajectory. They are relatively simple to construct since their focus is primarily on stringing together pre-implemented modules of a limited set of key design patterns (e.g. systolic arrays, SIMD, pipelining) and core low level hardware

features (e.g. registers, BRAMs streams). On the front end, these tools reduce the complexity of specifying hardware constructs in order to minimize the effort and lines-of-code (LOC) needed to express high performance target architecture. This frees up developers to focus only on specifying core components of the algorithm, instead of spending substantial overhead on engineering implementations of trivial design aspects e.g. interconnect fabric, floating point arithmetic. On the back end, these tools replace code-sequences/constructs/semantics in the abstraction with one or more functionally equivalent IP blocks. With the prevalent assumption being that IP blocks (vendor ones in particular) are the most optimal approach to implementing hardware, this method is expected to give the best possible performance.

Figure 1·7 illustrates a typical development process using HLS tools. The top half corresponds to code generation while the bottom half is deployment generation. We discuss both parts in detail below.

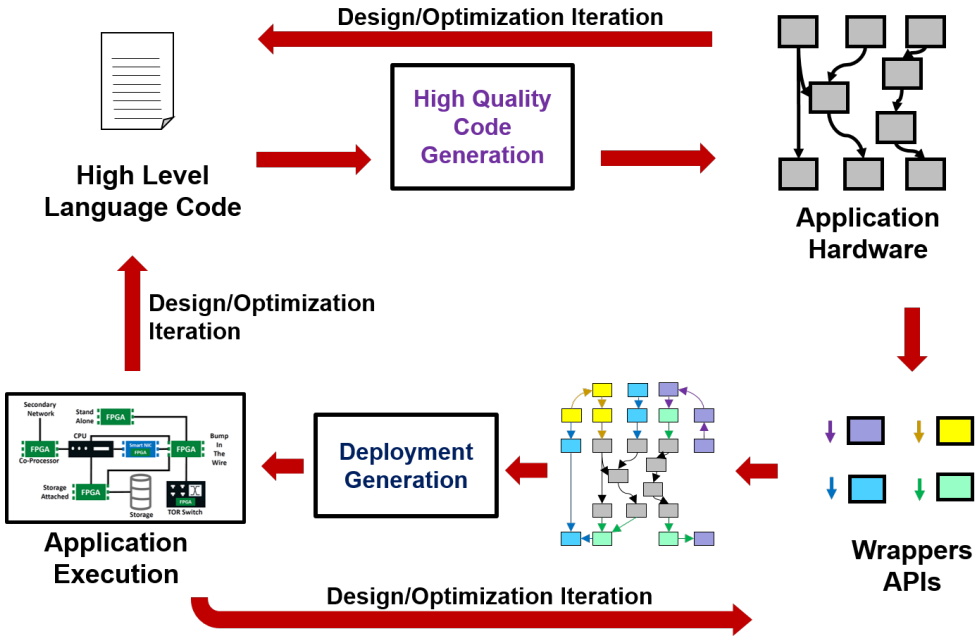


Figure 1·7: Typical HLS Flow showing the challenges of code generation and deployment generation

1.4.2 Code Generation

Code generation targets creating high quality application hardware. The process is similar to that of software development, as shown in Figure 1·7. Developers begin with an initial representation of their application. Then they perform a number of code generation iterations, first to develop a functionally correct design, and then to converge to an optimal implementation of this design. Unlike software, however, they are two complexities associated with hardware code generation which makes the process both difficult and time consuming.

First, hardware generation involves developing both an appropriate algorithm for an application and the architecture on which this algorithm will execute. As a result, code generation relies heavily on developer expertise in providing sufficient details regarding the algorithm and its execution. A vast design space may need to be explored before high quality hardware is generated. It is even possible for hardware developers to not get good performance at all due to abstractions of HLS which limit control over low level hardware.

Second, hardware generation can take hours/days per design iteration. Therefore, making FPGAs easier to program does not mean only decreasing initial development overhead. Code generation should also be fast i.e. it should enable a larger number of optimization iterations to be done within a given timeframe. This enables a faster convergence to a high performance solution.

1.4.3 Deployment Generation

Conversely, deployment generation builds hardware and software components needed to enable execution of workloads on FPGA systems. Provisioning of these components by default removes the burden on developers for building them, allowing developers to focus on implementing the core computations in their applications. Examples of

deployment generation include: controllers for external interfaces (e.g. host, memory network), APIs to leverage these interfaces from high level code, wrappers to ensure compatibility between arbitrary pipeline structures and standard system interfaces, drivers to reconfigure the FPGA from a host machine, and use of these drivers to provide complex high speed I/O between host and device.

While deployment generation covers a vast set of statically-provided default functionality, the drawback is that there is virtually no compatibility across these functions if they are a result of independent efforts. While developers may save time by not having to build these functions, they still need to build wrappers to address differences in APIs and functionality. As shown in Figure 1·7, this results in an additional stage prior to deployment generation where the developer needs to build wrappers and APIs to ensure compatibility between application hardware and the hardware operating system. Moreover, similar to code generation, this process is not one off. Multiple iterations may be needed to converge to an acceptable solution. In the worst case, inherent problems can result in the application hardware being completely scrapped, and the entire process restarting from the beginning.

1.5 HaaRNESS

To truly unlock the potential of FPGAs, not only should there be further exploration aimed at improving current practices of how hardware is generated and deployed, but these two branches must also begin to intersect. The traditional HLS framework must be enhanced so that it simultaneously addresses the challenges of both code generation and deployment generation in order to develop and deploy FPGA workloads with minimum effort and maximum efficiency. Perhaps the most critical impact is that this enhanced framework removes the need for developer interaction and, by extension, requirements for expertise in FPGA programming. The entire process can

be completely transparent, from high level representation to high quality hardware to the subsequent deployment on the FPGA; the same level of effort as for a single CPU core. We refer to this enhanced HLS framework as Hardware as a RecoNfigurable, Elastic and Specialized Service (HaaRNESS).

The motivation behind HaaRNESS is for Data Centers to effectively support:

- **Reconfigurablity:** In order to maximize the efficiency of compute resources, hardware must be as malleable as software. The ability to reconfigure hardware allows us to tap into the best of both CPU and ASIC worlds. Similar to ASICs, creating custom pipelines enables us to eliminate inherent inefficiencies, constraints and bottlenecks in hardware that would otherwise reduce the number of cycles spent doing useful work. Moreover, similar to CPUs, this custom hardware can be easily changed to cover a wide range of frequently changing Data Center workloads.

- **Elasticity:** Assigning fixed quantum of reconfigurable resource (such as a full FPGA) results in under utilization of resources, especially in workloads where demand is a function of physical variables e.g. time of day. As a result, the utility of unused fabric is wasted. Our goal is therefore to support elasticity i.e. partition the reconfigurable fabric and allocate variable quanta across one or more devices based on runtime demand.

- **Specialization:** Being reconfigurable does not guarantee that a design is specialized. If designs are selected from a library of pre-built architectures, developers can only choose the best case option, and not necessarily the best possible one. Our goal is to decouple a developer's ability to leveraging reconfigurable hardware effectively from the developer's expertise in hardware programming; this will allow them to build hardware that is tailor made for their use cases.

Figure 1·8 illustrates the target HaaRNESS flow. The input is a HLL code with minimum use of pragmas and low level constructs. The primary goal of a developer is to only specify the algorithm, and hence virtually no prior expertise in hardware development is required. Then, the code generation stage automatically infers high quality hardware from this high level representation, with negligible developer interaction and rapid turnaround if design iterations are needed. Once application hardware is generated, it can be directly passed to the deployment generation stage due to the compatibility between code and deployment generation (which eliminates the need for wrappers). Finally, this deployment generation wraps the application hardware with a hardware operating system and provides the software runtime needed to program and interface it.

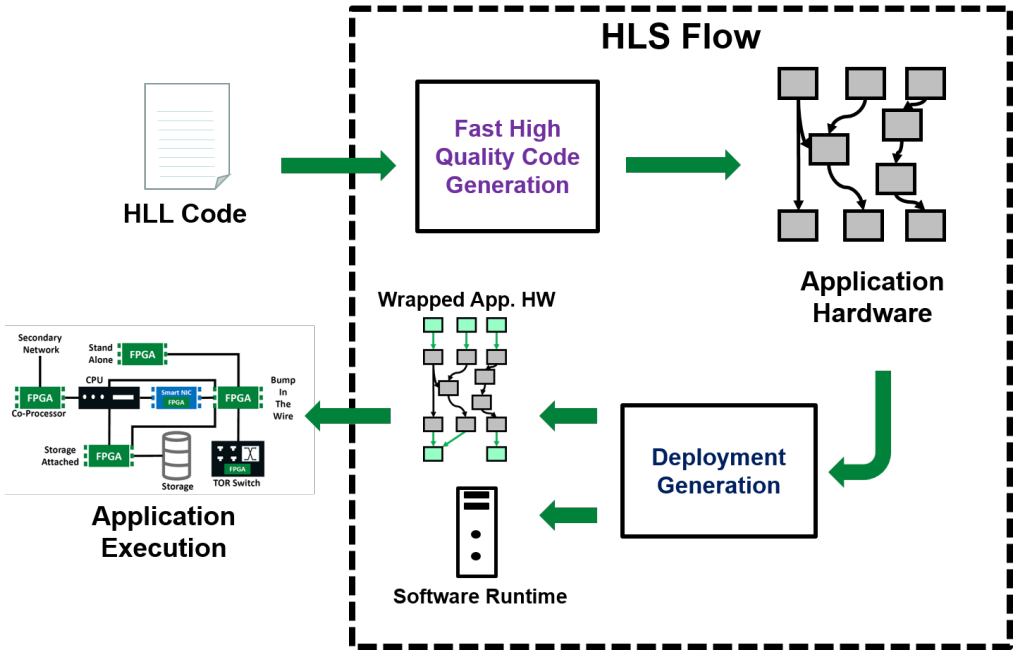


Figure 1·8: The HaaRNESS flow: fast high quality code generation and uniform deployment generation

1.6 Contributions

While building the entire HaaRNESS framework is beyond the scope of this dissertation, we target three of the core challenges here: i) reducing the turnaround time for design space exploration, ii) addressing optimization blockers to bridge the HDL/HLL gap, and iii) formalizing and generalizing hardware operating systems to address the*ad hoc*'ess in deployment generation. This dissertation makes the following specific contributions:

1. An empirically guided set of systematic transformations for generating high quality HLS code.

2. A framework for instrumenting HLS compiler to identify and remove optimization blockers.

3. A framework for RTL simulation and IP generation of HLS kernels for rapid turnaround.

4. A framework for generalization and formalization of hardware operating systems to address the *ad hoc*'ness of existing deployment generation, and ensure uniform structure and APIs.

As shown in Figure 1·9, contributions 1 and 2 add a new Pre-processor stage to the HaaRNESS toolflow that is not available in typical HLS tools, while contributions 3 and 4 augment what is currently available in the System Generator and Hardware Operating System stages of these HLS tools. The remaining stages are used as is.

Figure 1·10 illustrates the technical organization of this project. Here, numbered circles represent mapping of this dissertation's contributions to each stage. Starting with a simple HLL code, we first pre-processes it to help the HLS compiler infer opportunities for high quality hardware generation. The resulting transformed HLL

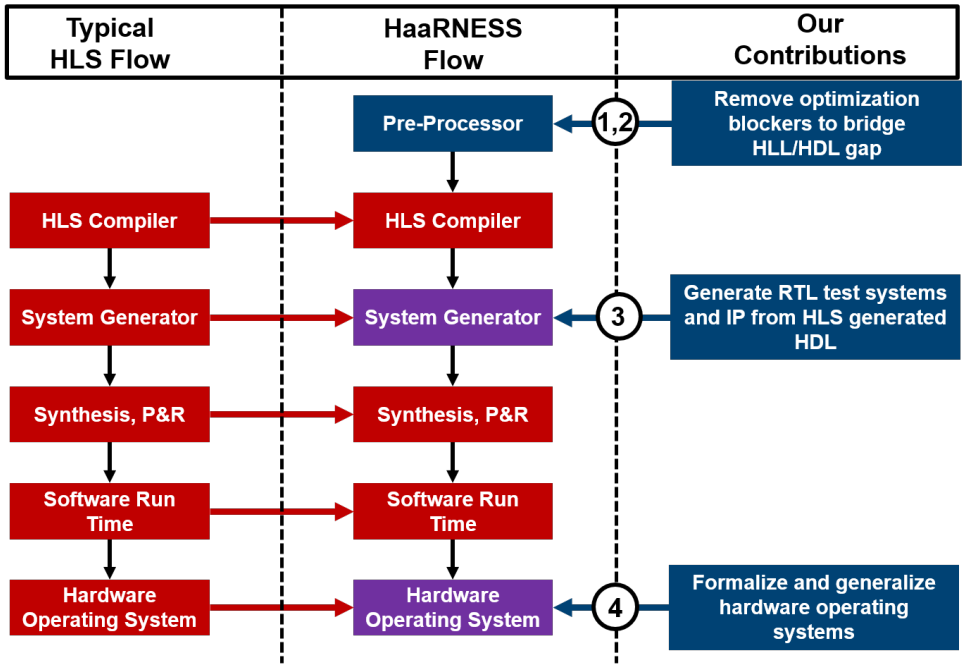


Figure 1·9: Specific contributions of this dissertation

code is referred to as HLL*. This is then converted into RTL using a HLS compiler; the compiler generates both application logic and wrappers, with the latter responsible for implementing interface mappings, schedulers, memory controllers (Load Store Units), resource multiplexing etc. Then, application logic is isolated from the overall RTL and interfaced with a testbench. The resulting RTL simulation provides a cycle accurate estimate of application logic behavior and expected performance. If this performance is not sufficient, feedback is provided to the developer to make modifications to the starting HLL code. On the other hand, if the performance is good, the developer has two options. They can either generate an IP for this application logic, or deploy the application on an FPGA system. In case of the latter, the complete RTL system generated by the HLS compiler is tested using RTL simulation. Once performance is deemed satisfactory, hardware and software runtime support is provided to enable execution of the application.

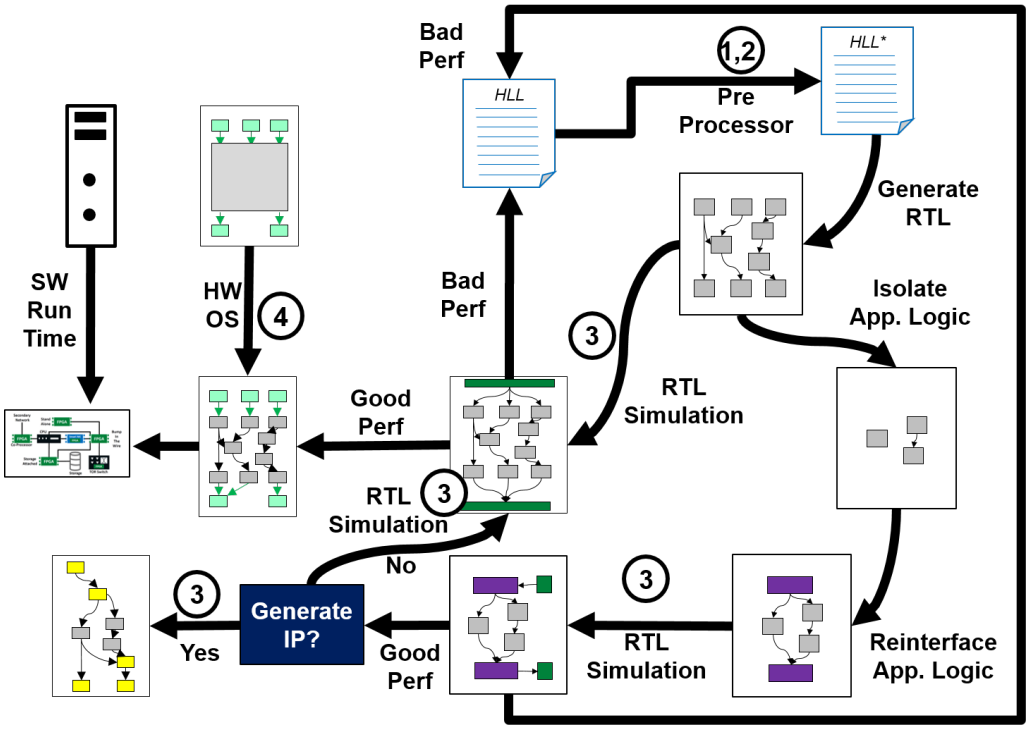


Figure 1·10: Teachnical/System organization

1.7 Underlying Systems and Tools

Here we provided details for different systems and tools used in this dissertation.

1.7.1 FPGA

Our implementations are primarily evaluated using the Gidel Proc10A board (Gidel, 2019), which contains a medium-end Intel Arria10X115 FPGA. It has 427,200 logic and 2713 RAM blocks, with a total on-chip memory of 53Mb. We use the Intel OpenCL SDK for FPGA v16.0.2 for compiling our codes. For a limited set of applications, we implement designs using a Stratix-VD8 FPGA. This has 262,400 ALMs and 2567 RAM blocks, with a total of 50Mb on-chip memory. OpenCL SDK v16.0.2 is used for Stratix-V compilations as well.

1.7.2 GPU

For GPU implementations, we use the high-end Tesla P100 PCIe 12GB GPU with CUDA 8.0. It has 3584 CUDA cores and peak bandwidth of 549 GB/s.

1.7.3 CPU

CPU codes are compiled with the Intel C++ Compiler v16.0.1. We use MKL (Intel, 2018a) libraries for CPU codes where-ever possible to ensure high performance for corresponding applications.

1.8 Overview

The rest of this dissertation is organized as follows.

Chapter 2 discusses the previous efforts with regards to the contributions of this dissertation. We first highlight the role FPGAs are playing in modern Data Centers and High Performance Computing. Then we discuss related work in pre-processing HLS kernels, modifications to the HLS compiler (primarily as an alternative to pre-processing), system generation for application simulation and implementation, and development of hardware operating systems.

Chapter 3 presents our approach for achieving fast turnaround for design space exploration using RTL simulations. These simulations target both application logic directly, as well as the overall OpenCL system. Moreover, they allow testbenches to be modified to effectively emulate devices external to the FPGAs. By moving hardware generation outside optimization iterations, we are able to substantially reduce the time taken to obtain data points.

Chapter 4 extends the approach from Chapter 3 and uses HLS generated application logic to build IP blocks. Not only can these blocks directly replace components in HDL systems, but the rapid turnaround and application specific design means they

can outperform both vendor IP and ASICs.

Chapter 5 presents our empirically guided set of systematic transformations for OpenCL kernels. These pre-processing transforms help the HLS compiler infer opportunities for parallelism and significantly improve the quality of generated hardware. We evaluate the effectiveness of our approach using parallel computing dwarfs and network packet processing applications.

Chapter 6 presents our first principles approach for identifying and removing optimization blockers. By instrumenting HLS compilers, we can identify what, where and how optimizations are blocked. As a result, we can then modify the compiler, or propose new pragmas for address the compiler limitations.

Chapter 7 presents our framework for formalizing and generalizing hardware operating systems, which we refer to as Reconfigurable Hardware Operating Systems (RHOS). We demonstrate that our proposed RHOS generator can be integrated into HLS tools, and can effectively target a number of different configurations in which FPGAs are deployed in Data Centers.

Finally, Chapter 8 gives our conclusion.

Chapter 2

Background

2.1 FPGA Enhanced Computing

FPGAs enable developers to design custom systems that leverage application-specific optimizations. Unlike GPUs, FPGAs are not constrained to data parallelism, a coprocessor configuration, standard data types, or other limitations of fixed architectures. They can support virtually any computation, can be tailored based on its nature and context, have very high resource utilization, and consume much less energy. FPGAs co-locate communication and computation on the same device, eliminating most communication overhead. Additionally, device-to-device bandwidth is high while latency is low and computation can be embedded within communication to drastically improve performance.

Here we give a sample of prior work in using FPGAs for high end computing. This includes general overviews (Herbordt et al., 2007b; Herbordt et al., 2008a; VanCourt and Herbordt, 2009), basic work in programmability and performance (Herbordt and VanCourt, 2005; VanCourt and Herbordt, 2005a; VanCourt and Herbordt, 2006a; VanCourt and Herbordt, 2006c), programmability and performance using a commercial tool chain (Yang et al., 2017a; Sanaullah and Herbordt, 2017; Sanaullah and Herbordt, 2018a; Sanaullah and Herbordt, 2018b; Sanaullah et al., 2018a; Sanaullah et al., 2018b), FPGA system design and architecture (Pascoe et al., 2010; Khan and Herbordt, 2012; Sheng et al., 2015; George et al., 2016; Sheng et al., 2016b; Sheng et al., 2016a; Sheng et al., 2017; Sheng et al., 2018b; Sheng et al., 2018a), FPGAs

used with middleware such a MPI (Xiong et al., 2018b; Xiong et al., 2018a; Xiong et al., 2019; Stern et al., 2017; Stern et al., 2018), and many case studies involving applications. Bioinformatics work includes studies of dynamic programming based algorithms (VanCourt and Herbordt, 2004; VanCourt and Herbordt, 2007), heuristic sequence alignment such as BLAST (Herbordt et al., 2006; Herbordt et al., 2007a; Park et al., 2009; Park et al., 2010; Mahram and Herbordt, 2010; Mahram and Herbordt, 2012a; Mahram and Herbordt, 2015), multiple sequence alignment (Mahram and Herbordt, 2012b), and other string matching applications (Conti et al., 2004). Molecular Dynamics studies include surveys (Chiu et al., 2008; Chiu and Herbordt, 2010a; Herbordt, 2013; Khan et al., 2013) integration (Gu et al., 2006c), datapath optimization (Gu et al., 2006a; Gu et al., 2006b; Gu et al., 2008), handling neighbor lists (Chiu and Herbordt, 2009; Chiu and Herbordt, 2010b; Chiu et al., 2011), particle mapping (Sanaullah et al., 2016a; Sanaullah et al., 2016b), the long range force using multigrid (Gu and Herbordt, 2007), the 3D FFT (Humphries et al., 2014; Sheng et al., 2014), the bonded force (Xiong and Herbordt, 2017) and complete FPGA integration (Yang et al., 2019b; Yang et al., 2019a). Other HPC applications include Discrete Molecular Dynamics (Model and Herbordt, 2007; Herbordt et al., 2008b), Molecular Docking (VanCourt et al., 2004; VanCourt and Herbordt, 2005b; VanCourt and Herbordt, 2006b; Sukhwani and Herbordt, 2008; Sukhwani and Herbordt, 2010), Microarray Analysis (VanCourt et al., 2003; VanCourt et al., 2004), Adaptive Mesh Refinement, (Wang et al., 2019b; Wang et al., 2019a), and Machine Learning (Sanaullah et al., 2018c; Geng et al., 2018b; Geng et al., 2018a; Geng et al., 2019b; Geng et al., 2019a).

2.2 Pre-Processor

In this section, we discuss the two broad categories of existing work in pre-processing HLL code for HLS compilers: i) General Pre-processing i.e. transforms generally applicable to nearly all application domains, and ii) Domain Specific Languages (DSLs)

2.2.1 General Pre-processing

Previous work here has primarily focused on either optimizing a given application, or characterizing the impact of individual optimizations. In case of the former, speedups are measured with respect to either a CPU baseline code, or an OpenCL baseline code with no optimizations (e.g., (Abedalmuhdi et al., 2017; Rodriguez-Donate et al., 2015; Weller et al., 2017)); in both cases, the utility of the result is low since performance improvements over these forms of baselines are expected even with trivial optimizations. In case of the latter, the space covered in previous studies for characterizing HLS optimizations (Rodriguez-Donate et al., 2015; Jin et al., 2017b; Krommydas et al., 2016; Jin et al., 2017a) has mostly been limited as well. Often only a common set of simple optimizations is applied, with the initial code being written for a GPU with Multiple Work Item Kernels. Characterizations may be limited to varying the number of SIMD lanes or compute units.

Overall, use of HLS has generally resulted in performance being sacrificed for programmability. In (Yang et al., 2017a), the authors have developed multi-producer single-consumer architectures using OpenCL for processing particle interactions. The design suffers from a significant reduction in performance as compared to Verilog designs. Authors in (Abedalmuhdi et al., 2017) have implemented a particle in cell simulation on an Arria 10 board using OpenCL, but only managed a 2.5x speedup after optimizations over a single core CPU. These algorithms have traditionally shown orders of magnitude better performance for FPGAs over CPUs due to significant

opportunities for parallelism. Similarly, for Smith-Waterman which has typically achieved speedups over GPUs using HDL implementations, (Rucci et al., 2017) has shown that GPUs outperform FPGAs when OpenCL is used. In (Ulutas et al., 2017), the authors have implemented an anisotropic Huber-L optical flow estimation algorithm on high-end FPGA and GPU boards. Their results show that the GPU implementation has an average speedup of 20x over FPGAs.

2.2.2 Domain Specific Languages (DSLs)

DSLs are typically built on top of existing HLS tools/frameworks and make it easier to express efficient algorithms for the target domain. This can be achieved through custom data structures, semantics, compiler directives etc. A popular area of research for DSLs in previous work has been network packet processing. Microsoft's ClickNP (Li et al., 2016b) uses an OpenCL front end to help developers specify target workloads using provided semantics and library components. The resulting design is translated to intermediate C code, and then compiled using the standard vendor High Level Synthesis (HLS) toolflow. As shown in (Li et al., 2016a), ClickNP uses both OpenCL code and custom RTL for building designs. The Xilinx SDNet PX programming language (Xilinx, 2019) uses object-oriented programming to specify required packet processing components and their connectivity. The actual implementation of rules applied to packets is done automatically. P4FPGA (Wang et al., 2017) uses Bluespec System Verilog to implement a match-action table model. The Flowblaze abstraction (Pontarelli et al., 2019) extends match-action languages by enabling them to store per-flow state. Emu (Sultana et al., 2017) is a framework which provides a library for building network functions, as well as run time and debugging support. The drawback of DSLs is that while they successfully address the programmability problem for a given domain, they cannot be used for other application domains. Moreover, even when using DSLs, the developer is still typically required to have existing knowledge

of how algorithms map to hardware.

2.3 HLS Compilers

While modifying the HLS compiler is not one of our contributions, we still discuss previous work here since these modifications can serve as an alternative to pre-processing.

2.3.1 Profiler Driven HLS

Profiler driven HLS uses analysis of the Intermediate Representation (IR) to guide the transformation process. Authors in (Huang et al., 2013) use profiling to accept or reject compiler passes based on predicted impact on the number of hardware execution cycles. The drawbacks of their approach are that i) all profiling is done based on a single estimated metric and ii) it does not provide feedback regarding inherent problems in the application algorithm. Work in (Wang et al., 2016) performs a more indepth analysis of the IR. Authors first provide a GPU-like execution model for FPGAs based on OpenCL. Through custom built-in LLVM passes, they capture a number of parameters in the LLVM IR, which are then used in combination with user inputs, compilation reports etc to identify performance bottlenecks. The drawbacks here are that i) a GPU model for FPGAs is inefficient (discussed in detail later) and ii) there is a reliance on data apart from static profiling e.g. kernel frequency.

2.3.2 Compiler Enhancements

Compiler enhancements typically improve the manner in which existing compilers perform certain operations, by either modifying how a particular pass is done or adding newer ones to augment it. Authors in (Ben-Nun et al., 2019) present an improved IR for targeting scientific data center workloads. Their Stateful DataFlow multiGraph (SDFG) separates code definition from its optimization, and allows high performance hardware to be generated without modifications to the original code.

The drawback to their approach, however, is that it requires programmer interaction (and by extension, expertise) to perform transformations on a SDFG. Another modification to the IR is provided in (Zhang et al., 2010), where authors present a Bitwise-Only Data Flow Graph (BO-DFG). This is aimed at providing expressibility for bitwise operations only, so that they can be analyzed and optimized. Results of this process can then be converted into instructions in the standard DFG. Authors in (Richter-Gottfried et al., 2016) provide limited enhancements to the OCLAcc compiler by only adding support for implementing conditional branches and custom data widths.

2.3.3 Custom Compilers

Custom compilers are perhaps the most complex solutions since they involve designing compilers that do a majority, if not all, of the code transformations needed to generate hardware. LegUP (Canis et al., 2011) is one of the most popular compilers in this regards, since it is open source and allows new HLS algorithms to be easily explored. Unlike traditional compilers, LegUP targets hybrid system generation, mapping code to a combination of specialized pipelines and softcores. Similar to LegUP, the Trident compiler (Tripp et al., 2005) also provides an open framework, but is specifically aimed at exploring floating point algorithms. Authors in (Koeplinger et al., 2018) present the Spatial compiler which generates Chisel RTL code. Unlike typical HLS compilers, Spatial automatically performs a number of code transformations that otherwise require manual tuning in traditional HLS tools.

2.4 System Generator

Here we discuss the existing state of the system generator stage. Our focus is primarily on: i) analyzing the output of HLS compilers for functional verification and performance estimation i.e. Test System Generation, ii) packaging of application logic

into IP blocks, and iii) alternative to (ii), wrapping application logic and providing appropriate runtime support to deploy and interface application logic on the FPGA system i.e. Board Support Packages.

2.4.1 Test System Generation

Emulation

Emulation is used to simulate kernel code for functional verification. Compiling for emulation allows the compiler to generate CPU equivalent code for FPGA-specific constructs, such as channels, and then execute the entire computation in software. This is not only useful for ensuring that computation and memory accesses have been correctly defined, but can also identify run-time faults, such as occurrences of deadlocks. It does not, however, provide any information regarding kernel code mapping to hardware or estimated performance.

Reports

We treat reports output by the HLS compiler as static systems i.e. a non compile-able representation of the OpenCL system. Reports are generated by the compiler to provide an overview of kernel translation to hardware. Here, we briefly list the main categories of these reports and their contribution towards code optimization. A comprehensive list of reports and their detailed description are provided in the Intel OpenCL best practices guide (Intel, 2018c).

- **Loop analysis** is used to determine initiation intervals (II) for loops in the kernel and the dependencies causing high IIs. Resolving these dependencies allows loops to operate stall free.

- **Area analysis** provides estimates of resource usage and implementation details for data structures. This is particularly useful for determining whether the

compiler has correctly inferred the optimal hardware based on access patterns, or is resorting to sub-optimal, high-resource "safe" options such as memory replication and barrel shifters.

- **System viewer** gives a graphical overview of the kernel computation and memory accesses. Kernel execution is represented as sequential blocks, with each block carrying out a varying number of operations such as memory transactions, channel calls and loop iterations. Details provided include latencies, stalls, types and sizes of Load-Store units created for each memory transaction, and the dependencies between blocks.

- **Kernel memory viewer** gives a graphical overview of the connectivity of Load-Store units with external memory banks. This can be used to verify that the compiler has correctly inferred off-chip access patterns.

Similar to Emulation, reports do not guarantee good performance. Kernel codes with no loop dependencies, initialization intervals equal to 1, efficient memories and low latencies can still be sub-optimal. This is because little information is provided regarding the composition, organization, and connectivity of compute pipelines. To truly identify bottlenecks in the design and optimize them, low-level details are required regarding implementation and behavior of the entire system.

Co-Simulation

Co-simulation refers to the automated generation of RTL test systems from HLL code. Developers provide both the HLL application function code and a set of HLL test inputs. The former is converted into RTL while the latter is used to generate test vectors that be applied to the inputs of this RTL. There are several drawbacks of co-simulation. First, it is not always supported in a toolflow e.g. Intel OpenCL. Second, it only tests the application logic and cannot simulate the execution of the

full HLS generated system (including wrappers). Finally, the testbench is constrained since it is built automatically from HLL test inputs. As a result, the testbench cannot effectively model the latencies and behavior of devices external to the FPGA. In the worst case, inaccurate feedback resulting from improperly modelled test environments can result in developers converging to a design that, when actually implemented on an FPGA, gives worse performance than naive code; it is likely to fail to execute altogether if deadlocks were not properly identified.

2.4.2 IP Generation

Tools such as Intel HLS (Intel, 2019) and Vivado HLS (Xilinx, 2018) provide the option of generating IP directly from application HLL code, instead of creating a full system as is in the case of Intel OpenCL. This enables the creation of a library of building blocks that can be joined together to form complex systems. Since each block is built separately, both the hardware generation time and complexity are reduced. The drawback of these IP generation tools, however, is that they can only generate IP for simple functions; designs containing structures such as multi-function code, global arrays or channels are not supported.

2.4.3 Board Support Packages

A Board Support Package (BSP) is a set of components needed to target a particular FPGA board using Intel OpenCL SDK for FPGAs. Every unique FPGA board will have its own BSP even if it uses the same FPGA as other boards. This is similar to the BSP in software, which contains code specific to the hardware on which the OS executes. The low level components in an FPGA BSP include drivers, APIs, compilation scripts, pin-out mappings, fabric partitions, freeze wrappers, and a hardware operating system. A number of such BSPs are available from different vendors (Nallatech, 2018; BittWare, 2018; Gidel, 2018). In this dissertation, we focus

on two primary challenges associated with the hardware operating system of the BSP. First, it is on the critical path of data movements between the kernel and host/off-chip-memory interfaces; if the hardware operating system is not well designed, it can limit the maximum achievable performance. Second, hardware operating systems consume a significant portion of the FPGA fabric which reduces the amount of logic available to applications; an inefficient hardware operating system design can cause compilations to fail due to insufficient fabric availability.

2.5 Hardware Operating Systems

Here we present previous efforts in hardware operating systems for FPGAs (not exclusive to those found in BSPs). These are commonly referred to as "Shells". A Shell is an arbitrary set of static features that encompass one or more custom logic regions. We refer to static logic as a part of the FPGA fabric that can only be modified through full reconfiguration, while custom logic regions are dynamically reconfigured through (significantly faster) Partial Reconfiguration (PR). The purpose of shell hardware is to reduce developer effort by providing key functionality, such as memory/network/host controllers and PR, as well as managing interconnects between different static and reconfigurable components. As shown by the literature survey below, the design of a Shell is ad hoc i.e. it is based on the requirements of individual contexts, instead of a formal structure.

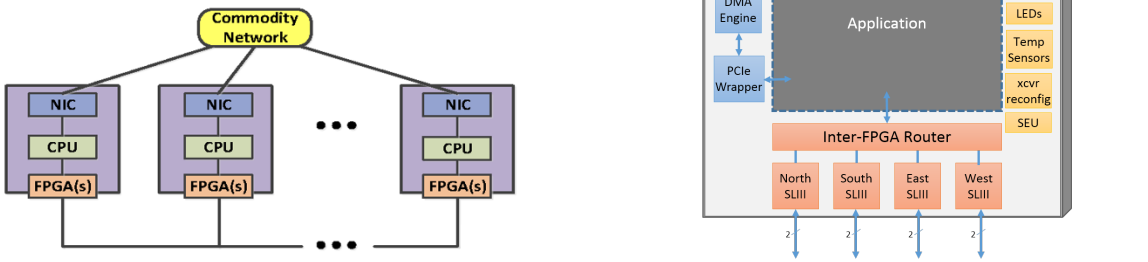
2.5.1 Microsoft

Microsoft has developed two Shells for deploying large scale multi-FPGA systems as part of the Catapult project: Catapult 1, which is a rack-scale infrastructure, and Catapult 2, which can scale to millions of nodes.

Catapult 1

Deployment of Catapult 1 (Putnam et al., 2014; Microsoft, 2018a) was done using a 1632 server testbed and initial evaluation showed a 95% per server throughput input. Figure 2·1a shows an overview of Catapult 1 nodes (purple). The CPU and FPGA in each node communicate via PCIe. For external connectivity, dual network interfaces are used. The first is Ethernet based, which enables CPUs to communicate with each other over the commodity data center network. The second is a specialized, low latency (order of 100ns) network implemented using 10Gb SAS cables which connects FPGAs to each other. The topology for the second network is a 2D torus. Figure 2·1b provides details of components within the FPGA. Here, custom offload logic is implemented within an FPGA fabric allocation called "Role", while the Shell logic is composed of:

- PCIe Core: This is used to implement an interface between the host CPU and the corresponding FPGA.
- SerialLite III (SL3): This is a lightweight protocol used for inter FPGA communication over the SAS links. Since each FPGA connects to four other FPGAs (2D Torus), there are four SL3 blocks in the Shell.
- DRAM Controllers: These are used to provide off-chip memory access to the Role or host (for DMA over PCIe).
- Inter-FPGA Router: A crossbar that connects the four SL3 blocks, PCIe Core and Role. A software configured routing table makes routing decisions, while a virtual cut-through transport protocol is used.



(a) Dual network deployment with FPGAs being connected through a secondary dedicated network

(b) Shell architecture (Microsoft, 2018b)

Figure 2·1: Microsoft Catapult 1

Catapult 2

To the best of our knowledge, Microsoft Catapult 2 (Caulfield et al., 2016) is the only production system (deployed on all new Microsoft servers) that offers direct connectivity between FPGAs and the data centre network in a Bump-in-the-Wire (BitW) configuration. Figure 1·4 shows an overview of the architecture of a Catapult 2 node. Each node contains two CPUs with Quick Path Interconnect (QPI) to enable fast inter-CPU communication, a discrete NIC and a discrete FPGA. CPUs do not have a direct network connection. Rather, all CPU based network traffic first passes through a discrete NIC, which in turn connects to the FPGA. Logic within the FPGA is then responsible for managing contention for Top of Rack (ToR) connectivity between CPU and FPGA based traffic. Use of a discrete NIC enables network offload and packet transport functionalities, without the resource, performance and complexity overhead of implementing them in the FPGA. On the other hand, use of a discrete FPGA (as opposed to an FPGA enhanced NIC), enables an independent FPGA-CPU

PCIe connection for interfacing offloaded logic.

Figure 2·2 shows the intra-FPGA connectivity of each FPGA. Similar to Catapult 1, regions for implementing custom offloads are referred to as "Roles". Unlike Catapult 1 however, the FPGA can have multiple Roles. The Catapult 2 Shell includes:

- Network Bridge and Bypass: This enables the functionality of passing packets between the NIC and TOR interfaces. A tap is also implemented which allows Roles to "inject, inspect and alter the network traffic as needed".
- Lightweight Transport Layer (LTL) Protocol Engine: This enables communication between the local FPGA and other local and remote devices in the data center.
- Elastic Router (ER): This is a credit based, input buffered crossbar switch that is used to support communication between different endpoints on the FPGA, such as individual Roles and Shell functions. The ER can be customized based on requirements for "number of ports, virtual channels, flit and phit sizes, and buffer capacities". Credit based flow control is used, with credits shared between multiple virtual channels.
- PCIe Gen 3: This is used to interface offloaded functions with the CPU, as well as perform DMA operations.
- DDR3 Controller: This is used to provide off-chip memory access to FPGA logic.

2.5.2 SAVI

The Smart Applications on Virtual Infrastructure (SAVI) testbed is a nation-wide deployment of heterogeneous infrastructure in Canada (SAVI, 2019). There are two

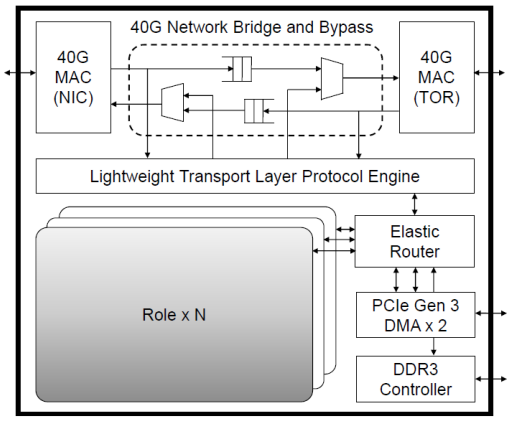


Figure 2·2: Microsoft Catapult 2 Shell architecture (Caulfield et al., 2016)

notable efforts for providing system support for FPGAs within this testbed, which we refer to as Byma et al (Byma et al., 2014; Byma, 2014) and Tarafdar et al (Tarafdar et al., 2017).

Byma et al

Figure 2·3 illustrates FPGA connectivity in the SAVI testbed. Each server node is composed of a host CPU and FPGA. CPU-FPGA connectivity is over JTAG (possibly USB based), which is used for providing i) UART interfaces to provider logic, and ii) Partial Reconfiguration (PR) functionality. The CPU runs an agent, which is responsible for provisioning custom logic by interfacing a remote OpenStack controller and implementing functions such as getting available resource on the FPGA, programming with a PR image, deleting PR image, and set/reset of MAC addresses. Both FPGA and CPU have Ethernet links (10G and 1G) with the testbed network.

Figure 2·4a shows the FPGA architecture. Custom logic is referred to as Virtualized FPGA Resources (VFRs), while Shell logic is referred to as static logic. The latter is controlled by the Provider and implements the following blocks:

- Input Arbiter: As shown in Figure 2·4b, the Input Arbiter moves incoming

unicast (multicast not supported) packets to their corresponding VFR. The CAM is used to lookup this VFR based on the Destination MAC field in the packet header. If it does not map to any VFR, the packet is dropped.

- Soft Processor: Interfaces the CPU Agent over UART and is used to set MAC values and their corresponding VFR IDs in the CAM.
- Output Queue: This adds the source MAC addresses to outgoing packets to ensure they cannot be spoofed.
- VFR Wrapper: Shown in Figure 2·4c, the VFR wrapper implements interfaces between individual VFRs and Network/DRAM. Moreover, it prevents bus values for these interfaces from changing when the VFR is being reconfigured.
- Memory Operations Queue: This is part of the VFR wrapper and is used to restrict VFR access to physical memory. Each VFR is assigned a limited private address space, and the Memory Operations Queue ensures tenants cannot access memory outside this.

Tarafdar et al

The authors have presented a framework for implementing logical connectivity of offloaded FPGA kernels that may reside on different physical FPGAs with network connectivity. Figure 2·5 shows the connectivity between CPU and FPGA within a node. The PCIe interface is used to program the FPGA through Partial Reconfiguration and perform DMA reads/writes to the off-chip DRAM. Network connectivity has not been illustrated in this figure but, based on the text, it can be derived that both CPU and FPGAs are directly connected to the network. Shell logic is referred to as a "Hypervisor" while custom offloads are implemented in the "Application Re-

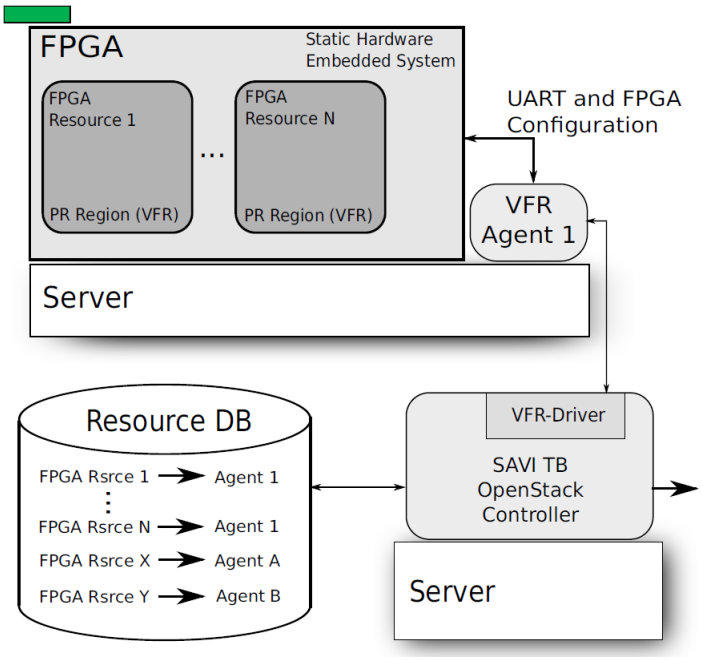


Figure 2·3: FPGA connectivity in the SAVI testbed (Byma et al., 2014). The VFR agent runs on the server CPU and interfaces the FPGA over UART for management. The FPGA also has a direct connection to the data center network (not shown).

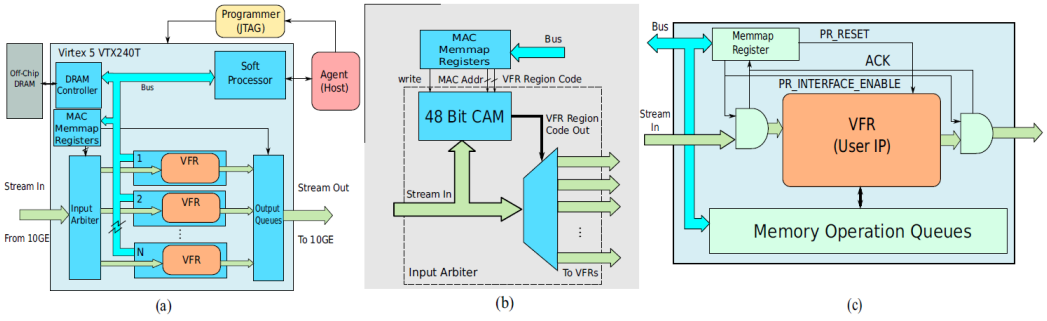


Figure 2·4: a) FPGA architecture showing an overview of important modules and their connectivity. b) The Input Arbiter for filtering incoming packets and mapping them to the correct VFR. c) VFR wrapper for implementing interfaces between VFR and Network/DRAM, and for freezing interfaces during reconfiguration. (Byma et al., 2014)

gion". It is important to note that authors have not implemented support for Partial Reconfiguration in their current design.

Figure 2·6a illustrates the layout of the "Partial Reconfig" region. The FPGA is composed of three blocks outlined below. Of these, Input and Output modules act as wrappers for the custom offloads.

- Input Module: As shown in Figure 2·6b, the Input Module consists of Input Bridge (IB) and Input Demultiplexer (ID) blocks. The former implements a firewall that only allows MAC addresses assigned to the FPGA to pass through. It also replaces Ethernet headers with information regarding the destination kernel within the FPGA. The latter is responsible for sending incoming valid data to the appropriate user kernel. If the destination is on a remote node, it bypasses local kernels and is sent directly to the Output Module.

- User Kernels: This is used to implement custom offloads. Based on Figure 2·6b, we assume that multiple kernels can be implemented within this assigned fabric.

- Output Module: As shown in Figure 2·6c, the Output Module is composed of Packet Formaters (PFs) and an Output Switch. The PF block adds MAC headers to each packet, while the Output Switch uses round robin arbitration to address contention for the egress network port.

2.5.3 cloudFPGA

The cloudFPGA Shell (Weerasinghe et al., 2015; Weerasinghe et al., 2016a; Weerasinghe et al., 2016b; Abel et al., 2017) by IBM Zurich targets disaggregated deployment of FPGAs in hyperscale data centers. By decoupling FPGAs from host CPUs and directly connecting them to the network, clusters of multiple stand alone FPGAs can

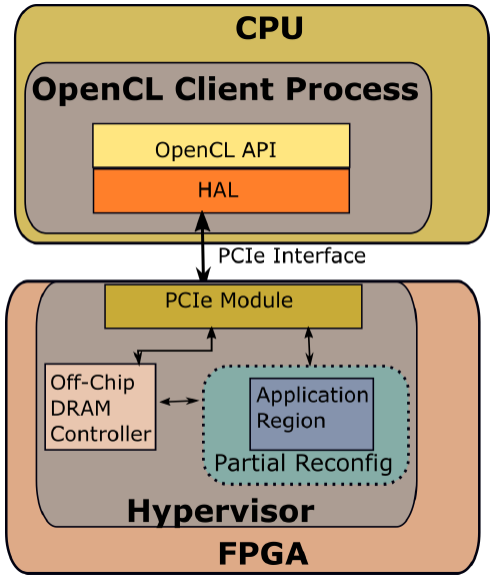


Figure 2·5: Overview of a SAVI testbed node, hypervisor configured in the FPGA, and CPU-FPGA connectivity (Tarafdar et al., 2017)

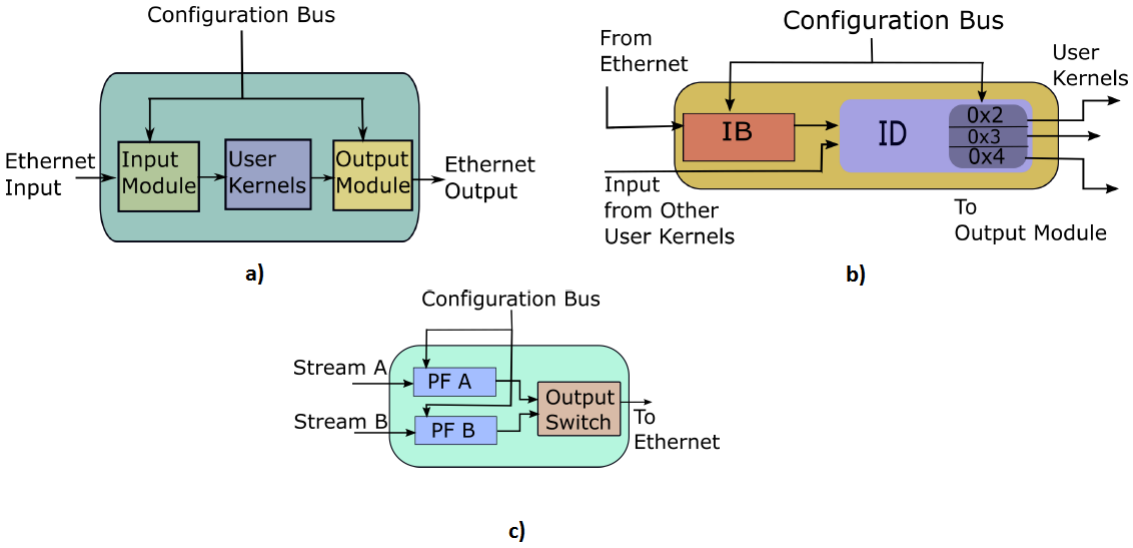


Figure 2·6: a) Layout of the Partial Config region. b) Overview of the Input Module. c) Overview of the Output Module. (Tarafdar et al., 2017)

be constructed. The authors claim that disaggregation enables greater compute capacity to be implemented within the same volume and power budget. Figure 2·7 gives an overview of cloudFPGA node. The FPGA is directly connected to the Data center traffic, independent of CPUs. The off-chip DRAM is assigned the same MAC and IP address as the FPGA, but a different port number, and can be accessed by both local logic and remote devices. The node also stores bitstreams in a nonvolatile flash memory; rebooting the FPGA reconfigures it using the stored bitstream. The authors also claim that the node contains optional scratchpad memories (not shown). Custom offloads are referred to as "user logic'. The user logic in turn can contain 1 or more "vFPGAs". Figure 2·8 shows the intra-FPGA connectivity of the cloudFPGA system. The Shell logic is primarily composed of Network, Memory and Application interfaces which are responsible for moving data between vFPGAs and memory/network. For the network interface, the authors have used static ARP tables, static UDP/IP tables, and vFPGA-MAC-IP tables (which contain MAC and IP addresses of each vFPGA and are dynamically assigned).

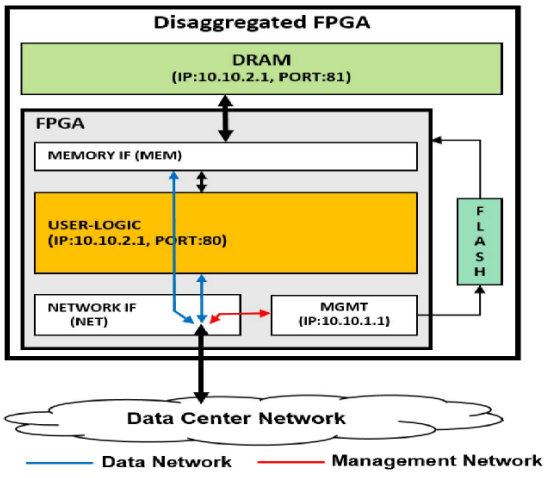


Figure 2·7: Overview of cloudFPGA node(Weerasinghe et al., 2016a)

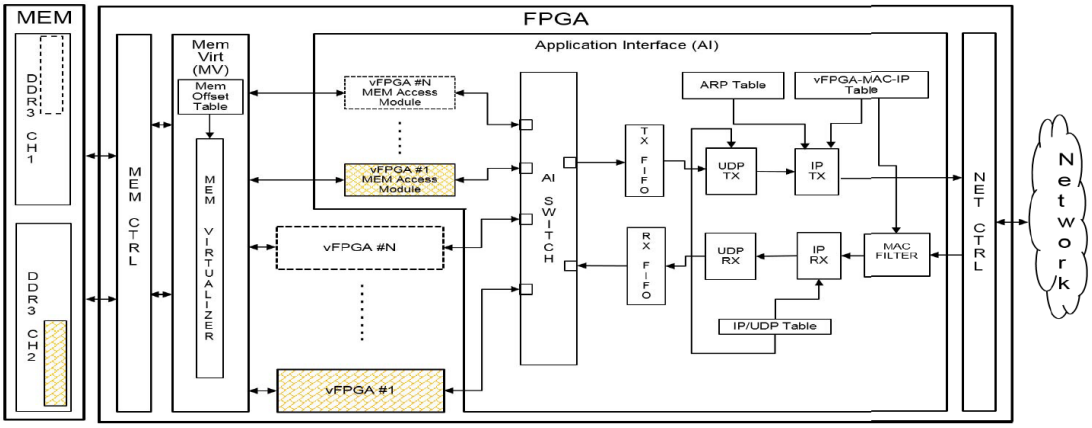


Figure 2·8: FPGA Architecture and connectivity for cloudFPGA(Weerasinghe et al., 2016a)

2.5.4 AWS F1

Amazon's AWS F1 (Amazon, 2018) instances are one of the most popular production FPGA deployments, with a global user base/community. Figure 2·9 illustrates the intra-FPGA connectivity. Custom offloads are called Custom Logic (CL). The Shell contains the following major components:

- Management Physical Function (MGT PF): "The Management PF provides access to various control functions like Virtual-LED, Virtual-DIP Switch, Virtual JTAG, FPGA metrics, and AFI management (load, clear, etc...)."
- Application Physical Function (APP PF): This is used to implement functionality for custom offloads, such as DMA.

2.5.5 Novo G#

Novo-G# (George et al., 2016) is a dual network multi-FPGA system, with independent networks for CPUs and FPGAs. The focus of this work is research based i.e. to explore integration of communication and computation using FPGAs, without being

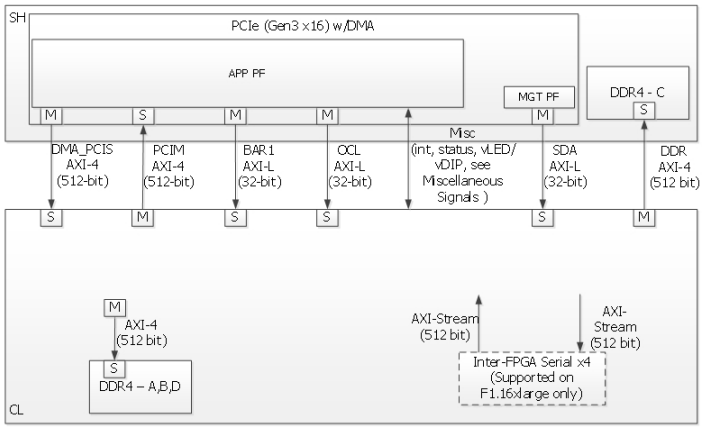


Figure 2·9: Overview of the FPGA Shell in an AWS F1 instance(Amazon, 2019b).

restricted by particular applications, and in a manner that enables general usability and applicability to a wider pool of workloads.

Novo-G# connectivity is similar to Catapult 1, shown in Figure 2·1a. The FPGAs are connected in a 4x4x4 3D Torus (as opposed to 2D Torus for Catapult 1), using 24 high-speed transceivers. There are six connections in total (+x,-x,+y,-y,+z,-z) and so four transceivers are used per connection. The CPUs are connected through a commodity network, and communicate with FPGAs using PCIe. Figure 2·10 shows the Shell architecture. Custom offload logic is referred to as "App Logic" while the Shell itself implements the 3D torus network stack. This stack is composed of two major components:

- Routing Blocks: These are used to route incoming data to egress FIFOs. Each block implements a statically scheduled node-table routing and a Virtual Channel based switch with 4 or 7 stage pipelines (based on supported features).

- Transceiver Blocks: These are low level IP blocks which interface the high-speed transceivers.

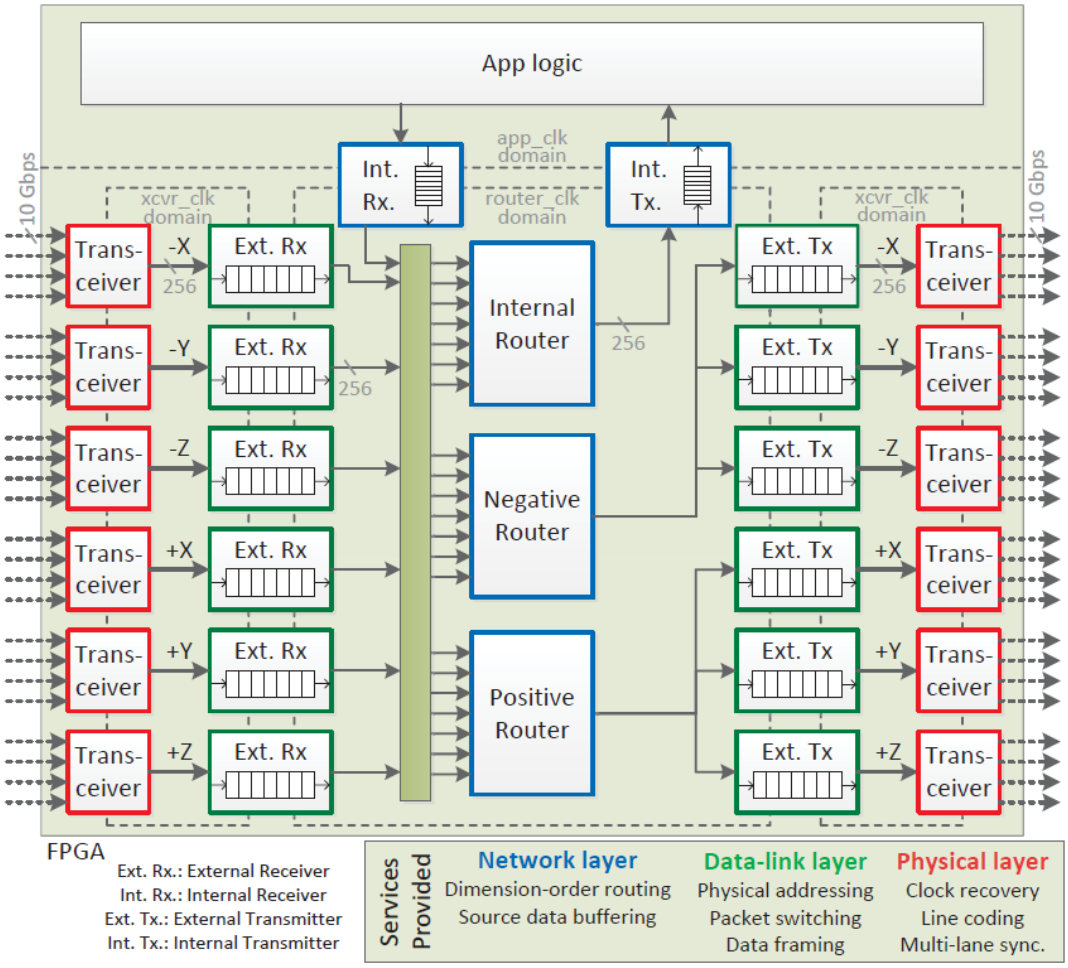


Figure 2·10: Overview of FPGA architecture and connectivity for Novo-G# (George et al., 2016)

Chapter 3

Enabling RTL Simulation for Intel FPGA OpenCL Kernels

3.1 Motivation

One of the primary goals of this dissertation is to achieve fast code generation. As discussed earlier, the design space for FPGAs is massive since both the algorithm and the architecture on which this algorithm executes need to be built. A substantial number of code generation passes are thus spent on first building an initial architecture, and then tuning it to maximize performance and minimize resource usage. In order to achieve fast code generation, we need to reduce the time taken per code generation pass. This is achieved by reducing the overhead of analyzing the effectiveness of code generation; if we can perform this analysis faster, we can do more design/optimization iterations per unit time and converge to an optimal solution in smaller timeframes. This will allow us to not only apply current best practises of writing OpenCL code relatively quickly, but to also leverage the rapid turnaround to explore other approaches that can bridge the performance-programmability gap in HLS.

3.2 Analyzing Code Generation

Here we discuss three alternatives for analyzing code generation. Alternative 1 is to do a full compilation, i.e. synthesis and place& route, and then analyze the resulting

hardware. This is illustrated in Figure 3·1. The drawback of this approach is that hardware generation is placed in the critical path of each design/optimization iteration. This means that each individual iteration can take hours/days based on the complexity of input code, and the overall process can take many months to complete. Alternative 2 is co-simulation; as discussed in chapter 2, this has a number of limitations and hence is not a viable approach. Alternative 3 is referred to as "transplant". While hardware generation can take a long time, the RTL for the design is available within a few seconds/minutes. If we can leverage this RTL to estimate performance and behaviour, by isolating it from the hardware operating system and wrapping it with a custom test environment with negligible constraints, we can achieve both reliable and rapid turnaround.

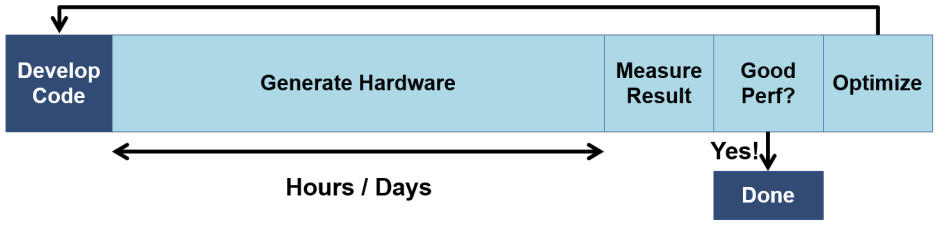


Figure 3·1: Analyzing code generation effectiveness by fully compiling to hardware. Each design/optimization iteration can take hours/days, and the overall process can take many months.

Figure 3·2 illustrates our two approaches for "transplanting" OpenCL kernels. The first approach is a simulation of the full OpenCL kernel by implementing a custom Board Support Packet (BSP), called SimBSP. Unlike a traditional BSP which implements support for memory and host controllers, SimBSP wraps kernel logic with a testbench which initializes kernel parameters and interfaces the data and control signals. The second approach is to isolate application logic from the overall kernel logic and test the former. Kernel logic contains a number of wrappers which can limit the test vectors that can be applied to the actual application hardware. By extracting the OpenCL generated application logic, which we refer to as OpenCL-HDL, we can

overcome this limitation. These two approaches are discussed in detail below.

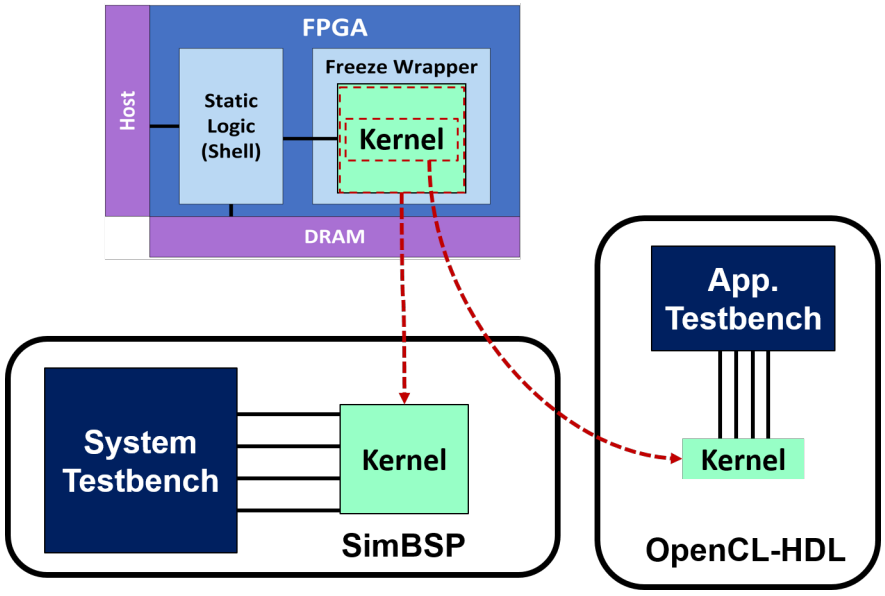


Figure 3·2: Our "transplant" based framework for simulating OpenCL kernel logic (SimBSP) and application logic (OpenCL-HSL).

3.3 SimBSP

In this section, we present our implementation of SimBSP. It is composed primarily of two components, (i) a testbench template that can interface Intel OpenCL generated kernels, and (ii) compilation scripts for generating simulation models and setting up the simulation environment.

3.3.1 Testbench

Here, we discuss details regarding the SimBSP testbench template. We first describe the interfaces exposed by the kernel module (instantiated within the testbench). We then list the configuration registers that are used to set kernel parameters; the testbench must assign appropriate values to these registers before kernel execution can be started.

Kernel Interfaces

Table 3.1 provides details regarding instantiated kernel ports. Clock frequency is determined at compile time based on post-routing timing models and so an accurate value cannot be known for simulation. Therefore, performance estimates made using SimBSP are a measure of compute latencies, instead of actual execution time. *kernel_cra* is an Avalon Memory Mapping (AVMM) slave interface to configuration registers within the kernel while *kernel_mem0* is an AVMM master interface for reads/writes to external memory. *kernel_irq* is a 1-bit flag raised on kernel completion. Finally, the *crc_snoop* streaming slave interface is not used in SimBSP since it is not directly involved in kernel execution.

Table 3.1: Kernel Interface and Descriptions

Name	Type	Interface Description
clock_clk	Clock	Kernel clock
clock_reset_n	Reset	Active low kernel reset
cc_snoop	Streaming	Not used
kernel_cra	Memory Mapped	Interface to configuration registers
kernel_irq	Interrupt	Interrupt to host machine
kernel_mem0	Memory Mapped	Interface to global memory

Configuration Registers

Table 3.2 lists the addresses and kernel parameters that are stored in configuration registers. These registers are 64 bits wide, and the testbench can set the value of an entire register (2 32-bit parameters) every cycle using the *kernel_cra* port. Once all configuration registers have been assigned required values, setting Bit 0 of the register at address 0x0 triggers the start of kernel execution.

Apart from specifying the shape and size of work-items/work-groups, configuration registers are also used to store 64-bit pointers to off-chip memory for kernel arguments. Since the number of kernel arguments can vary for individual applications, addresses from 0x60 onwards can all be used for this purpose.

Table 3.2: OpenCL Configuration Registers

Address	**Bits [63:32]**	**Bits [31:0]**
0x0	-	Start (Bit 0)
0x28	Workgroup_Size	Workgroup_Dimensions
0x30	Global_Size[1]	Global_Size[0]
0x38	Number_of_Workgroups[0]	Global_Size[2]
0x40	Number_of_Workgroups[2]	Number_of_Workgroups[1]
0x48	Local_Size[1]	Local_Size[0]
0x50	Global_Offset[0]	Local_Size[2]
0x58	Global_Offset[2]	Global_Offset[1]
0x60 - end	Argument_Pointer[63:32]	Argument_Pointer[31:0]

3.3.2 Compilation Scripts

In this section, we present the compilation scripts that are used as part of the Intel OpenCL toolflow to enable RTL simulation. There are two such scripts used by SimBSP as shown in Figure 3·3, i.e. simulate.tcl and msim_setup.tcl, while the entire process is divided into three stages. These stages are discussed in detail below. It is important to note that only simulate.tcl is a new contribution, while msim_setup.tcl is automatically generated when compiling for simulation.

- Stage 1: We use the standard command for kernel compilation, i.e. *aoc kernel.cl*, to invoke a C to HDL translation stage. The result of this translation is a QSYS (Intel, 2019) system file which contains the kernel implementation.

- Stage 2: After generating the QSYS file, the compiler automatically runs our custom script called *simulate.tcl*. This script performs three important functions. First, it removes logic that cannot be simulated from the QSYS system; this logic can be safely eliminated since it corresponds to modules that do not impact kernel execution, e.g., System Description ROM. Next, the QSYS file is compiled for simulation in order to generate the required HDL files. Finally, the default testbench is replaced with a SimBSP testbench. At this point, the simulation directories have been set up, and so the command *aoc kernel.cl*

terminates.

- Stage 3: In the last stage, we use Modelsim to manually source the final script, i.e. *msim_setup.tcl*. This will compile the generated HDL (from stage 2) and launch the RTL simulation.

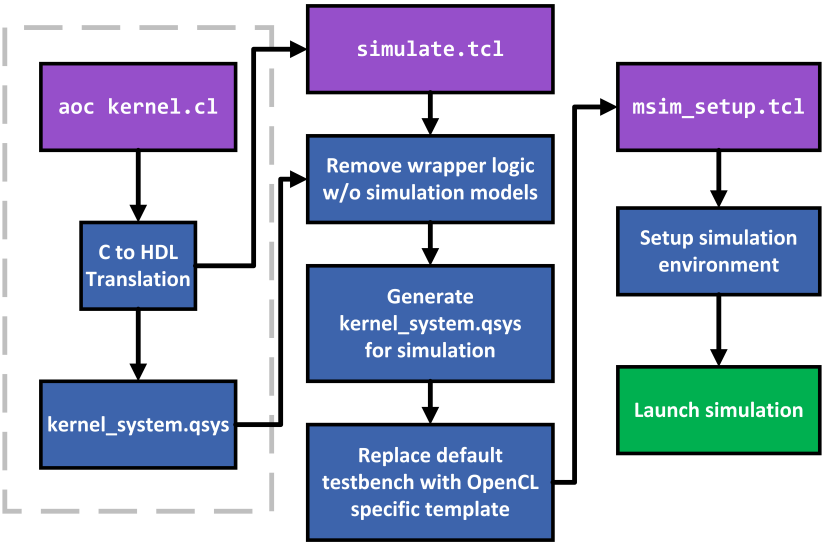


Figure 3·3: The compilation process used by SimBSP to generate simulation models. Blocks in the dashed rectangle represent the standard Intel OpenCL toolflow, while the remaining blocks are specific to SimBSP.

3.4 OpenCL-HDL

OpenCL-HDL is available early in the compilation process. Therefore, similar to SimBSP, we create a custom BSP for OpenCL-HDL which terminates the compilation process once HDL is generated. This prevents unnecessary time spent in compiling the full design, and allows the process to be automated by eliminating the need for manual termination.

The compiler places source files in a directory relative to the kernel file. Specifically this is *[Path to Kernel File]/<kernel_filename>/kernel_subsystem/<kernel_filename*

>_system_140/synth/. The file <kernel_filename>.v contains the implementations for all kernels in the compiled OpenCL source. Other files include <kernel_filename>_system.v, which is a wrapper, and the Intel modules that form the lowest level in the design hierarchy. These building blocks can range from commonly used components, such as FIFOs and Load-Store Units (LSUs), to kernel-specific ones, such as floating point arithmetic units. While Intel maintains a large number of modules, only those identified by the compiler as required are copied here. Before using these files, however, it is important to change their file extension to *.sv* from *.v* due to certain syntax that would otherwise prevent compilation as Verilog. In the remaining sections of this paper, we refer to <kernel_filename>.v as the OpenCL-HDL source.

3.4.1 Basic Blocks

The OpenCL toolflow compiles kernels through multiple modules called basic_blocks, and uses a function module to describes their connectivity. The observed rules for basic_block module generation are as follows. Typically, each loop generates a basic_block module. Nested normal loops will generate their independent modules and are connected by their parent loop module. Unrolled loops will also generate a separate module. However, consecutive unrolled loops, or any unrolled nested loop within an unrolled loop, will not generate a new module.

3.4.2 Interfaces

We broadly categorize common interfaces to be either LSU, Direct, Control, Feedback, or Don't Care. A basic_block can have variations in the number and types of these ports based on individual applications and problem sizes. Modifying interfaces eliminates hardware that was required for interfacing the pipelines with the BSP OpenCL wrappers (and hence does not impact the functionality of our application logic). This reduces resource usage and simplifies data paths resulting in lower

latency. Once the interfaces have been mapped, the module can be used for RTL simulation or integrated into a different HDL shell for compilation.

Load Store Unit (LSU) ports consist of Avalon interfaces to LSU modules. LSU modules source and sink data to application logic and are typically created, one per variable per operation (read/write), when a memory access depends on the outer loop iterator. Both the module, as well as the Avalon protocol, have associated overheads which can be removed by interfacing pipelines directly with variable data. We bypass these LSU modules by creating explicit variable input/output ports and connecting them to the corresponding LSU module's source/sink interfaces (*o_readdata*/*i_writedata*). The Quartus compiler then removes the LSU modules during fitting optimizations.

Direct ports are automatically generated by the compiler to supply data directly to application logic without requiring LSU modules or the Avalon protocol. They can be further categorized into Constants, Variables, and Initialization. Constant direct ports correspond to kernel inputs that are individual elements (instead of pointers). Variable direct ports correspond to cases where the LSU unit is moved outside the basic block. This can occur if the outer loop variable is not used to index/address memory accesses, or if the problem size is too small. Initialization direct ports are used to load initial values for (outer/inner) persistent variables.

Control ports primarily consist of clock, reset, *Stall*, and *Valid* ports. *Stall* ports are used by a basic_block to stall upstream modules, while *Valid* ports are used to stall downstream basic_blocks. Since we isolate the application logic from the overall system, we hardwire both *Stall* and *Valid* ports. As a result, the Quartus compiler minimizes or removes the associated stall hardware during fitting optimizations.

Feedback ports are typically generated to maintain the state of loop carried dependencies. They are created in input-output pairs, with the output feedback

ports being looped back and connected to input ports of the same basic_block. If the OpenCL compiler generates individual feedback inputs (and not pairs), then we hardwire them since these correspond to selecting between initial and run-time values of the loop carried dependencies.

Don't Care ports can correspond to a variety of logic such as parallel control and data paths that do not interact with application logic, or logic that interfaces LSU modules, e.g., address computation. Leaving Don't Care ports unconnected optimizes away these unnecessary resources.

3.5 Validation

We validate our approach using RTL simulation and comparing the results of the simulation with those obtained through Emulation (discussed in chapter 2) for the same test test vectors. For example, Figure 3·4 shows the waveforms obtained from compiling a simple matrix multiply kernel using SimBSP. The final results for the RTL matched those from CPU, within a reasonable difference due to floating point computations.

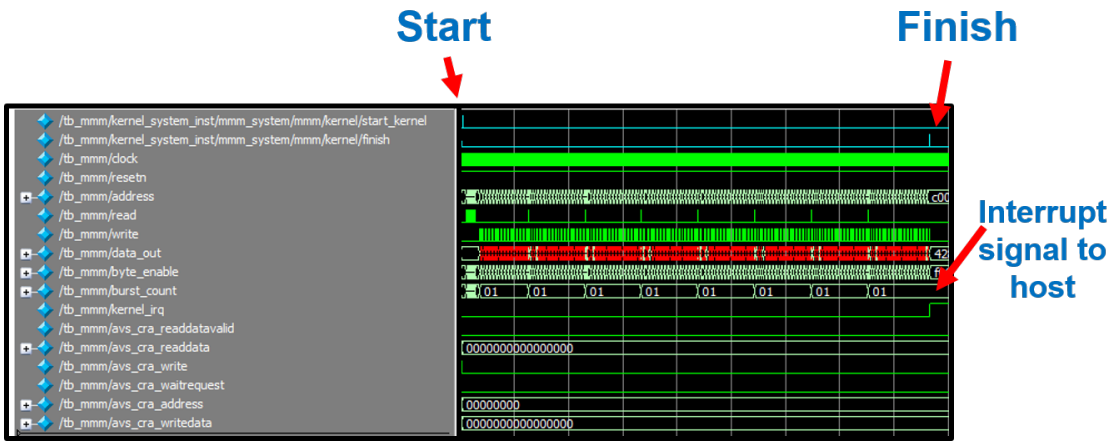


Figure 3·4: Waveforms for RTL simulation of a Matrix Matrix Multiplication kernel

3.6 Impact Contribution

As illustrated in Figure 3·5, use of SimBSP and OpenCL-HDL moves hardware generation out of the design/optimization iterations. As a result, we can now obtain substantially more data points for our design space exploration within a give timeframe, which results in a faster convergence to the optimal solution as compared to Figure 3·1.

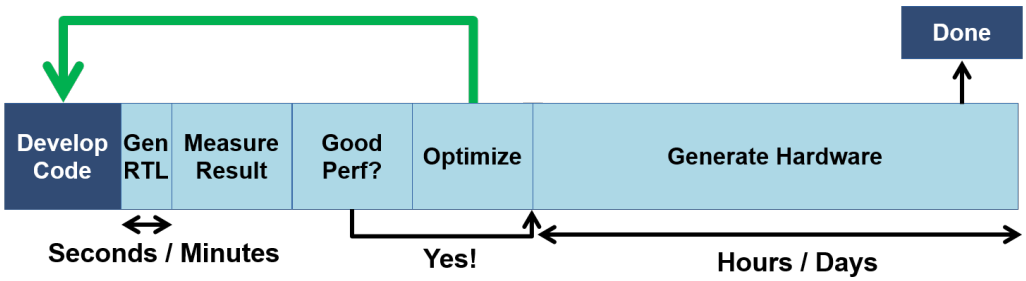


Figure 3·5: Use of our RTL simulation framework can reduce the time taken per design/optimization iteration from hours/days to seconds/minutes.

3.7 Conclusion

In this chapter, we presented a "transplant" based framework for generating RTL test systems. This was composed of two parts. The first was integration of full OpenCL kernel simulations into the Board Support Package due to the standard interfaces exposed by wrapper logic. The second was generating OpenCL-HDL by decoding ad hoc interfaces of application logic to reduce the complexity of interfacing it. As a result of this contribution, we can perform experiments towards exploring OpenCL optimizations in substantially small timeframes.

Chapter 4

IP Generation using HLS toolflows

4.1 Motivation

A number of 3rd party vendors offer IP blocks for implementing various functions. These include both open source IP, e.g. OpenCores (OpenCores, 2019), commercial ones (Amazon, 2019a; Zipcores, 2019; Digital Blocks, 2019; Algo-Logic, 2019; Avnet, 2019), as well as native ones shipped with standard FPGA development tools e.g. Intel FPGA IP cores (Intel, 2019). Native IP blocks, in particular, typically offer the best possible performance since they are designed by engineers familiar with the internals of the FPGA. Developers can connect multiple of these IP blocks together to implement target architectures. Using software tools like Qsys (Intel, 2019), the complexity of designing even an interconnect fabric can be substantially reduced. The drawback of IP blocks, however, is that the utility of these blocks depends on how well they match developer requirements. This is because customizing IP cores is not a viable solution; IP cores only support modifications to a limited set of parameters in order to maintain predictable performance/behavior and protect intellectual property. Two exceptions to this are open source IP, which do not provide the same performance guarantees as commercial and native ones, and Intel/Vivado HLS, which are not a viable solution as discussed in chapter 2.

In this chapter, we explore the use of OpenCL-HDL to generate custom IP. This enables greater application specificity than vendor IP, and can even be used to build custom Domain Specific Languages. We provide two case studies that use OpenCL-

HDL: i) 3D FFT and ii) Multi Layer Perceptron Inference. The validation of our approach is similar to that in chapter 3 i.e. functional validation by comparing Emulation results against RTL simulations.

4.2 Case Study 1: 3D FFT

In this case study, we implement a broadside 64 point Radix-2 FFT module using Intel OpenCL that outperforms CPUs, GPUs, and even HDL for FFT sizes that can fit on a single FPGA without folding. Once the compiler translates code into HDL, we isolate application logic from the overall OpenCL system using the processes described earlier. We also show that this generated logic can be seamlessly integrated into existing 3D FFT shells that were initially constructed for IP core based designs (Which are typically pencil based FFT).

Algorithm 1 gives an overview of our baseline 1-D Radix-2 FFT code based on OpenMM (Eastman and Pande, 2010). Each N-element input vector is processed for log(N) stages. For a given stage and iteration, vector elements are processed in pairs; each pair writes two values to the input vector for the subsequent stage. The variables L and m are used to determine these input and output pairs, as well as corresponding twiddle factors. The entire computation is done using two float2 arrays in a ping-pong manner, with the buffers swapped after every stage.

Algorithms 2 and 3 show the result of applying our code transformations to Algorithm 1. All loops are fully unrolled and the log(N) stages of the computation are evaluated concurrently. Individual arrays are used to store the result of each stage, which not only ensures that they are small enough for corresponding pipeline registers to be inferred as registers, but also helps infer pipelines which may not be implemented effectively using the ping pong buffer approach in Algorithm 1. A large number of intermediate variables are also used to simplify dependency analysis.

Algorithm 1 Radix-2 1D FFT Baseline Kernel

```
1: N ← FFT Size
2: for Array of 1D Vectors →j = 1 : M do
3:     float2 data_0[N], data_1[N]
4:     Initialize data_0
5:     int L = N/2
6:     int m = 1
7:     for Stage →s = 1 : log(N) do
8:         data_1 ← COMPUTE(data_0,L,m)
9:         Swap (data_0 , data_1)
10:        L = L >> 1
11:        m = m << 1
12:    end
13: end
```

Moreover, since the size of interest for the FFT implementation is known beforehand, we implement twiddle factors as constant arrays instead of inputs to the kernel or real-time computations.

Algorithm 2 Optimized 1D FFT Kernel

```
1: N ← FFT Size
2: for Array of 1D Vectors →j = 1 : M do
3:     float2 stage_0[N] .... stage_logN[N]
4:     Initialize stage_0[N]
5:     int L = N/2 , int m = 1
6:     #pragma unroll
7:     for Loop 1 →k = 0 : (N/2) − 1 do
8:         stage_1 ← COMPUTE(stage_0,L,m)
9:     L = L >> 1 , m = m << 1
10:            ⋮
11:    #pragma unroll
12:    for Loop logN →k = 0 : (N/2) − 1 do
13:        stage_logN ← COMPUTE(stage_logN-1,L,m)
14: end
```

Figure 4·1 shows the processing element generated for each iteration of each compute loop in our kernel. FIFOs implemented using registers are used to source and sink data to the pipelines, along with providing delays to synchronize pipelines and

Algorithm 3 Compute Function Code

```
1: for base = 0 : (N/2) − 1 do
2:     int j = base/m; int k = j*32/L; int p = (j+1)*m;
3:     float c0_r = stage0_r[base];
4:     float c0_i = stage0_i[base];
5:     float c1_r = stage0_r[base+32];
6:     float c1_i = stage0_i[base+32];
7:     stage1_r[base+j*m] = c0_r+c1_r;
8:     stage1_i[base+j*m] = c0_i+c1_i;
9:     float c2_r = c0_r-c1_r; float c2_i = c0_i-c1_i;
10:    stage1_r[base+p] =w_r[k]*c2_r - w_i[k]*c2_i;
11:    stage1_i[base+p] =w_r[k]*c2_i + w_i[k]*c2_r;
12: end
```

reducing the critical path. The compiler also infers the two different forms of dot products and generates appropriate modules (Dot2 and MDot2). As discussed before, for twiddle factor values (w_r and w_i) equal to -1, 0, or 1, dot product units are omitted and extra depth is added to the first input FIFO of the data-path. Processing elements are placed in a data parallel manner for a given stage and cascaded to implement inter-stage stall-free connectivity. Therefore, the FFT unit can sink and source an entire N-element 1-D vector every cycle.

Given a 1D FFT module generated using our approach, we now demonstrate that both OpenCL-HDL and IP core based designs can utilize the same shell for performing a 3D FFT. Consequently, the former can be seamlessly integrated into existing logic initially designed for the latter. Since FFT is a linear operation, a high dimensional FFT can be broken down into a series of 1D directional transforms in any order. In order to avoid constructing complex and expensive memory structures that can stream data every cycle in all dimensions, a transpose is performed on the overall directional FFT result to reorder data within buffers. This reordering rotates the grid and allows a different directional 1D FFT to be performed for the same data access pattern.

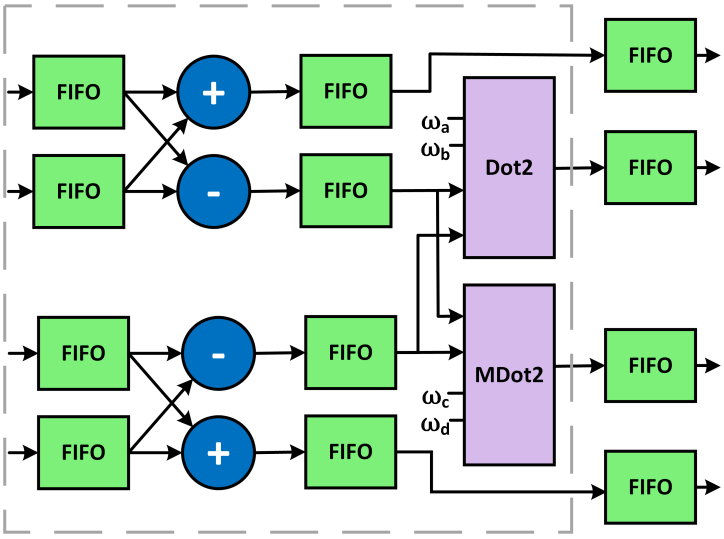


Figure 4·1: OpenCL-HDL processing element architecture generated by the OpenCL compiler. Dot product units are inferred and implemented instead of individual adders/multipliers. Twiddle factors are declared as constants and hard coded to corresponding inputs of the dot product units.

Figure 4·2 shows the typical architecture for performing a 3D FFT using 1D FFT modules. A set of ping-pong buffers are used to source and sink vectors. The initial grid is loaded into Buffer 1 while the final result can be read from Buffer 2. By providing a variable data path for writes to both buffers using MUXs, the 3D FFT shell can be used as an intermediate stage for a number of different applications .

Figure 4·3 shows the detailed organization of Buffer 1 and 2 for 4 element, 1-D FFTs in the x-dimension. The throughput of both IP core and OpenCL-HDL Radix-2 implementations is constrained to be that of the Radix-2 design. Therefore, for an N^3 FFT, buffers are constructed using N banks of size N^2 each in order to stream the required N elements per cycle. The crossbar performs a transpose using a parallel to serial transformation since all N results per cycle are written to the same bank. It can operate stall free since the same output bank is accessed once every N cycles. From the figure, we observe that FFTx for OpenCL-HDL Radix-2 is the same as FFTy for IP cores. By extension, $FFTy(OpenCL - HDL) = FFTz(IPCore)$ and

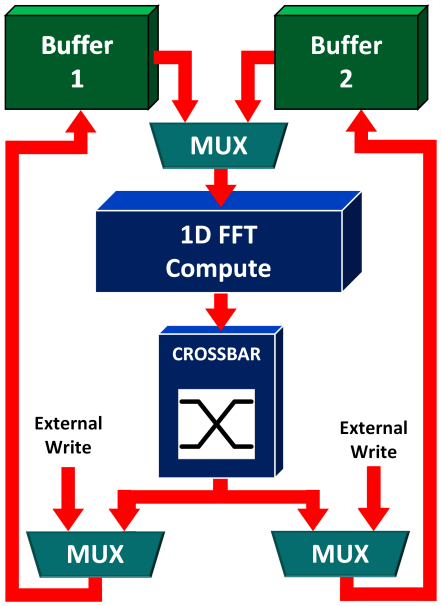


Figure 4·2: Architecture for a 3D FFT using 1D FFT compute pipelines.

$FFTz(OpenCL-HDL) = FFTx(IPCore)$. Since FFT is a linear operation and the order is independent, $FFTxyz(IPCore) = FFTzxy(OpenCL-HDL)$. This means that we can replace FFT IP cores with OpenCL-HDL designs without having to modify almost any data or control paths in existing 3D FFT implementations. One potential exception (to the control path) is adjusting for differences in loading and unloading latencies.

We evaluate the effectiveness of our design by implementing the FFT module on an Intel Arria10X115 FPGA with 427,200 ALMs, 53Mb of BRAM, and 1518 DSP blocks. OpenCL code is compiled using Intel OpenCL SDK v16.0.2. The IP Core used is (Intel, 2010). Our CPU code is implemented on a fourteen-core 2.4 GHz Intel Xeon E5-2680v4 with ICC compiler and MKL DFTI (Intel, 2018a). The GPU used is NVIDIA TESLA P100 PCIe 12GB. It has 3584 Cuda cores and peak bandwidth of 549 GB/s. FFT code is written using cuFFT library (NVIDIA, 2018) and compiled with CUDA 8.0.

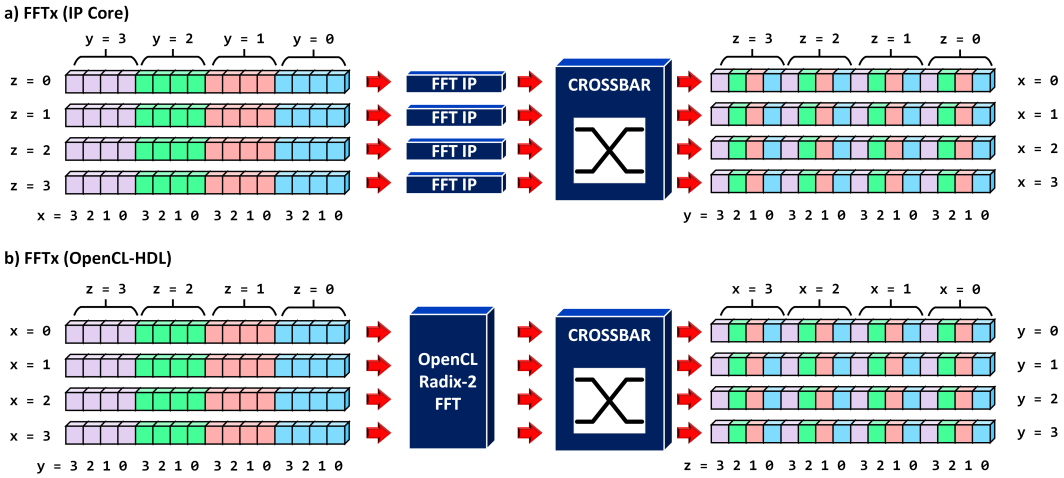


Figure 4·3: Data source and sink patters for (a) IP Cores and (b) OpenCL-HDL pipelines. The same 3D FFT shell can be used for both since the access pattern is persistent. Only the dimensional order of FFT changes, but since FFT is a linear operation, the final result will remain the same.

Table 4.1 and 4.2 show the latency and resource usage for OpenCL-HDL and IP Core designs for $8^3, 16^3, 32^3$ and 64^3 FFTs. Figure 4·4 further illustrates the IP core resource usage as ratio of OpenCL-HDL usage for these FFT sizes. From the results we can see that OpenCL-HDL is significantly more resource efficient, with an average of 7.5x fewer ALMs used and 1.6x fewer DSP blocks. For N=64, we could not fit N parallel IP cores when using hard DSP blocks. Therefore, only 50 were implemented on DSP blocks while the remaining used ALMs. With regards to latency, the IP Core takes $2N$ cycles since input and output ordering is set to *Natural*. On the other hand, OpenCL-HDL latency grows by an average of only 1.4x for every 2x increase in FFT size.

Table 4.1: Latency and Resource usage for OpenCL-HDL 1D FFT

FFT Size	Latency (cycles)	ALM	DSP
8	20	1,849(<1%)	56(4%)
16	37	4,387(1%)	168(11%)
32	41	7,237(2%)	456(30%)
64	53	18,705(4%)	1160(76%)

Table 4.2: Latency and Resource usage for IP-Core 1D FFT

FFT Size	Latency (cycles)	ALM	DSP
8	16	26,759(6%)	96(6%)
16	32	11,132(3%)	256(17%)
32	64	63,322(15%)	832(55%)
64	128	176,285(41%)	1412(93%)

Finally, we compare the total execution time for $16^3, 32^3$ and 64^3 OpenCL-HDL against CPU, GPU and IP core implementations. Results (Table 4.3) show that the average speedup achieved is 29x vs CPU-MKL, 4.1x vs GPU cuFFT and 1.1x vs IP Core FFT.

Table 4.3: Execution Time (us) for 3D FFT Implementations

Design	16^3	32^3	64^3
CPU	22.0	55.0	288.0
GPU	20.7	23.6	43.1
IP Core	1.8	6.8	31.1
OpenCL-HDL	1.8	6.6	25.8

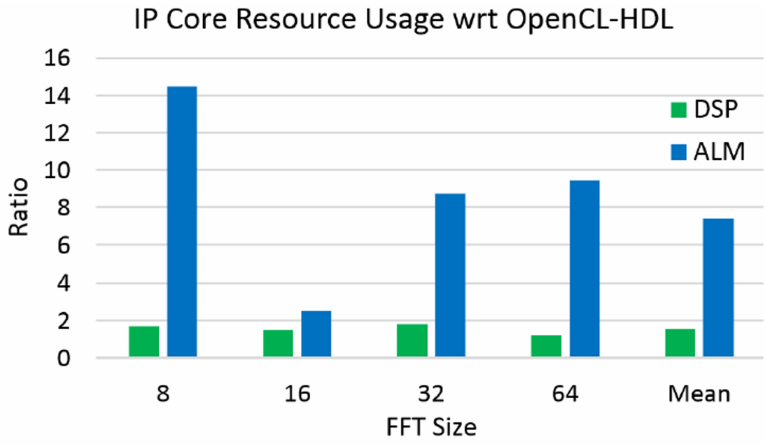


Figure 4·4: IP core resource usage with respect to OpenCL-HDL. OpenCL-HDL designs consume both fewer ALMs and DSPs.

4.3 Case Study 2: Multi Layer Perceptron Inference

Thus far, we have presented a model for eliminating optimization blockers which enables simple representations of arbitrary applications to compile to efficient hardware. In this chapter, we explore the impact of this approach on specialization i.e. the ability to design hardware that has been tuned for an application of interest. Specifically, we provide a detailed case study on building an inference engine for Multi Layer Perceptrons (MLPs), a type of Deep Neural Network. The typical approach here has been to use IP cores, parameterizable HDL templates or dedicated ASICs. In all these cases, the resulting architecture provides limited opportunities for application specific optimizations. To build a specialized system, we leverage the optimizations from the previous chapter for Intel OpenCL, as well as methods for extracting OpenCL-HDL. As discussed in detail later, we do not build a full OpenCL system; using OpenCL-HDL allows us to independently explore the optimization space of individual components and component interfaces.

4.3.1 Background

Machine learning (ML) is playing an integral part in solving key scientific problems, including finding of cancer treatments (Argonne, 2017), performing weather simulations (Balaprakash et al., 2014), and designing new nanocatalysts (Sen et al., 2017). Multi-Layer Perceptrons (MLPs) are an important subset of ML that optimize existing applications to meet more stringent requirements of reliability, performance, complexity, and portability. MLPs consist of multiple layers of firing neurons, with each layer using responses of previous neurons as stimulus. Use of MLPs is divided into two stages: training and inference. Training determines neuron connection strengths in a given MLP by iteratively converging to values that minimize evaluation error. Inference, on the other hand, refers to the process of performing classification (or

regression) on a set of test input cases using the trained model. In our work, we focus on reducing the evaluation time for real-time inference models. While training has large computation timeframes, potentially reaching the order of days or weeks, reducing these is not critical to the application. On the other hand, meeting real-time inference timing constraints is integral to success, especially in end-user-facing services (Jouppi et al., 2017).

Since real-time inference prefers latency instead of throughput (Patterson, 2004), it is difficult for traditional processors to implement complex systems that can meet the application latency bounds.For CPU-based implementations, batch processing is not possible due to latency bounds and processing individual test cases results in a large number of cache misses. This is because MLPs have virtually no data reuse for the same test case. GPU performance is also hurt by low data reuse and batch-less processing since the computations may not be sufficiently parallel to fully utilize the thousands of available cores. For cores that are assigned work, a significant number of cycles are likely to be idle as threads wait for off-chip data to be accessed. ASIC-based designs have typically managed to fill this gap by providing massive amounts of resources and specialized pipelines that are tailored for Deep Neural Networks (DNNs); one example is the Google TPU (Jouppi et al., 2017). However, as the number of diverse applications and their associated models grows, these ASICs effectively address a broad domain, rather than a particular application, and hence are unable to use application-specific optimizations at the level needed.

Reconfigurable architectures, such as FPGAs, are becoming increasingly popular for Machine Learning in general, and MLPs in particular. Application developers can generate logic for their desired model, and ensure that the resulting compute pipelines are optimal, for specific MLP application. Designs can be scaled and transferred to newer generation technologies by recompilation. Moreover, complex HDL codes do

not have to be written; rather designs can be generated with simple scripts because variations across different MLP models, i.e. number and sizes of layers, can be easily parameterized. Advances in FPGA capability are resulting in real-time MLP implementations transitioning from being memory bound to compute bound. On-chip memories are now big enough for all model parameters to fit on-chip. Moreover, FPGAs offer support for most common high-speed protocols and thus minimize the latency of memory and I/O transactions by supplying data directly (from sensors) to compute pipelines (George et al., 2016; Sheng et al., 2018a).

In our work, we explore the application-aware optimization space for real-time compute bound MLP inference processors using our proposed modular architecture. Minimizing latency requires all pipelines to operate both stall-free and at high bandwidth. The former is achieved by ensuring that modules in the design source/sink data at rates that are constrained by the latter. We have observed that since modules in MLP architectures are tightly coupled and operate within the same clock domain, constraints can occur even for indirect connectivity. Therefore, by selecting higher interface bandwidths for particular ports, the complexity (and hence latency) of multiple modules can become large. This increase in complexity can outweigh the benefits of higher module throughput and thus increase the overall latency of the computation. By identifying modules in the critical path and their interconnectivity, we can determine and optimize parameters for a given application in order to minimize the latency of the entire model evaluation. Our contributions in this case study are:

- We propose a modular architecture for application specific MLP inference processors on FPGAs. Components can be easily added, updated, or deleted to facilitate exploration of the optimization space.

- We identify tightly coupled design parameters that are critical to performance and develop a latency model to estimate the impact of varying them.

- We develop OpenCL templates for component generation to ensure standard optimizations are applied and reduce the programming effort.
- We present a state machine based control module that can be automatically generated and used to orchestrate the flow of computations without requiring external instructions or feedback from compute pipelines.
- We demonstrate the effectiveness of our architecture using the MNIST, Poker, and ECP-Candle P3B1-TL benchmarks.

4.3.2 Related Work

FPGA-based MLP implementations have received significantly less attention than other DNNs such as Convolutional Neural Networks (e.g., (Geng et al., 2018b; Geng et al., 2018a)). The most prominent design is Microsoft's FPGA-based inference architecture, Brainwave (Chung et al.,), which target low compute-to-data ratio DNN applications such as MLPs, Long Short-Term Memories (LSTMs), and Gated Recurrent Units (GRUs). The memory bandwidth bound is alleviated by using on-chip block RAM for weight matrix storage. For an 8-bit integer implementation on Stratix V FPGAs, Brainwave achieves 2.0 TOps/s, while on Stratix 10 FPGAs they claim to have 31 TOps/s performance running at 500 MHz. The drawback of Brainwave is that it uses HDL templates for generating hardware for models.

Ortigosa, et al. (Ortigosa et al., 2006) present an MLP architecture which addresses the memory bound by storing 8-bit weights on-chip. Their design consists of multiple processing elements, each having a single multiplier and accumulator, that operate on individual neurons. Resulting outputs of the processing elements are converted from 24 bits to 8 bits using a constant scaling factor. This approach is suboptimal since the range of output values depends on both input data and the layer under evaluation, and a single predetermined factor can result in not being able

to utilize the entire 8-bit range (with a consequent loss of accuracy). In our design we compute this scaling factor at run-time for individual layers and input vectors (similar to (Tensorflow,)), and ensure that the entire data-range is used.

In (Panicker and Babu, 2012; Taright and Hubin, 1998), the authors propose FPGA-based MLP architectures, but their work serves as a proof-of-concept and is also constrained by off-chip memory bandwidth. Sharma et al. present an automated DNN design generation framework, DNNWeaver (Sharma et al., 2016), which also depends on DRAM access speed for performance. Moreover, the DNNWeaver compute units are constrained to Digital Signal Processor (DSP)-only implementations and logic cells are not used for ALUs. Gomperts et al. introduce a general purpose architecture for MLP based on FPGAs (Gomperts et al., 2011). Their design generates individual processing elements for each layer, which is not feasible for large neural networks where resources on a single FPGA may not even be sufficient to compute a single layer in parallel.

To the best of our knowledge, there is no existing work on optimization space exploration of design parameters for compute bound FPGA based MLPs.

4.3.3 Multi Layer Perceptrons

In this section, we provide an overview of Multi-Layer Perceptron based neural networks using both logical and computational models. Multi-Layer Perceptron models are typically composed of an input layer containing feature values measured using sensors, an output layer containing the diagnosis result, and, potentially, multiple hidden layers that perform the required computations on the input data. Each layer consists of one or more neurons, depending on the model. MLPs are fully connected as illustrated in Figure 4·5a: a neuron in any given layer is connected to the outputs of all neurons in the previous layer.

Inputs to the hidden and output layer neurons are scaled and accumulated by

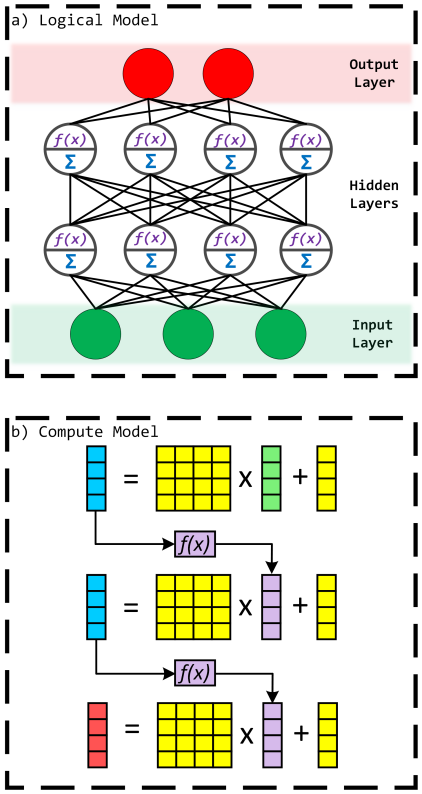


Figure 4·5: Illustration of (a) Logical and (b) Compute models of Multi-Layer Perceptrons

using weights (connection strengths) that are determined during training. A non-linear function, called the activation function, is applied to the result, which then becomes the neuron output. Figure 4·5b shows this operation represented as a Matrix Vector Multiplication (MVM). Each layer has a unique weight matrix which contains connection strengths and a bias vector. Layer inputs are the result vector from the previous layer, or the input vector if it is the first hidden layer. A given row from the $X \times Y$ weight matrix for a given layer represents the connection strengths of Y neurons in the previous layer assigned to one of the X neurons in the current layer. The activation function is applied to individual elements of the output vector. During the training process, all data are floating point. Classification can be performed by using integer arithmetic without loss of accuracy.

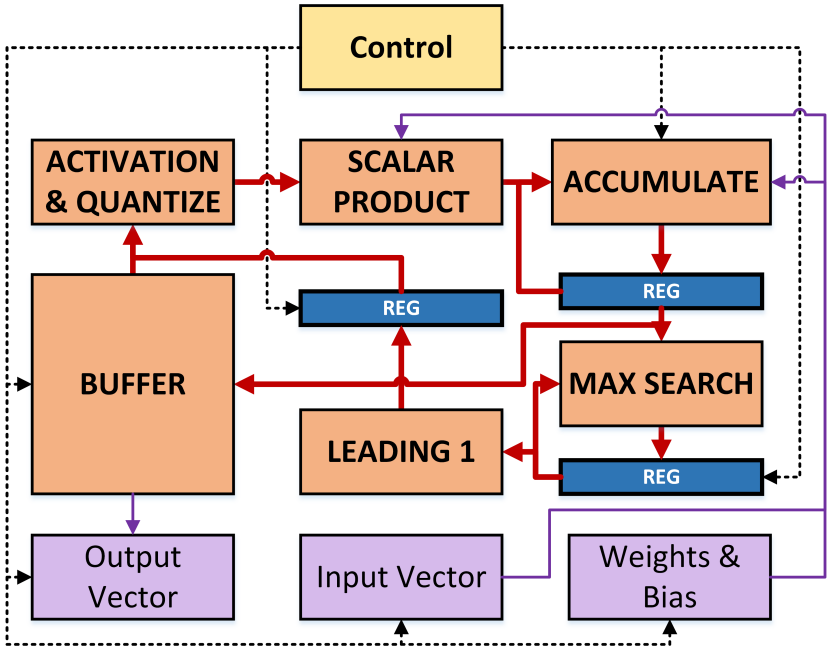


Figure 4·6: Architecture of the proposed low latency MLP inference system. The modular approach and well defined boundaries/interfaces enables updates, additions, and deletions to be performed easily and with minimum changes to adjacent components

4.3.4 Proposed Architecture

In this section, we provide an overview of the proposed MLP inference architecture (see Figure 4·6).We use a modular approach to component design that enables parameters to be varied in order to implement optimal component sizes for a given application. Compute and control planes are segregated, enabling additions, deletions, and updates to be performed easily. Individual components within each plane also have well-defined boundaries and interfaces to ensure that design changes can be made at large granularity, with minimal effort, and without necessitating changes to other logic beyond parameter updates. The MLP inference architecture processes layers sequentially, with modules performing all computations for a given layer before evaluating the next.

Control

One of the benefits of a highly application-specific architecture is the ability to design cycle accurate event triggers. These state machines can control data flow without feedback from the compute modules and have seamless transitions between different evaluation stages. By having an instruction-free implementation, overhead of fetching and decoding instructions is avoided and end-to-end data flow for the entire application can be made stall free.

Memory

Memory modules are composed of multiple blocks of on-chip Block RAMs (BRAMs). FPGAs have thousands of BRAMs that can be used either independently or connected together to form larger memories and/or complex data structures. The BRAM architecture enables data to be supplied stall-free to the Scalar Product module. Trained module parameters are initialized in these BRAMs as part of the bit-stream which configures the MLP architecture on the FPGA.

Scalar Product

As mentioned previously, the system does not benefit from data reuse and is thus batch-less. Consequently, we replace the traditional Matrix-Matrix multiplication component with individual Scalar Product modules. Developers can specify both the number of Scalar Product evaluations and their size. Computations are performed using 8 bit variables while results are accumulated into 32 bit outputs. In order to minimize latency, we use tree based structures for implementing the Scalar Product modules instead of systolic arrays. This ensures that we can scale well to larger input vectors.

Accumulate

If the size of a Scalar Product module is smaller than the input vector, multiple iterations are required to accumulate partial sums and obtain a final value. Moreover, a bias value must be added to the sum. The Accumulate module performs all of these functions by using dedicated *Accumulate Registers*. It receives triggers from the control plane on whether to accumulate or re-initialize the registers for a new operation cycle.

Activation & Requantization

The Activation & Quantization module reads data from the buffer, performs 32-bit ReLU activation ($RELU(x) = max(x, 0)$), and then quantizes data back to 8 bits for the next layer. Quantization is performed by using truncation because of the high costs of division hardware. Because of the nature of the operation (compression), the difference in results is small in this particular context. Moreover, ReLU activation ensures that the Most Significant Bit (MSB) of our 8-bit result is always 0. Therefore the effective compression target is 7 bits, which further reduces the difference between division and truncation results.

Max Search

Being able to perform quantization requires knowledge of the upper and lower data limits. Because of the ReLU activation, we are guaranteed a lower limit of 0. Searching for the upper limit must be done without stalling the data stream. Consequently, we use the Max Search module to perform local maxima searches on data as it becomes available and update an associated register if a local maximum exceeds the current global maximum. This approach ensures that latency is based on the dimensions of the accumulator outputs and not the full input vector. Employing a tree-based search further reduces the delay.

Leading 1

Once the maximum value for a given output has been determined, we use the Leading 1 module to find the most significant non-zero bit and use this position to perform truncations for quantization. The output is constrained to be between 6 and 30 (on a scale of 0-31) since the former means all values are already within 8 bits while the latter represents the largest possible positive numbers. As with Scalar Product and Max Search, the evaluation is performed with logarithmic complexity.

Buffer

The Buffer module stores result vectors for both the current and the previous layer (input to current layer). While the Buffer is used purely as a memory resource, it is included in our compute plane because of its tight coupling with the Accumulate and Activation & Quantize modules. It is implemented by using registers in order to meet throughput demands for architecture-specific source and sink sizes. A two-bank architecture comprising of separate input and output memory banks is used. The output memory bank stores results from the previous layer and supplies this data to the Activation & Requantization module. On the other hand, the input memory bank stores results of ongoing computations by sinking data from the Accumulate Registers.

Critical Path

Of the compute plane modules discussed above, all but the Buffer lie in the critical path. We divide these modules into two categories; the Variable Critical Path (VCP) and the Persistent Critical Path (PCP). The Variable Critical Path is entirely the Scalar Product module. It is equal to the number of calls needed for the Scalar Product module to perform all multiplication operations. Since it is based on the relative dimensions of the weight matrix and Scalar Product module, it will vary

for each layer. Once the last set of results has been produced, modules that need to evaluate this last result before a new layer can be processed are referred to as the Persistent Critical Path. It corresponds to a fixed number of cycles independent of layer dimensions. Applicable modules include Accumulator (with register), Max Search (with register), Leading 1, and Activation & Quantize.

4.3.5 Implementation Details

In this section, we present the details of our compute and control module implementations using Intel OpenCL. We address multiple challenges here, ranging from scaling the code based on values of M and N, to ensuring stall free execution to meet latency requirements. A method for specifying tree-based computations is also presented which can be used if the compiler is unable to infer them automatically.

Scalar Product

Our Scalar Product implementation is illustrated in Algorithm 4. Typical implementations focus primarily on DSP based resources for this stage. However, we provide users with the capability of selecting arbitrary numbers of both DSP and ALM multipliers at the granularity of a Multiply-Add module. Based on board resources, user can specify the number of available DSPs while the remaining compute entities are synthesized with ALMs. All Multiply-Add module outputs are summed using ALM based adders. By using an unrolled inner loop, the compiler is able to infer and generate an adder tree automatically.

Accumulate

The Accumulate module is implemented in pure HDL, due to its low complexity, using N parallel 32 bit integer adders and registers. Register values are initialized with bias values for each new set of corresponding N scalar products (layer outputs)

Algorithm 4 Scalar Product Code Structure

```
1: DSP ← Number of DSP Multiply-Accumulate
2: I [M] ← Declare and Initialize Input Vector
3: O [N] ← Declare and Initialize Output Vector
4: W [M*N] ← Declare and Initialize Weight Matrix
5: #pragma unroll
6: for i = 1 : DSP do
7:     #pragma unroll
8:     for j = 1 : 2 : M do
9:         O[i] += MulAdd( I[j,j+1] , W[i*N+(j,j+1)] )
10: #pragma unroll
11: for i = DSP + 1 : N do
12:     #pragma unroll
13:     for j = 1 : 2 : M do
14:         O[i] += MulAdd( I[j,j+1] , W[i*N+(j,j+1)] )
```

computed.

Max Value Search

While an adder tree was inferred for Scalar Product, using the same code structure did not generate a tree for performing the maximum value search in $\approx log(N)$ stages. Instead, a sub-optimal N-stage sequential pipeline was inferred which performed one comparison per stage. An alternative code structure to explicitly define a tree was using nested loops, with the inner loop limit being based on the value of the outer loop variable. However this was not feasible since OpenCL requires loop limits to be constant in order to get good performance. Consequently, we implement a tree based comparison by utilizing the HDL compiler's ability of synthesizing away logic that does not have a source or sink, as shown in Algorithm 5. We define all loop limits as constant values i.e. $log(N)$ for the outer loop and $N/2$ for the inner loop. We then load and store data into converging locations so that by the end of the last iteration, the actual result is stored in the 0th location. By only selecting that particular value as the final output, the compiler optimizes away unnecessary logic which results in a

tree being created.

Algorithm 5 Max Value Search Code Structure

```
1: LOGN ← ⌈log2(N)⌉
2: SIZEN ← 1 << LOGN
3: HSIZEN ← SIZEN >> 1
4: Layers [SIZEN*(LOGN+1)]
5: for i = 1 : SIZEN do
6:     Layers[i] = Input[i];
7: #pragma unroll
8: for i = 1 : LOGN do
9:     Shift ← (HSIZEN >> (i-1)) +1
10:    #pragma unroll
11:    for j = 1 : HSIZEN do
12:        X ← (j < Shift) ? Layer[i*SIZEN + j] : 0
13:        Y ← (j < Shift) ? Layer[i*SIZEN + j + Shift] : 0
14:        Layer [(i+1)*SIZEN+j] ← ( X > Y ) ? X : Y
15: Output ← Layer [LOGN*SIZEN+1]
```

Leading 1

We implement the Leading 1 module by using the same approach as Max Value Search. The loop limits in this case, as shown in Algorithm 6 are independent of the values of M and N, and depend on the data sizes being used (32 bits in our implementation).

Activation & Quantize

Activation & Quantize uses the result of the Leading 1 module to quantize Buffer outputs, which are then input to the Scalar Product module. It is a two-stage data parallel computation, as shown in Algorithm 7. The first is a ReLU activation which simply uses the MSB of input data as the select line, with 0 causing a pass through of the original value while 1 selecting a zero output instead. Our second stage shifts data to the right based on the location of the leading 1.

Algorithm 6 Leading 1 Search Code Structure

```
1: MAX ← Result of Max Value Search
2: Layers [32*(5+1)]
3: for i = 0 : 31 do
4:     Layers[i+1] = ((MAX >> i)&1) ? i : 0;
5: #pragma unroll
6: for i = 1 : 5 do
7:     Shift ← (16 >> (i-1)) +1
8:     #pragma unroll
9:     for j = 1 : 16 do
10:        X ← (j < Shift) ? Layer[i*32 + j] : 0
11:        Y ← (j < Shift) ? Layer[i*32 + j + Shift] : 0
12:        Layer [(i+1)*32+j] ← ( X > Y ) ? X : Y
13: Output ← Layer [5*32+1] > 6 ? Layer [5*32+1] : 6
```

Algorithm 7 Activation & Quantize Code Structure

```
1: I [M] ← Declare and Initialize Input Vector
2: O [M] ← Declare and Initialize Output Vector
3: Loc ← Location of Leading 1
4: #pragma unroll
5: for i = 1 : M do
6:     X ← (I[i] > 0) ? I[i] : 0
7:     O[i] ← (X >> (Loc-6))&255
```

Buffer

Since the Buffer does not lie in the critical path, we generate it using HDL. As shown in Figure 4·7, the Buffer is a two bank design implemented entirely using 32 bit registers.

To reduce addressing complexity, we set the number of required registers as the first Least Common Multiple (LCM) of M and N which is greater than the largest layer size in the application. Input data is written to contiguous blocks of N registers using the input DeMUX. Once all results have been written, the transfer enable immediately triggers a copy of data from Bank 1 to Bank 2. These set of registers are accessed in contiguous blocks of size M and are used to source data to the Scalar Product. As discussed earlier, the dual banks are necessary to maintain a copy of the results of previous layer, while also storing Accumulator outputs from the current one.

Control Plane

Using the modules and their functions outlined previously, Figure 4·8 provides an overview of the algorithm for performing inference on a given test vector. The control unit is responsible for generating event triggers to coordinate the flow of data between different modules (green). These triggers are based on system states (blue) and include start/end indicators, variable updates (e.g., resets, increments, swap), read/write signals, and data source selection (e.g., the buffer module, external on-

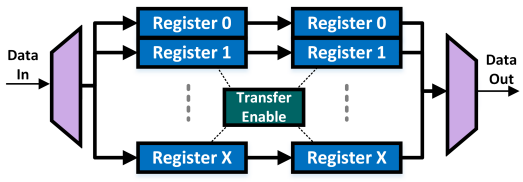


Figure 4·7: Structure of the Buffer Module. The two stage design enables us to hold values of a previous layer (input vector to Scalar Product) while also storing results of current layer computations

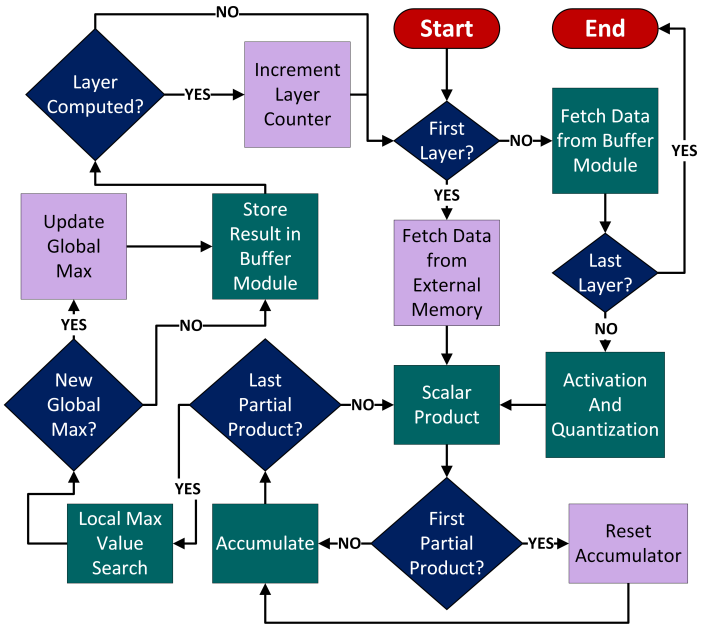


Figure 4·8: An overview of the algorithm used to determine the event triggers needed to control the flow of data in the inference processor.

chip memory).

A detailed implementation of the control unit is shown in Figure 4·9. For given values of M and N and application model dimensions, we can determine how long each layer will take, at which cycle individual triggers will be given, and the computation is performed by each module at any given time. All these can be coordinated based on a single global counter (Main Counter). By having an instruction-free implementation, the overhead of fetching and decoding instructions is avoided, and end-to-end data flow for the entire application can be made stall free.

To define ranges and trigger points for setting and resetting values of control signals and state machine counters, we use latency tags. Each tag is based on the latency of an individual module in the corresponding data path. Table 4.4 lists these tags and their values. QA, MM, AC, MX, and LO refer to latencies of the Activation & Quantization, Scalar Product, Accumulate, Max Value Search, and Leading 1 modules, respectively. Constants represent latencies of registers at the output of the

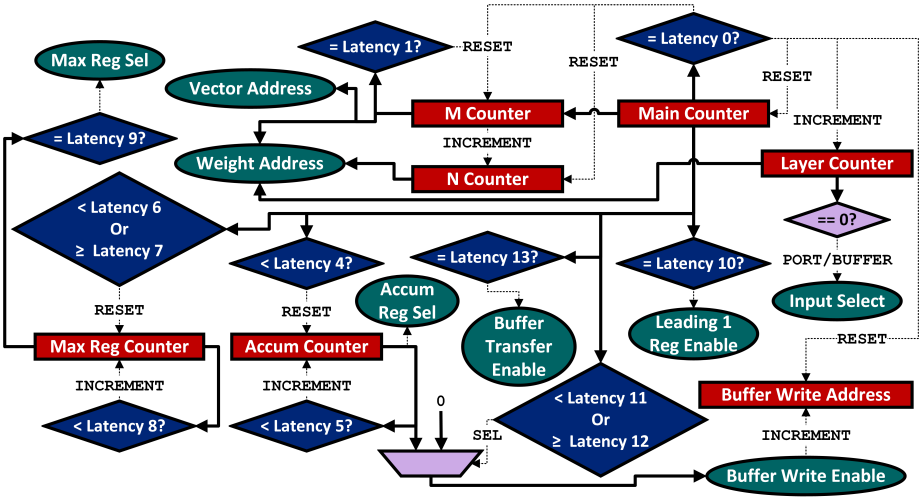


Figure 4·9: The Control Flow Graph. Execution of the entire application model can be done based on a single global counter without any feedback from modules or user supplied run-time instructions. Values of latency tags for each layer are hard coded in to the system and selected based on layer counter value. M Counter, N counter, Vector Address and Weight Address will all be removed in future work since this information can be directly evaluated from the Main Counter value.

Accumulate and Max Value Search Modules. MBLOCKS and NBLOCKS refer to the number of blocks the weight matrix of a layer can be divided into in each dimension, while BLOCKS is the product of these, that is, the number of cycles needed for all layer input data to be picked up by the system. Tags 2 and 3 are reserved for external connectivity in future work. The entire control architecture is automatically created using an RTL generator, with the user only specifying latency values of individual modules.

4.3.6 Results

In this section, we present the results from our implementation. Parameters M and N are varied in steps of powers of 2, from 8 to 256 to show the impact on latency of the different modules in our inference testbed. We also demonstrate that latency is heavily reliant on module complexity since the OpenCL compiler creates deeper pipelines for

Table 4.4: Latency Tags for Defining Trigger Ranges

Tag	Value
Latency_0	L_QA(M) + L_MM(M,N) + BLOCKS[i] + L_AC(N) + 1 + L_MX(N) + 1 + L_LO
Latency_1	MBLOCKS[i]
Latency_4	L_QA(M) + L_MM(M,N)
Latency_5	MBLOCKS[i] -1
Latency_6	L_QA(M) + L_MM(M,N) + L_AC(N) + 1 + L_MX(N)
Latency_7	L_QA(M) + L_MM(M,N) + BLOCKS[i] + L_AC(N) + 1 + L_MX(N)
Latency_8	MBLOCKS-1
Latency_9	MBLOCKS-1
Latency_10	L_QA(M) + L_MM(M,N) + BLOCKS[i] + L_AC(N) + 1 + L_MX(N) + 1 + L_LO -1
Latency_11	L_QA(M) + L_MM(M,N) + L_AC(N) + 1
Latency_12	L_QA(M) + L_MM(M,N) + BLOCKS[i] + L_AC(N) + 1
Latency_13	L_QA(M) + L_MM(M,N) + BLOCKS[i] + L_AC(N) + 1

large modules in order to meet fan out and combination path constraints. We also use the ECP-Candle, Poker and MNIST benchmarks to compare the performance of our application specific design with reference GPU code compiled using TensorFlow. Microsoft Brainwave was not available for public use at the time of writing and hence could not be instrumented.

Hardware Specifications

We have tested our designs on the Intel Arria-10AX115H3F34E2SGE3 which has 427,200 ALMs, 1518 DSP blocks and 53Mb of on chip memory. The GPU used is a Tesla P100 which has 3594 CUDA cores and a 12GB HBM2 memory with a peak bandwidth of 549 GB/s. GPU reference designs are implemented in floating point using TensorFlow (Abadi et al., 2016) r1.4, python 3.6.2, cuDNN 6.0 and CUDA 8.0.

Measured Latency

Figure 4·10 shows the latency of the Scalar Product module. Data points for (M $\leftarrow$ 8:256 , N $\leftarrow$ 8), (M $\leftarrow$ 8 , N $\leftarrow$ 8:256), (M $\leftarrow$ 16, N $\leftarrow$ 16), (M $\leftarrow$ 32, N $\leftarrow$ 32) and

(M $\leftarrow$ 64, N $\leftarrow$ 64) are measured values while the remaining points are estimated based on observed trends. As illustrated by the graph, larger M values correspond to higher latency, while the latency due to N is mostly constant (due to simple design replication).

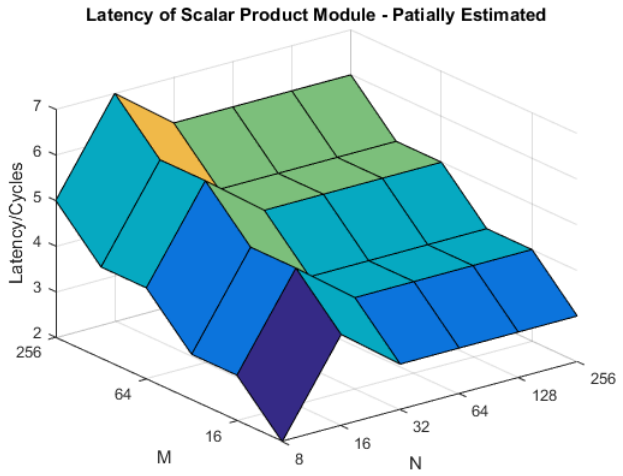


Figure 4·10: Latency comparison of varying M and N. Latency increases with larger M (more tree stages) but is invariant of N (tree replication)

Figure 4·11 illustrates the latency of modules in the Persistent Critical Path (except for Scalar Product accumulation). As was demonstrated with our system model previously, most modules have latency offsets based on pipeline depth and thus have nearly invariant latencies with respect to their associated parameter. With regard to Max Value Search, however, we get very large latencies despite it being a tree-based implementation. This is because of the resource overhead of a signed comparator as compared with a simple adder (≈2x more ALMs per comparator based on synthesis results). Figure 4·12 shows the total latency of our system for a single iteration of all modules. We observe that having larger values of M, instead of N, reduces latency by 20% on average.

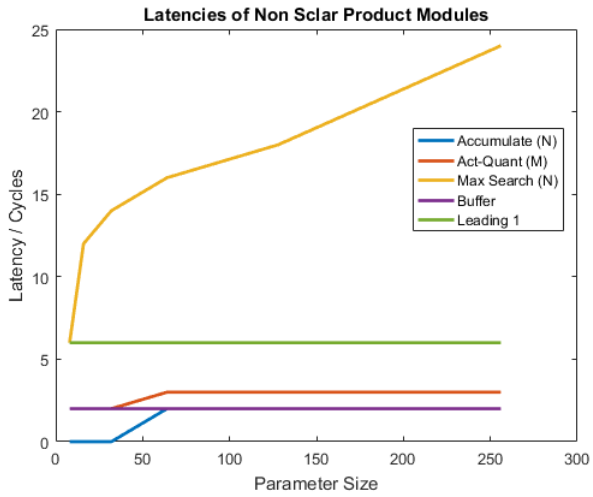


Figure 4·11: Latency of critical path modules based on their constraining parameter

Benchmarks

To demonstrate the impact of our approach, we evaluate performance using the Poker (UCI, 2018b), ECP-Candle P3B1-TL (TL) (ECP,) and MNIST (UCI, 2018a) benchmarks. Table 4.5 lists the model parameters of each benchmark. The first and last dimensions represent input and output vector sizes respectively. Table 4.6 gives the post-fit resource usage of the processor. We compile the design with M=256 and N=8. These values minimize overall inference latency based on the dimensions of benchmark models and latency results from Figure 4·12. From the resource usage, we see that the design occupies less than half the chip. Therefore, either a larger value of M can be used to further reduce latency, or a second (independent) inference processor can be included.

Table 4.5: Benchmark Dimensions

Model	Dimensions	Test Cases
MNIST	**784** x 256 x 256 x **10**	10000
Poker	**10** x 512 x 512 x **10**	25010
TL	**400** x 1200 x 1200 x 1200 x **2**	86

Figure 4·13 gives the measured speedups compared to the GPU. From the results,

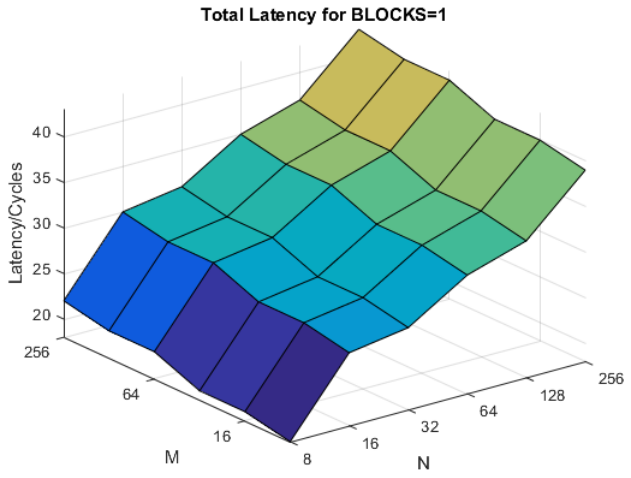


Figure 4·12: Total latency of our system for a single iteration. Having a larger value of M gives significantly lower latency than larger values of N

Table 4.6: FPGA Implementation Details

M	N	ALM	DSP	Frequency
256	8	57008/427200(13%)	512/1518(34%)	295MHz

we show that our FPGA design has an average of 1.47x better performance than the high end GPU. GPU results are heavily reliant on batch processing where large number of test cases, such as in Poker, reduce the average execution time for individual vectors. However, this throughput comes at the cost of higher individual latencies, which is a negative result since inference has hard latency bounds. Our design, on the other hand, processes individual vectors in sequence and uses optimal design parameters to achieve both high throughput and low latency.

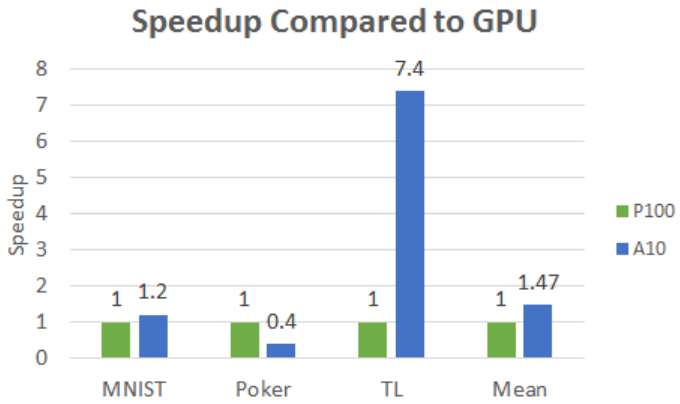


Figure 4·13: Performance of Arria-10 (FPGA) as compared to P-100 (GPU)

4.4 Conclusion

In this chapter, we extended our approach of creating OpenCL-HDL based test systems by utilizing the isolated application logic to generate custom IP. These IP blocks can be used to built performance critical and complex components, which are then inserted into existing HDL templates. Two case studies were provided in this regards: i) 3D FFT and ii) Multi Layer Perceptron Inference. Results showed the OpenCL-HDL can outperform both vendor IP and state-of-the-art ASICs due to rapid design space exploration and application specific tuning of hardware.

Chapter 5

Empirically Guided Optimization Framework for FPGA OpenCL

5.1 Motivation

Despite the advantages of OpenCL, it has still not managed to effectively bridge the performance-programmability gap for FPGAs. OpenCL kernels often end up having orders of magnitude worse performance than functionally equivalent HDL designs. Significant expertise in how the C-to-Hardware translation works is typically required for good performance.

Automatic compilation to a complex architecture is well-known to be an extremely difficult problem; even after decades of research it has been only partially solved. In HPC, programmers achieve high performance by augmenting the coding process, first, by integrating optimized libraries, and then, when these are not sufficient, by optimizing the code themselves (Chellappa et al., 2008). Since this process is challenging even for experienced programmers, a vast area of research has grown up around automating it through autotuning (Moura et al., 2005).

We propose that an analogous approach be applied to coding OpenCL for FPGAs. We broadly categorize types of code transformations in this domain into four sets: i) Intel Best Practices (IBPs), ii) Anti-IBPs, iii) Universal Code Transformations (UCTs), and iv) FPGA-Specific Transformations (FSTs). **IBPs** refer to design strategies given in the Intel Best Practices guide (Intel, 2018c). These provide insights

into how to effectively express hardware using OpenCL semantics. We separate these from UCTs and FSTs because IBPs are well-known to the FPGA OpenCL community and there have been several studies characterizing their behavior. **Anti-IBPs** are practices that are part of IBPs, but which (we have found) should actually be avoided. **UCTs** consist of general approaches to optimizing programs that, to a large degree, are independent of the compute platform. Examples of UCTs include use of 1D arrays, records of arrays, predication, loop merging, scalar replacement, and precomputing constants. While described, e.g., in (Chellappa et al., 2008), they are largely missing from IBP documentation. **FSTs** consist of a number of FPGA-specific optimizations that typically augment IBPs. They are based on (a) obtaining a particular FPGA-specific mapping not found as an IBP, and (b) facts stated in IBPs, but which must be leveraged and converted into transformations.

We propose a systematic and empirically guided series of code transformations for creating High Performance FPGA OpenCL kernels, using a combination of IBPs, Anti-IBPs, UCTs, and FSTs. These are aimed at giving the OpenCL compiler sufficient freedom to infer and exploit all possible forms and degrees of parallelism, along with more aggressive steps such as restructuring pipeline stages to minimize latency and resource overhead. It is important to note here that, while some FSTs may be commonly known, since they can be based on UCTs and IBPs, our work is novel in their application to FPGAs. We also characterize and measure the impact of all optimizations. These results not only enable programmers to follow a script when optimizing their own kernels, but also open the way for the development of autotuners to perform optimizations automatically.

We also demonstrate that, by applying these proposed code modifications to a number of parallel computing dwarfs, OpenCL FPGA code can achieve performance within 12% of hand tuned HDL. This is an average of 5x better performance than

existing FPGA OpenCL designs. We also apply these transformations to a number of network packet processing workloads . Results show that OpenCL generated hardware can operate at >50Gb/s line rates. Moreover, these designs have low resource overhead which allows them to be replicated multiple times and achieve even higher aggregate throughput.

5.2 Previous Work

One of the most important studies in this area, by Zohouri, *et al.* (Zohouri et al., 2016), implements both GPU based and FPGA-specific codes. The latter, referred to as loop-pipelined kernels, employ Single Work Item kernels and IBPs such as sliding windows and shift registers. Our work advances that of (Zohouri et al., 2016) in a number of ways. We have implemented many more optimizations; authors in (Zohouri et al., 2016) only use IBPs, while we characterize Anti-IBPs, UCTs and FSTs as well. Performance values for best case implementations in (Zohouri et al., 2016) show an average of 35× improvement over un-optimized baselines while our work shows an average speedup of 288× for the final optimized version (with respect to the baseline). Moreover, the average speedup reported by Zohouri, *et al.* over GPUs is approximately 0.25×, while we achieve approximately 0.5×. Finally, unlike our reference GPU implementations, the GPU used in (Zohouri et al., 2016) does not have High Bandwidth Memory.

5.3 Transformations

In this section, we present our design pattern agnostic OpenCL code transformations that help the compiler infer and generate optimal architectures. There are seven code versions, which are incrementally developed. Each version contains one or more applied transformations. Version 1 is a cache optimized CPU code for the application

Table 5.1: Summary of code versions and transformations applied therein

Ver.	Transformations	Type
0	(**GPU code** for porting to FPGA OpenCL)	—
1	**Single thread code with cache optimization**	IBP,FST
2	Implement **task parallel computations in separate kernels** and connect them using channels	IBP
	Fully unroll all loops w/ *#pragma unroll*	IBP,UCT
	Minimize variable declaration outside compute loops – use temps where possible	IBP,UCT
	Use **constants** for problem sizes and data values – instead of relying on off-chip memory access	IBP,FST,UCT
	Coalesce memory operations	IBP,UCT
3	Implement the entire computation within a **single kernel** and avoid using channels	Anit-IBP
4	**Reduce array sizes** to infer pipeline registers as registers, instead of BRAMs	FST
5	Perform **computations in detail**, using temporary variables to store intermediate results	FST,UCT
6	Use **predication** instead of conditional branch statements when defining forks in the data path	FST,UCT

of interest. Version 2 is obtained by applying the IBPs to the baseline code. Versions 3-6 involve applying a number of additional transformations that, not only maximize opportunities for parallelism, but also reduce the complexity (and hence resource usage and latency) of the generated control and data planes. Table 5.1 summarizes the optimizations and their type (IBP, Anti-IBP, FST, and/or UCT). We illustrate each set of transformations through a running example, the Needleman-Wunsch (NW) benchmark.

5.3.1 Version 0: Sub-Optimal Baseline Code

A popular starting point (e.g., in (Krommydas et al., 2016)) is kernels based on Multiple Work Items (MWI) such as is appropriate for GPUs. Advantages of starting here include ease of exploiting data parallelism through SIMD, and Compute Unit Replication (CUR), which is exclusive to MWI structures.

Algorithm 8 shows a V0-type kernel (based on (Che et al., 2009)). The core operation is to populate a matrix using known values of the first row and the first

Algorithm 8 Needleman Wunsch-V0

```
1: int tx = get_local_id(0)
2: __local int* temp
3: __local int* ref
4: Initialize temp from global memory
5: barrier(CLK_LOCAL_MEM_FENCE);
6: Initialize ref from global memory
7: barrier(CLK_LOCAL_MEM_FENCE);
8: for i = 1 : SIZE do
9:     if tx≤i then
10:        compute t_idx_x and t_idx_y based on tx and i
11:        temp[t_idx_y][t_idx_x] =
12:              max( temp[t_idx_y-1][t_idx_x-1] +
13:              ref[t_idx_y-1][t_idx_x-1],
14:              temp[t_idx_y][t_idx_x-1] - penalty,
15:              temp[t_idx_y-1][t_idx_x] - penalty);
16:    barrier(CLK_LOCAL_MEM_FENCE);
17: barrier(CLK_LOCAL_MEM_FENCE);
18: for i = SIZE − 2 : 0 do
19:    Perform computations similar to above
20: barrier(CLK_LOCAL_MEM_FENCE);
21: Store temp to global memory
```

column. Each unknown entry is computed based on the values of its left, up, and up-left locations. This is achieved using loops which iterate in-order over all matrix entries. The max function is implemented using 'if-else' statements. In Algorithm 8, SIZE represents the dimension of blocks of matrix entries being processed.

5.3.2 Version 1: Preferred Baseline Code (used for reference)

Algorithm 9 Needleman Wunsch-V1

```
1: for i = 1 : Vector_B_Size do
2:     for j = 1 : Vector_A_Size do
3:         Out[i,j] = max( Out[i-1,j] - penalty,
4:                 Out[i-1,j-1] + ref[i,j] , Out[i,j-1] - penalty)
```

A less intuitive, but preferred, alternative is to use (as a baseline) single threaded CPU code. In particular, initial designs should be implemented as Single Work Item (SWI) kernels. While use of MWI kernels best matches the original purpose of OpenCL, enabling various expressions of parallelism, there are number of disadvantages for FPGAs.

Scheduling work-groups/items: Similar to GPUs, a scheduler is required to balance workloads and ensure all pipelines are kept filled. Using CUR increases scheduling effort since work-groups must be scheduled across the different compute units. Use of a simple scheduler can result in under-utilized pipelines. But more complex schedulers can require more resources and increase latency.

Under-utilization: CUR helps fill the chip in order to maximize use of available resource. However, because entire workgroups are assigned to each compute unit, the latter must divide the former perfectly; often, areas of the chip are mostly idle.

Static SIMD size: MWI kernels require a global SIMD size to be defined. This is sub-optimal for asymmetric pipelines where opportunities for data parallelism can vary frequently.

Global synchronization overhead: Compute units cannot communicate with each other directly. As with GPUs, data transfers between workgroups across compute units must go through off-chip memory. In addition to large synchronization overhead, the advantage of connectivity within FPGAs is lost.

Wrapper Overhead: Extra OpenCL wrapper logic is required to individually interface compute units with the host and external memory. Arbitration is often required, which incurs overheads similar to that of the scheduler.

Symmetry constraints: All work-items and work-groups are assigned equal amounts of private and local memory, respectively. This symmetry is not favorable when the workload varies as a factor of the work-item/work-group number.

In contrast, SWI kernels do not require a scheduler; pipelines are customized for a given application; parallelism is inferred and exploited in SWI kernels by analyzing the computational flow and dependencies; the compiler can employ an arbitrary number of registers and dimensions of BRAMs; communication is global because any set of pipeline registers can transfer data to each other; and wrapper overhead is substantially reduced.

The CPU-like baseline code should also be optimized for cache performance; this helps the compiler infer connectivity between parallel pipelines (i.e., whether data can potentially be directly transferred between pipelines instead of being stored in memory), improves bandwidth for on-chip data access, and efficiently uses the internal cache of Load Store Units which are responsible for off-chip memory transactions.

Algorithm 9 shows the preferred baseline kernel. The first row and column of the matrix are Vector_A and Vector_B respectively.

5.3.3 Version 2: IBPs

Given the preferred baseline code, we then apply the following commonly used IBPs.

Algorithm 10 Needleman Wunsch-V2

```
1: N ← Size of systolic array
2: PE_k → Kernel Begin
3: int up, left, up_left, cached_up, cached_up_left
4: for i = k : N : Vector_A_Size do
5:     Initialize cached_up & cached_up_left
6:     for j = 1 : 1 : Vector_B_Size do
7:         left ← read_channel (PE_{k-1})
8:         up = cached_up
9:         up_left = cached_up_left
10:        cached_up = max(up - penalty,
11:                left - penalty , up_left + ref [j,i])
12:        cached_up_left = left
13:        Out[j,i] = cached_up
14:        write_channel (PE_{k-1})← cached_up
```

Multiple Task Parallel Kernels

Task parallelism is conventionally leveraged by implementing independent tasks as individual kernels. FPGA OpenCL implements direct connectivity between these kernels using channels (FIFOs) of variable data widths and depths, with support for both blocking and non-blocking operations. Channels are critical to performance in this approach, since all kernels operate concurrently and potentially large amounts of data are transferred between them. Availability of data transfers directly between pipelines located in separate kernels avoids off-chip memory accesses.

Fully unroll all loops

All loops must be fully unrolled whenever possible. Partial unrolls should be *avoided* if resources are limited since that can add significant complexity and overhead to pipelines. Rather, the problem can be folded by increasing the outer loop limit and doing less work per iteration. This allows the compiler to exploit all forms and degrees of parallelism at very fine granularity.

Minimizing State Register Usage

State registers are a special type of pipeline register whose value persists across algorithm iterations. They are inferred using variables declared outside the loops where they are used. The compiler generates feedback hardware for them to explicitly pass values to and from compute pipelines every iteration. Moreover, the state register hardware can also interface off-chip memory for loading initialization values. The initialization loop should be unrolled so that the state registers can be loaded in parallel.

We observe that the compiler is bound to ensure state register availability across subsequent iterations. For computations involving complex updates to the variable (as opposed to simple operations like increment/decrement), the compiler can generate enough pipeline stages to prevent data from being forwarded stall-free to the next iteration. As a result, the pipeline is either stalled, or the operating frequency is lowered to accommodate a larger combinational path. There is also a resource overhead of implementing the feedback logic.

As a result, we attempt to minimize stage register use by moving variable declarations to within the outer loop whenever possible. Since the compiler is then no longer bound to ensure availability of these variables across iterations, it can perform more aggressive optimizations such as pipeline re-ordering.

Constant Arrays

Determining whether variables of known values should be initialized as constant arrays is a simple, yet important design decision since it impacts the implementation beyond reducing memory accesses. The compiler analyzes how the values are used in the context of corresponding computations and generates hardware accordingly. If the constant array has static accesses, i.e. persistently accessing the same value at a

particular pipeline stage, the compiler can attempt to minimize the resources needed for that computation by pre-computing results if possible. For example, if a number is persistently multiplied with 1, the compiler will replace the DSP block with a delay module that simply passes the input to the output whilst ensuring data-paths continue to be synchronized. On the other hand, random accesses of the constant array by multiple pipelines can result in memory replication with a unique copy of the constant array generated for each pipeline. If the array was large enough, this could saturate board resources and the design will not compile. Therefore, while constant arrays should be used whenever possible to improve performance, their size should be minimized to ensure the design can successfully compile.

Coalescing

As with GPUs, coalescing is critical to effectively utilizing memory bandwidth. Even a single un-coalesced read per iteration can result in stalls which leads to poor performance. It is preferable to fetch larger blocks of data in one access and initialize state registers, which in turn supply data to the pipelines.

Algorithm 10 shows the Needleman Wunsch kernel structure after we apply IBPs. Parallelism is exploited using a systolic array, with each Processing Element (PE) implemented in a separate kernel. Channels are used to connect PEs in a specified sequence. For each inner loop iteration, PEs compute consecutive columns within the same row. This ensures spatial locality for memory transactions. The drawback is data dependencies between kernels, which cannot be reliably broken down by the compiler since it optimizes each kernel as an individual entity. Thus, the overhead of synchronizing data paths can result in performance degradation.

5.3.4 Version 3: Single Kernel Design

In Version 3 we merge the IBP optimized task parallel kernels and declare all compute loops within the same kernel. Expressing the computation as a single kernel as opposed to multiple task parallel kernels is an Anti-IBP, since the latter approach is given in the Intel best practises guide. There are a number of advantages in using a single kernel.

Algorithm 11 Needleman Wunsch-V3

```
 1: N ← Size of systolic array
 2: int value[N+1], left[Vector_B_Size]
 3: left ← Vector_B
 4: for i = 1 : 1 : Vector_A_Size/N do
 5:     base = f(i)
 6:     value ← Vector_A[base:base+N+1]
 7:     for j = 1 : 1 : Vector_B_Size do
 8:         int up_left[N+1]
 9:         for k = 2 : 1 : N + 1 do
10:             up_left[k] = value[k-1]
11:         value[1] = left[j]
12:         #pragma unroll
13:         for k = 2 : 1 : N + 1 do
14:             value[k] = max(value[k-1] - penalty,
15:               up_left[k] + ref[j, base+k] , value[k] - penalty)
16:         left[j] = value[N+1]
17:         Out ← value[2:N+1]
```

There is inherent global synchronization since all computations are tied to the same outer loop variable. Moreover, there is direct connectivity between pipelines which lowers communication overhead and enables pipelines stages, previously isolated due to channels, to be merged. Within a kernel, these loops should be placed in the same outer loop, which represents the algorithmic flow. Each loop iteration can correspond to a complete application stage, or have multiple variables derived from the loop iterator to emulate nested loops. Outer loops are used by the compiler to determine data-path latencies and synchronize them. Delay modules using FIFOs

are added in case of a latency mismatch. Since the compiler only needs the correct variable values at the end of an iteration, it is not bound to follow explicit C code steps. Rather, pipelines can be reordered which can result in merged computation with reduced resource usage, as well as overlap of delay modules across pipelines to reduce/eliminate them. This approach also reduces the control logic for tracking application progress, which could be as simple as a counter-comparator circuit. Nested loops are typically avoided since there are a small number of stall cycles after each outer loop iteration.

Algorithm 11 shows the kernel structure for implementing the systolic array as a single kernel. The compiler can now optimize the entire computation, as opposed to individual PEs. Synchronization overhead is also reduced since almost all computation is tied to a single loop variable (j). Nested loops are used since, in this particular case, the cost of initiation intervals is outweighed by the reduction in resource usage. This is because the compiler was unable to infer data access patterns when loops were coalesced.

5.3.5 Version 4: Reduced Array Sizes

OpenCL limits the size of a register array in SWI kernels. If this limit is exceeded, the arrays are converted to BRAM based storage. While this is acceptable for data memory/cache, inferring pipeline registers as BRAMs can have significant drawbacks on the design. Since BRAMs cannot source and sink data with the same throughput as registers, barrel shifters and memory replication is required which drastically increases resource usage. Moreover, the compiler is also unable to launch stall-free iterations of compute loops due to memory dependencies. Our solution is to break large arrays corresponding to intermediate variables into smaller ones. Ideally, arrays should be avoided altogether where ever possible. Instead, scripts can be used to create and reference individual variables.

Algorithm 12 Needleman Wunsch-V4

```
1: N ← Size of systolic array
2: int value_1, value_2 ... value_N_plus_1
3: int left [Vector_B_Size]
4: left ← Vector_B
5: for i = 1 : 1 : Vector_A_Size/N do
6:     base = f(i)
7:     value_1 ← Vector_A[base]
8:             ↓
9:     value_N_plus_1 ← Vector_A[base+N+1]
10:    for j = 1 : 1 : Vector_B_Size do
11:        int up_left_2 ... up_left_N_plus_1
12:        up_left_2 = value_1
13:                ↓
14:        up_left_N_plus_1 = value_N
15:        value_1 = left [j]
16:        value_2 = max(value_1 - penalty,
17:            up_left_2 + ref[j,base+2], value_2 - penalty)
18:                ↓
19:        value_N_plus_1 = max(value_N - penalty,
20:                up_left_N_plus_1 + ref[j,base+N+1],
21:                value_N_plus_1 - penalty)
22:        left[j] = value_N_plus_1
23:        Out ← value_2 ... value_N_plus_1
```

Algorithm 12 shows the kernel structure for inferring pipeline registers as registers. All arrays are expressed as individual variables, with the exception of local storage of Vector_B in 'left,' which has low throughput requirements.

5.3.6 Version 5: Detailed Computations

The OpenCL compiler does not reliably break down large computations being assigned to a single variable into intermediate stages. As a result, dependency across iterations can be considered as the worst case, i.e., the overall result of the computation is required for the next iteration's first evaluation in the computation chain. The compiler thus stalls the pipeline for the required number of cycles to address this. Our solution is to do computations in as much detail as possible by storing results in intermediate variables. This helps the compiler infer potential pipeline stages with forwarding hardware. Memory dependencies are removed and the critical path is decreased. If the pipeline is already optimal, these variables will be synthesized away and resource is not wasted.

Algorithm 13 shows the kernel structure after performing computations in detail with a number of intermediate variables added. The 'max' function is also explicitly implemented.

5.3.7 Version 6: Predication

We optimize conditional operations by explicitly specifying architecture states which ensure the validity of the computation. Since hardware is persistent and will always exist once synthesized, we avoid using conditional branch statements. Instead, variable values are conditionally assigned such that the output of invalid operations is not committed and hence does not impact the overall result. Examples of this include zeroing out variables and pointer arithmetic. Algorithm 14 shows the 'if-else' operations replaced with conditional assignments.

Algorithm 13 Needleman Wunsch-V5

```
1: N ← Size of systolic array
2: int value_1, value_2 ... value_N_plus_1
3: int left [Vector_B_Size]
4: left ← Vector_B
5: for i = 1 : 1 : Vector_A_Size/N do
6:     base = f(i)
7:     value_1 ← Vector_A[base]
8:             ↓
9:     value_N_plus_1 ← Vector_A[base+N+1]
10:    for j = 1 : 1 : Vector_B_Size do
11:        int a_2 = value_1 + ref[j,base+2];
12:        value_1 = left[j]
13:
14:        int b_2 = value_1 - penalty
15:        int a_3 = value_2 + ref[j,base+3];
16:        int c_2 = value_2 - penalty
17:
18:        if ((a_2 ≥ b_2) && (a_2 ≥ c_2))
19:                value_2 = a_2
20:        else if ((b_2 > a_2) && (b_2 ≥ c_2))
21:                value_2 = b_2
22:        else
23:                value_2 = c_2
24:
25:        int b_3 = value_2 - penalty
26:        int a_4 = value_3 + ref[j,base+4];
27:        int c_3 = value_3 - penalty
                   ⋮
28:
29:        left[j] = value_N_plus_1
30:        Out ← value_2 ... value_N_plus_1
```

Algorithm 14 Needleman Wunsch-V6

```
1:        ⋮
2: int a_2 = value_1 + ref[j,base+2];
3: value_1 = left[j]
4:
5: int b_2 = value_1 - penalty
6: int a_3 = value_2 + ref[j,base+3];
7: int c_2 = value_2 - penalty
8:
9: int d_2 = (a_2 > b_2) ? a_2 : b_2
10: value_2 = (c_2 > d_2) ? c_2 : d_2
11:       ⋮
```

5.3.8 Case Study: Parallel Computing Dwarfs

In this case study, we apply our transformations to a number of parallel computing dwarfs. This includes the Fast Fourier Transform (FFT), Range Limited Molecular Dynamics (Range Limited), Particle Mesh Ewald (PME), Dense Matrix Matrix Multiplication (MMM), Sparse Matrix Dense Vector Multiplication (SpMV), and Cyclic Redundancy Check (CRC) benchmarks. We not only evaluate the impact of individual transformation to each benchmark, but also demonstrate the importance of selecting the correct baseline model. We also compare the performance of generated pipelines against other platforms/approaches such as CPU, GPU, Verilog, and exiting FPGA OpenCL implementations. To ensure fairness, values used for comparisons are all either obtained from literature, or from implementations of available source codes/libraries. Figure 5·1 provides a summary of these benchmarks, their associated dwarfs, tested problem sizes, and applicable code versions. Blank table entries indicate that the version was not created since the corresponding transformations were not possible in the context of the application. For example, predication does not apply to FFT since there is a single, fixed data path.

We implement the designs using an Altera Arria 10AX115H3F34I2SG FPGA and

Benchmarks	Dwarf	Problem Size
NW	Dynamic Programming	16K x16K Integer Table
FFT	Spectral Methods	64 point Radix-2 1D FFT, 8192 Vectors
Range Limited	N-Body	180 particles per cell, 15% pass rate
PME	Structured Grids	1,000,000 Particles, 32^3 grid, 3D Tri-Cubic Interpolation
MMM	Dense Linear Algebra	1K x 1K Matrix, Single Precision
SpMV	Sparse Linear Algebra	1K x 1K Matrix, Single Precision, 5%-Sparsity, NZ=51122
CRC	Combinational Logic	100MB CRC32

Benchmarks	V-1	V-2	V-3	V-4	V-5	V-6
NW	✓	✓	✓	✓	✓	✓
FFT	✓	✓	✓	✓	✓	
Range Limited	✓	✓	✓	✓	✓	✓
PME	✓	✓	✓			✓
MMM	✓	✓	✓		✓	
SpMV	✓	✓	✓		✓	✓
CRC	✓	✓	✓		✓	✓

Figure 5·1: Benchmark Summary – Not all optimizations are applicable to all codes

Altera OpenCL SDK 16.0. The FPGA has 427,200 ALMs, 1506K Logic Elements, 1518 DSP blocks, and 53Mb of on-chip storage. For GPU implementations, we use the Tesla P100 PCIe 12GB GPU with CUDA 8.0. It has 3584 CUDA cores and peak bandwidth of 549 GB/s. CPU codes are implemented on a 14 core 2.4 GHz Intel Xeon E5-2680v4 with Intel C++ Compiler v16.0.1.

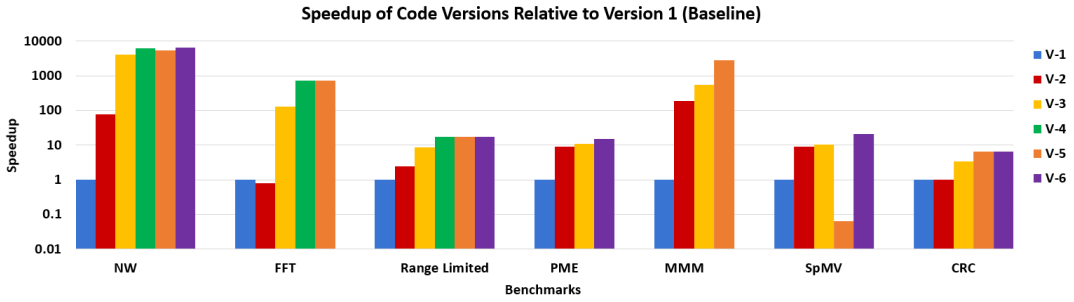


Figure 5·2: Impact of systematic application of proposed optimizations to a cache-optimized CPU baseline code. In almost all cases, every subsequent code version shows increasing performance, with up to orders of magnitude better performance possible for fully optimized kernels over ones with only IBPs (V-2).

Figure 5·2 shows the results of individual transformations. In almost all cases, the

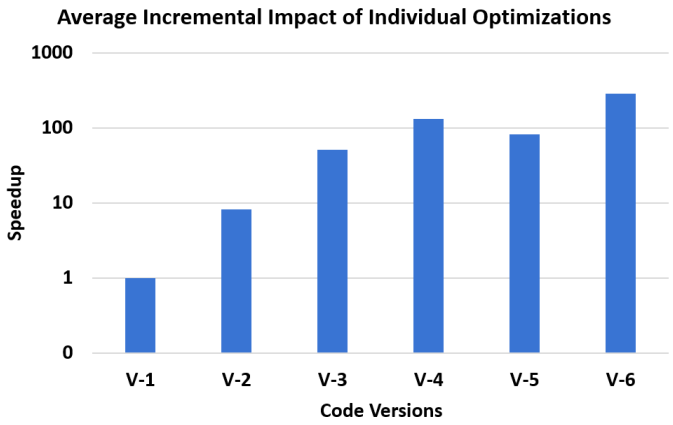


Figure 5·3: Performance for different code versions, obtained by averaging the speedup of all applicable benchmarks.

same trend can be observed where IBPs (V-2) only result in a fraction of the speedup possible. The shortcomings of IBPs are especially highlighted in CRC and FFT. For the former, the V-2 code for CRC has the same performance as the baseline. Results for the latter, FFT, are worse still since implementing multi-kernel designs with channels results in a lower performance than even the baseline. On the other hand, by applying the additional transformations on top of V-2, the achieved performance is improved by orders of magnitude. The average impact of individual transformations is shown in Figure 5·3. Generally, each successive set of transformations applied results in increasing performance. The exception is V-5. This is due to higher execution times of V-5 for NW and SpMV. In both cases, performing computations in as much detail as possible results in the use of conditional statements that outweigh benefits of the transformation. Once these statements are removed in V-6, the speedup increases.

We also highlight the importance of selecting the correct baseline model for kernel development by implementing MMM kernels with three different approaches (Figure 5·4). MMM-JIK is the naive approach: the outer loops, i and j, select the row and column of two matrices A and B, respectively. The inner-most loop, k, iterates over

all elements in the selected row and column. j is selected as the outermost loop so that the column vector, which has a poor access pattern, is only read once from global memory, stored in local variables and reused. MMM-KIJ swaps the loops, moving the k loop to the outermost location in the hierarchy. It is unable to outperform MMM-JIK, despite a better access pattern, because of the writes in the inner loop. Finally, MMM-Block is a blocked version that targets high data reuse and minimal memory access. It is thus able to achieve the lowest execution time. In the case of MMM-Block, we also demonstrate that despite having the worst performance in V-1, the final version V-6 has the lowest execution time since it benefits more from the applied transformations. On the other hand, while improvements are seen for the other two versions as well, the benefits are relatively small.

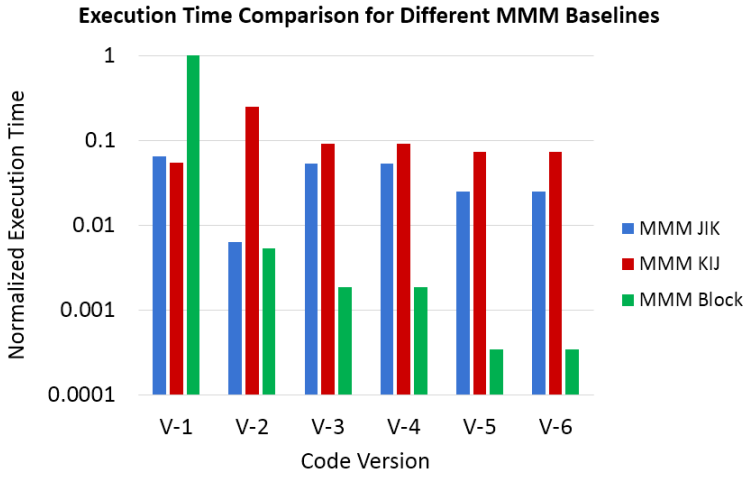


Figure 5·4: Transformations performed for MMM versions with different initial code structures. The observed trend is that better memory access patterns of a given baseline results in a larger impact of each transformation and lower overall execution time.

Finally, to demonstrate the overall effectiveness of the approach, we compare the performance of the optimized kernels against existing CPU, GPU, Verilog, and FPGA-OpenCL implementations. Table 5.2 lists the references for these designs; they are either obtained from the literature or implemented using available source

Table 5.2: References for Existing Implementations

Benchmarks	CPU	GPU	Verilog	OpenCL
NW	Rodinia* (Che et al., 2009)	Rodinia* (Che et al., 2009)	Benkrid (Benkrid et al., 2009)	Zohouri (Zohouri et al., 2016)
FFT	MKL* (Intel, 2018a)	cuFFT* (NVIDIA, 2018)	Sanaullah (Sanaullah and Herbordt, 2018a)	Intel* (Intel, 2018b)
Range Limited	-	-	Yang (Yang et al., 2017b)	Yang (Yang et al., 2017b)
PME	Ferit (Büyükkeçeci et al., 2013)	Ferit (Büyükkeçeci et al., 2013)	Sanaullah (Sanaullah et al., 2016a)	-
MMM	MKL* (Intel, 2018a)	cuBLAS* (NVIDIA, 2008)	Shen (Shen et al., 2018)	Spector* (Gautier et al., 2016)
SpMV	MKL* (Intel, 2018a)	cuSparse* (NVIDIA, 2014)	Zhou (Zhuo and Prasanna, 2005)	OpenDwarfs* (Krommydas et al., 2016)
CRC	Brumme* (Brumme, 2018)	-	Anand (Anand et al., 2016)	OpenDwarfs* (Krommydas et al., 2016)

code/libraries. The latter is illustrated using an asterisk (*). Verilog FFT measurement from (Sanaullah and Herbordt, 2018a) has been extended to include off-chip access overhead. Figure 5·5 shows the average speedup achieved over the CPU code while Figure 5·6 shows the normalized execution times for all implementations.

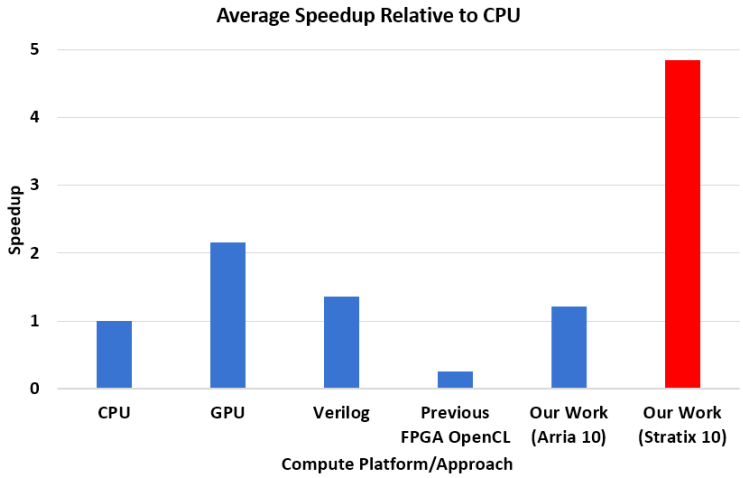


Figure 5·5: Average speedup wrt CPU across all applicable benchmarks.

From the results, we observe that our work outperforms multi-core CPU implementations by approximately 1.2× due to the performance of codes written using Intel MKL. We have also achieved an average of approximately 5× lower execution time than existing FPGA OpenCL work. The exception is FFT, where we have a 3.7× higher execution time than the reference FPGA OpenCL implementation. Similar to MKL, this reference design was developed by Intel engineers, who are familiar with

the low level details of the C-to-HDL translation, and the optimizations performed cannot be applied in a general way to all applications.

The GPU speedup of 2.4× relative to our work is due to the use of a high-end GPU, Tesla P100, and a medium-end FPGA, Arria-10. We therefore also provide an estimate of a high end FPGA performance, Stratix-10, using a conservative factor of 4× that accounts for an increase in resource only. Results show that the optimized kernels on Stratix-10 are expected to outperform GPU designs by 65%, on average. Comparison with existing Verilog implementations shows that the kernels are, on average, within 12% of hand-tuned HDL. This demonstrates that the transformations are successfully able to bridge the performance-programmability gap for FPGAs and deliver HDL-like performance using OpenCL.

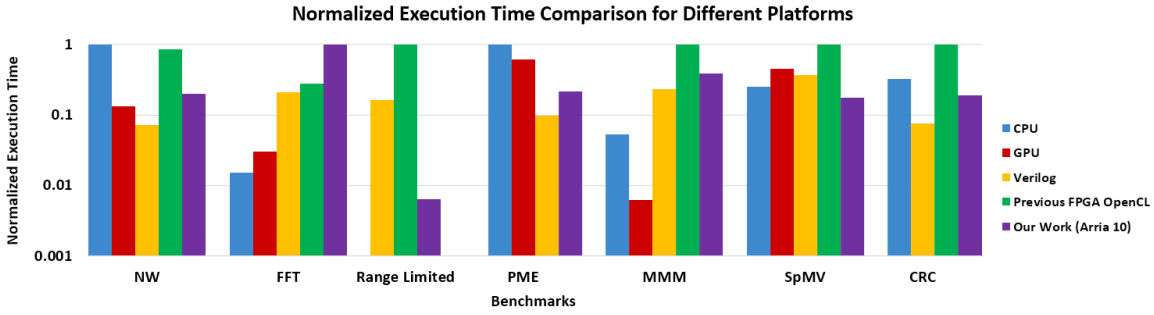


Figure 5·6: Performance of Our Work as compared with existing CPU, GPU, Verilog and FPGA OpenCL implementations. Our work outperforms CPU and OpenCL for most of the benchmarks. Moreover, we also achieve speedups over GPU (SpMV, PME) and Verilog (SpMV, Range Limited).

5.3.9 Case Study: Network Packet Processing

In the previous case study, we explored a number of ad hoc applications for evaluating the effectiveness of our transformations. Here, we focus on an actual core use case for FPGAs in Data Centers i.e. Network Packet Processing (NPP). NPP involves operating on data as it moves through the network e.g. IPSec, load balanc-

ing, collective, Network Function Virtualization, and Software Defined Networking. These applications are well suited for FPGAs since spatial computing enables data processing at line rates, whereas 1000s of instructions are needed in software. The specific workloads targeted in this case study include matching (Cuckoo hash table, Regular expression match), encryption (AES-256), hashing (Murmur3, SHA-1) and probabilistic data structures (HyperLogLog, Bloom filters).

Application of Transformations to NPP Workloads

Figure 5·7 illustrates the impact of applying our transformations to NPP workloads. Instead of looping over different operations and multiplexing resources, a pipeline is created by replicating logic as needed. This enables data to flow from input buffers to output buffers at high throughput and without stalling.

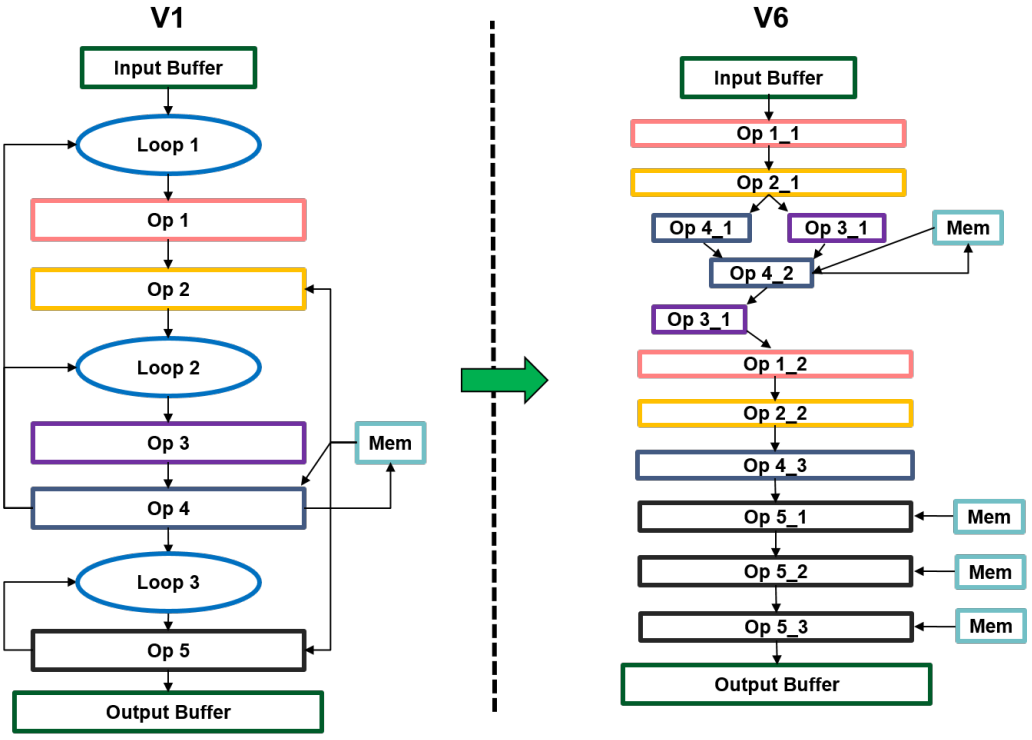


Figure 5·7: Applying transformations to Network Packet Processing

One observation made when implementing NPP workloads is that a Read-After-

Write (RAW) hazard can exist when using BRAMs. BRAMs are a common component of NPP applications since data structures, such as match-tables, use them to store or lookup rules. The limitation of BRAMs is that they cannot read from a location in the cycle immediately after the location is written to. Since these tables can get quite large, register based storage cannot be used. However, we can use registers to temporary hold on to stored values for access immediately after writes. This is referred to as a register cache.

The code for addressing the Read-After-Write hazard using a register cache is inserted using the following process: i) for each array, a single variable is declared outside the main loop in order to preserve its state across iterations, ii) inside the loop, the contents of this variable are copied to a temporary variable, iii) the BRAM array is accessed as a function of the loop iteration; this is critical since it enables the compiler to determine if this memory location was accessed in the previous cycle, iv) the read address of the current iteration is compared with the write address of the previous iteration, v) if they match, the value in the temporary variable from (ii) is used in subsequent computation, vi) if there is no match, the value of BRAM array read in (iii) is used, vii) the temporary variable is stored in the BRAM array, and finally viii) writes to the BRAM array are instead made to the state-preserved variable so that the actual write to BRAM happens in the next iteration.

Workload Evaluation

We provide the implementation results of our optimized application kernels. Percentage resource usage of ALMs and BRAMs is plotted on the primary y axis, while the performance (in the form of Gbps throughput) is plotted on the secondary y axis. Our experimental results demonstrate that applying our code transformation to OpenCL network packet processing workloads results in the generated hardware exceeding 100 Gbps line rates with low resource overhead. We also compare the implementation

of two applications, i.e. AES-256 and SHA-1, against the state-of-the-art Microsoft ClickNP (Li et al., 2016b).

Regular Expression Match (Regex) is commonly used to detect a particular sequence of characters in network packets e.g. for intrusion detection. Given a reference pattern, each character of the input string is compared against all characters in the pattern. Based on this comparison, a two dimensional boolean table is populated, and the comparison result appears at the last element of this table. With regards to the pattern, it is composed of both standard and special characters. Standard characters match with themselves i.e. they can be found in both the pattern and string. Special characters are only found in the pattern, and are used for wildcard matching. In this work, we demonstrate support for special characters by implementing '.' (any character is allowed) and '*' (zero or more occurrences of the preceding character are allowed).

To evaluate Regular Expression Match, we vary the pattern size and measure corresponding throughput. The results are illustrated in Figure 5·8. Since the systolic array processes all bytes of the pattern every cycle, we see that throughput increases with increasing problem size (at the expense of higher latency). This ranges from 100Gbps for a 32 byte pattern to 19 Tbps for the 10K byte pattern. Moreover, the design uses very little chip resource (<2.5% for 10K byte pattern) which allows the Regular expression match modules to be replicated multiple times.

AES-256 is a popular encryption algorithm that encrypts 16 Bytes of data per function call using a 256 bit key. The encryption happens over 14 rounds of processing, typically implemented using a loop, plus an initial round where the key is simply applied to the input data and stored in a state array. With the exception of the last one, each round performs the same set of operations. First, the 256 bit key is expanded to generate a new one for the round. Then, Rijndael's S-Box is used

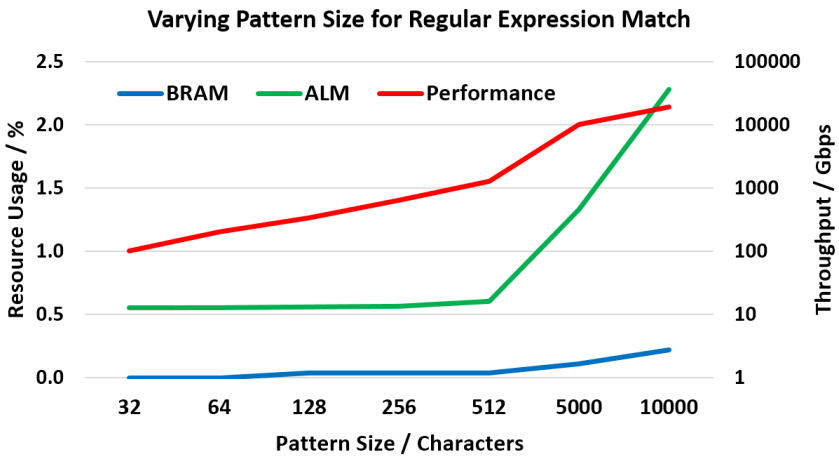


Figure 5·8: Impact of Regular Expression Match pattern size on resource usage and throughput.

to substitute bytes in the state array, followed by a number of bit and byte level operations on the state array. Each round ends with the application of the expanded key to this state array.

Table 5.3 shows the implementation results of AES-256 for i) our designs, ii) ClickNP, and iii) improvement of our Stratix-V implementation over ClickNP. From the results, we can see that our implementation achieves 18% higher throughput, while using 20% fewer RAM blocks (on the Startix-VD5) and only 1% more logic blocks. This is primarily because of manual unroll, which gives the compiler greater freedom in how the computation gets structured, as well as a more reliable view of data dependencies. When operated in block cipher mode of operation, the AES-256 module can be replicated a maximum of 8 times for the Arria 10 FPGA, resulting in an aggregate throughput of 256 Gbps.

Table 5.3: AES-256 Implementation Comparison

	FPGA	**ALM**	**BRAM**	**Throughput**
This Work	Arria 10	10845	305	32Gbps
This Work	StratixV	8168	331	33Gbps
ClickNP	StratixV	6904	465	28Gbps
Improvement	StratixV	-1%	20%	18%

SHA-1 is a popular cryptographic hash function for adding digital signatures to network packets, which are subsequently used to verify the authenticity of transmissions. The input message is processed in chunks of 512 bits and, unlike AES, these chunks cannot be processed in a pipelined manner. This is because the result of a preceding chunk is used as the starting value by the current one. Each chunk is evaluated over 80 rounds of computation, composed primarily of bit operations and integer additions. These rounds require the 16 - 32bit input values to be expanded to 80 - 32bit values, with one input value used per round of computation. Table 5.4 shows the implementation results of SHA-1 in the same format. Similar to AES-256, our SHA-1 implementation outperforms ClickNP while using significantly fewer RAM blocks and only 2% more logic blocks.

Table 5.4: SHA-1 Implementation Comparison

	FPGA	**ALM**	**BRAM**	**Throughput**
This Work	Arria 10	25063	21	1.4Gbps
This Work	StratixV	17708	22	1.3Gbps
ClickNP	StratixV	13635	133	1.1Gbps
Improvement	StratixV	-2%	16%	20%

Another, simpler form of hashing is required in packet processing to index into various data structures e.g. forwarding tables. The input to these hash functions are typically a n-tuple of packet meta data. Unlike cryptographic hashes, these do not have strict requirements for avoiding collisions and can thus be easily parallelized. In this work, we use **Murmur3** as a representative example of such hashing. The Murmur3 algorithm operates on arbitrary number of input bytes in two stages, and transforms an initial seed value to the final hash based on this input. Stage 1 processes input data in chunks of 128 bits. Stage 2 then finishes up the remaining smaller 8 bit chunks.

We evaluated Murmur3 by using four potential Tuples sizes. 1-Tuple consists of a single 32-bit IP address. 3-Tuple is used in layer 3 hashing, and is composed of the source IP, destination IP and IP protocol. 5-Tuple is commonly used for TCP/IP

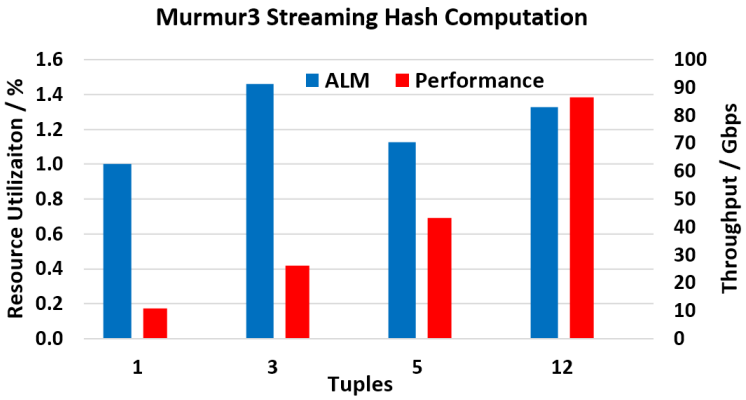


Figure 5·9: Impact of N-Tuple size on Murmur3 resource usage and throughput.

connections, and extends the 3-Tuple by also including the source and destination ports. The total size of 3- and 5-Tuples is 12 and 16 bytes respectively. Finally, we evaluate a 32 bytes OpenFlow 12-Tuple (OpenNetworkingFoundation, 2019). Figure 5·9 shows the results of our implementation. There is no BRAM usage in all four cases while ALM usage is low and does not have significant variations. Since the implementation is fully pipelined, throughput increases linearly with Tuple size; from 10Gbps for 1-Tuple to 90Gbps for 12-Tuple. Similar to Regex, however, the latency will be higher for larger Tuples.

Bloom filters are probabilistic data structures that are used to determine if a given input is present in a set. They trade off accuracy for speed and resource i.e. a bloom filter can give false positives, but is guaranteed not to give a false negative. For a given input, multiple different hashes are computed. If the operation is 'add', the Bloom filter table is indexed by these hash values and corresponding entries set to 1. If the operation is 'find', these entries are ANDed. If even a single entry is 0, the result is false which guarantees that the input is not present in this set. On the other hand, all entries being 1 does not guarantee presence since these bits could have been set by different inputs (collisions). Selection of the number of hashes and table

size is thus important for determining the probability of false positive generation.

To evaluate Bloom filter performance, we perform three experiments: a) we implement a 16Kb Bloom filter and vary the number of hash functions, b) fix the hash functions to 2 and vary Bloom filter size from 2Kb to 1Mb, and c) we repeat experiment (b) but with 4 hash functions. Figure 5·10 shows the results of these experiments. We observe that resource usage increases and throughput decreases with number of hash functions. The former is because the compiler assigns independent RAM blocks to each table, which is inefficient since it leads to under utilization of individual blocks. Thus, more blocks are needed to construct the same 16Kb table. On the other hand, throughput decreases because larger resource overhead reduces operating frequency (all hash functions are computed from the same fixed size key, and so the effective input size is independent of the number of hash functions). When number of hash functions is fixed and table size varied, we observe that BRAM usage increase, ALM usage remains consistent, and performance has small variations.

HyperLogLog is another probabilistic data structure used to estimate the cardinality of a multi-set, that is, how many unique values are present in the set. This is particularly useful for large data sets, where calculating the exact value is impractical since it requires a large memory footprint. We test HyperLogLog for bucket sizes ranging from 512 to 16K. Figure 5·11 shows the results of our experiements. In all cases, the kernel operates in a fully pipelined manner, processing a new input every cycle. While ALM usage is invariant to the number of buckets, BRAM usage grows proportionally since we are using BRAM based arrays. The performance also drops as expected due to increase in BRAM usage. However, this drop is relatively small (5Gbps). An interesting observation is that 2048 buckets give a better throughput than 1024, despite using larger resources. This highlights the need to explore design space before selecting architecture parameters, since certain sizes map better to the

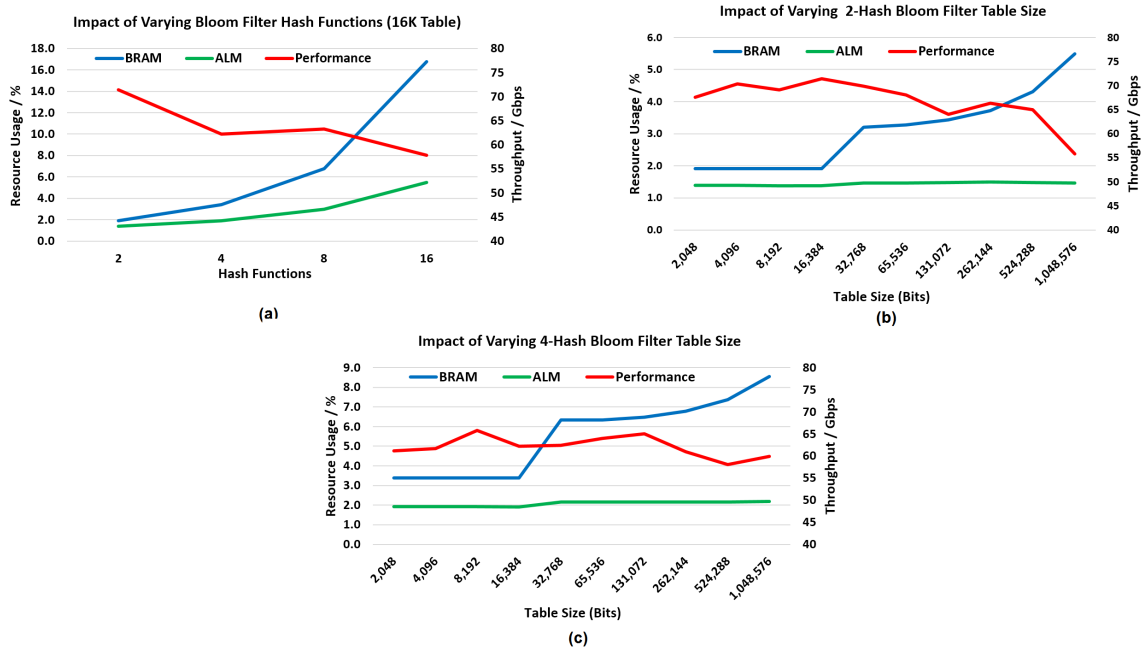


Figure 5·10: Impact on Bloom filter resource usage and throughput due to variations in a) number of hash functions, b) table size for 2 hash functions, and c) table size for 4 hash functions.

FPGA.

Cuckoo Hash Tables attempt to reduce collisions by using two independent tables, each with its own hash functions. Each table entry typically contains the unique key used to compute hashes, as well as an association value (e.g. action value for a match-action table). Lookups and deletions occur with O(1) complexity by simply addressing each table and performing the appropriate operation. On the other hand, insertion is unbounded. An insert is first attempted on one table. If a collision occurs, the existing value is kicked out and the algorithm attempts to insert this value in the next table. The above process repeatedly inserts and kicks out table entries until either no valid entry is kicked, or a maximum attempt limit is exceeded. An unbounded insert cannot be reliably implemented since the compiler cannot synchronize this data path with the bounded find and delete operations. As a

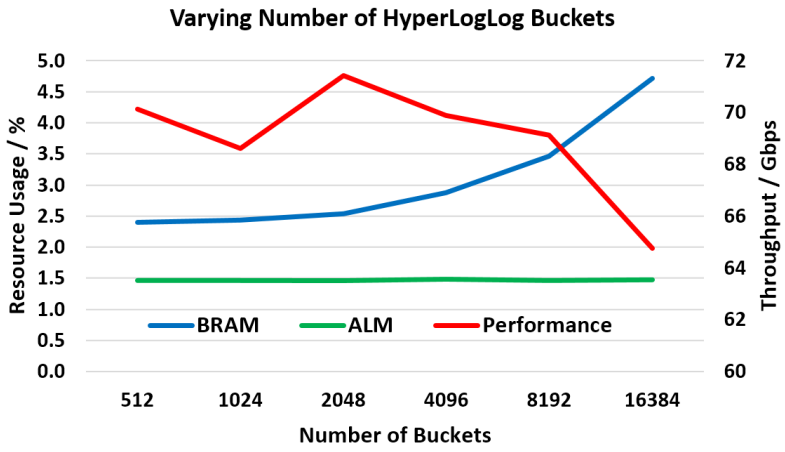


Figure 5·11: Impact of tables size on HyperLogLog resource usage and throughput.

result, we make a simplifying assumption that inserts can be converted into an O(1) problem by only performing a single insert per input. If a table entry is kicked, we assume that is handled by logic external to the Cuckoo module and applied as a new input to the module.

Similar to bloom filter, we perform three sets of experiments for Cuckoo hash: a) using a 4 Byte key, 16 Byte table entry and varying the size of tables, b) repeating (a) but with 32 Byte table entries, and c) using 16K total table size, 32 Byte entry and varying key size. In all cases, the input size is equal to the table entry size. Figure 5·12 shows the results of these experiments. Consistent with previous results, we observe that in both Figure 5·12a and Figure 5·12b , increase in BRAM usage is proportional to increase in table size. On the other hand, throughput decreases due to lower operating frequencies. We also observe that ALM usage doubles when entry size doubles, but does not change with table size. Finally, we observe in Figure 5·12c that key sizes in our tested range do not significantly impact either resource usage or throughput.

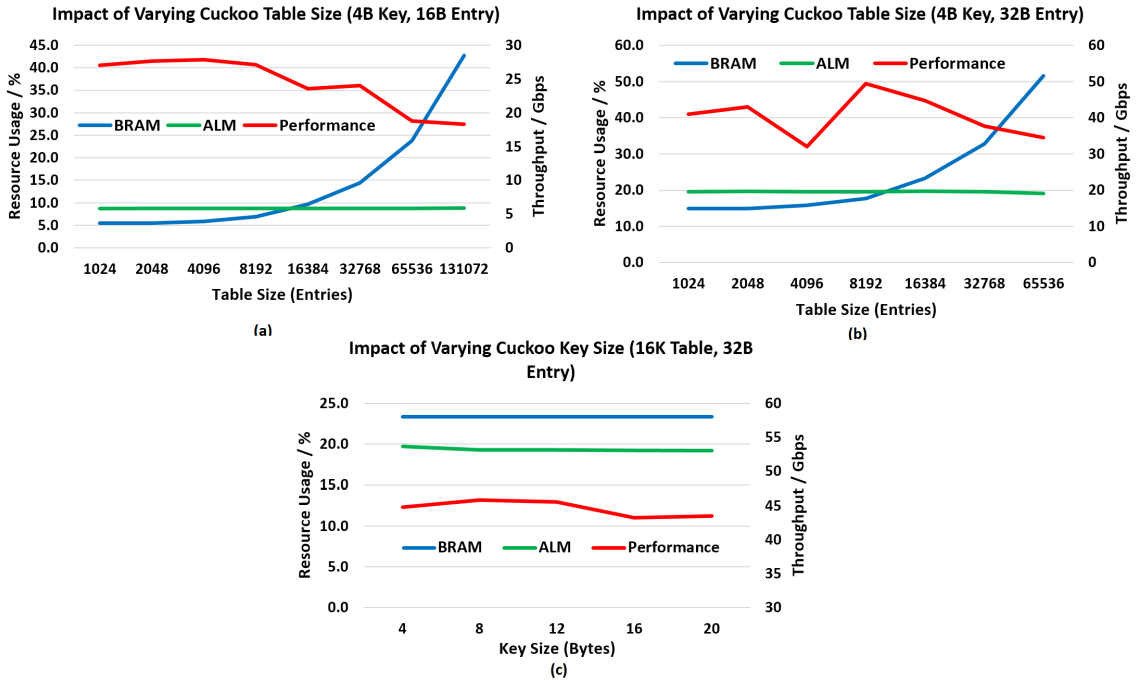


Figure 5·12: Impact on Cuckoo hash resource usage and throughput due to variations in a) table size of 16 byte entries, b) table size of 32 byte entries, and c) key size for 16K size tables and 32 byte entries.

5.4 Conclusion

In this chapter, it is shown that the Performance-Programmability gap of FPGA OpenCL can be reduced and that performance comparable to GPUs, and even Verilog-based implementations, can be achieved. By using CPU code as a starting point and performing a series of simple optimizations that augment common best practices, highly efficient FPGA hardware can be generated. The performance impact of all of the optimizations is characterized using a number of parallel computing dwarfs. The overall impact of this characterization is that programmers can now follow a script for optimizing their FPGA OpenCL kernels and achieve HPC performance. Moreover, auto-tuners can be developed to automate the generation of efficient hardware. For parallel computing dwarfs, the optimized kernels have been shown to outperform CPU

and previous FPGA OpenCL designs. Using Stratix-10, they can also outperform a high-end GPU. Most importantly, it is demonstrated that the optimizations can, on average, achieve performance values within 12% of hand-tuned HDL code. For network packet processing, results show that OpenCL generated hardware can exceed 50 Gbps line rates with low resource overhead. Moreover, we demonstrated that our designs for AES-256 and SHA-1 can perform≈20% better than current state of the-art.

Chapter 6

A First Principles Approach to Indentifying and Removing Optimization Blockers

6.1 Motivation

The logical extension to work done in Chapter 5 is to either write scripts/pre-processor for automatic application of the proposed transforms, or to integrate them into a HLS compiler with certain options. However, we found our approach in Chapter 5 to be brittle. We only conjecture why the transformations work. This means that any small change to the compiler can the either change the set of applicable transforms, or their ordering, or both. Before a large amount of time is spent on modifying compilers or building pre-processors, we need to get a better understanding of optimization blockers in HLS. We need to know: i) what those optimization blockers are? ii) what input transformations work? iii) What additional transformations are possible? and iv) what pragmas are needed?

6.2 Optimization Blockers

For every transformation, compilers are forced to use a conservative, safety-first approach, and only perform a transform if the validity can be guaranteed. An optimizing transform may be blocked if it: i) modifies code functionality, instead of structure only, ii) can result in a failure to compile (e.g. exceeding available target device resources),

iii) is based on information available at run-time (can sometimes be addressed by creating multiple versions of the data path, with some catering to special cases and others to generic ones e.g. implementing both division with a commonly-occurring value and division with a variable), iv) requires a global view of the computation (e.g. dedicating too many resources to a local computation may result in resource starvation for the other procedures), and/or v) is based on implicit code behavior that may be visible to the developer, but cannot be reliably extracted by the compiler. Simply put, optimization blockers occur when a compiler writer is not being allowed to infer an optimization.

An example of optimization blockers in C/C++ is memory aliasing. This occurs when a compiler cannot guarantee that two objects, pointed to by separate pointers, do not fully or partially overlap in memory i.e. two pointers point to different locations. Algorithm 15 illustrates an example of this. Here, the value e can be either 100 or 200 (and not always 100), since it is possible for both $*c$ and $*d$ to be pointing to the same memory location. Memory aliasing can be avoided by using the "restrict" keyword. However, this requires the developer to have sufficient expertise in low level behavior of C/C++ code.

Algorithm 15 Memory Aliasing

```
1: int foo(int* c, int* d)
2:     int a = 100;     int b = 200;
3:     *c = a;          *d = b;
4:     int e = *c;
5:     return e;
```

We conjecture that all optimization blockers in HLS are of one type i.e. compiler writers expect developers to implement HLS as HDL. This is because compilers typically perform a literal translation of HLS to HDL e.g. one variable is one register and one thread of execution is one pipeline. Getting good performance for an algorithm thus becomes difficult since it involves having an understanding of: i) how syntax

maps to machine code, ii) how the machine then executes the code, iii) what the performance is, relative to other binary code, iv) identification of where optimization blockers may occur, and v) how to modify sub-optimal code structures.

6.3 Task: Identifying Optimization Blockers

In this section, we outline our approach towards identifying optimization blockers for a HLS compiler. Figure 6·1 illustrates this approach. We first build a model for

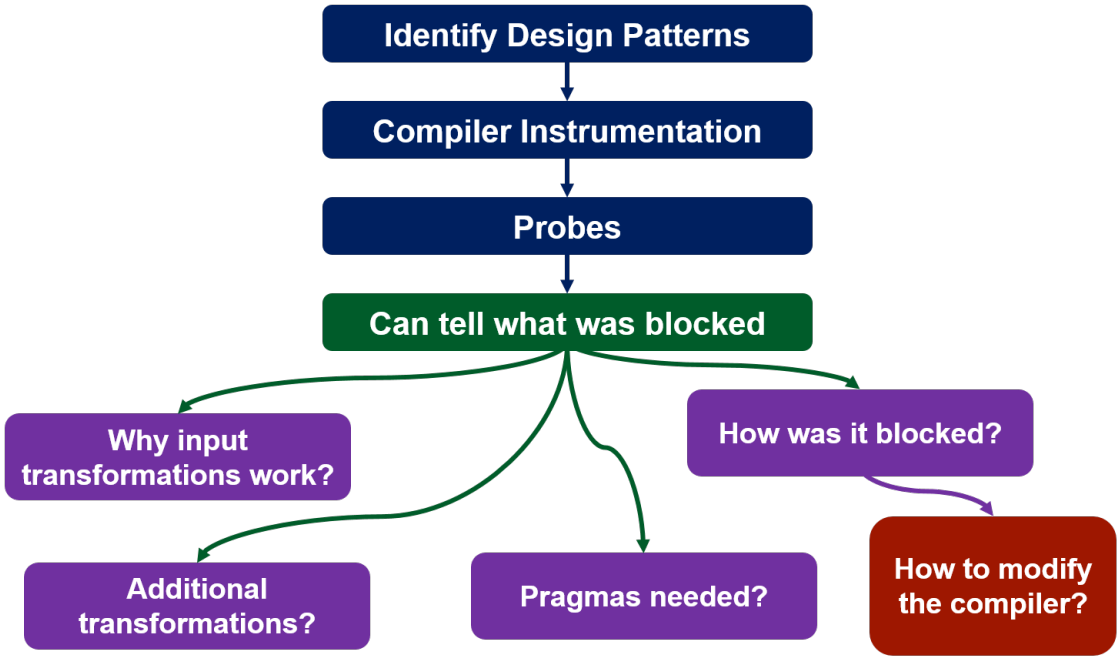


Figure 6·1: Process for Identifying Optimization Blockers

FPGAs by identifying a set of core design patterns that the compiler should be able to infer and implement effectively. Then, we instrument the HLS compiler to determine what it has inferred given an input code. We then build a set of probes which contain individual design patterns in relative isolation, so that we can determine compiler effectiveness for each. Finally, by running these probes through the compiler and looking at instrumentation results, we can tell what optimizations are blocked. By extension, we can start to answer the questions asked above. Moreover, if we can also

determine how these optimizations were blocked, we can better understand how the compiler can be modified.

6.4 Eigenspace Mapping

We formalize characterizing compiler effectiveness using Equation 6.1. Let $\mathbf{A}$ be code transformations applied to a multi-dimension eigenvector $\mathbf{v}$ (Equation 6.2). Each dimension of this eigenvector represents a different feature of the code that impacts either performance, or resource usage, or both. The magnitude of this impact is represented by a multi-dimension vector of eigenvalues $\boldsymbol{\lambda}$ (Equation 6.3). The goal of our profiler is to measure these eigenvalues based on analysis done on the post-transformation eigenvectors. Given that these eigenvalues capture implementation effectiveness of core design patterns, it is even possible to estimate the run time of a given code without compiling it to RTL (Equation 6.4).

$$\mathbf{A}\mathbf{v} = \boldsymbol{\lambda}\mathbf{v} \tag{6.1}$$

where

$$\mathbf{v} = \left[\begin{array}{c|c|ccc|c} v_{11} & v_{12} & v_{13} & \dots & v_{1N} \\ v_{21} & v_{22} & v_{23} & \dots & v_{2N} \\ v_{31} & v_{32} & v_{33} & \dots & v_{3N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ v_{N1} & v_{N2} & v_{N3} & \dots & v_{NN} \end{array}\right] \tag{6.2}$$

$$\boldsymbol{\lambda} = \left[\lambda_1 \mid \lambda_2 \mid \lambda_3 \quad \dots \mid \lambda_N\right] \tag{6.3}$$

and

$$Performance = P(\boldsymbol{\lambda})\ cycles \tag{6.4}$$

6.5 The FPGA Computing Model

In this section, we present our proposed computing model for FPGAs. Specifically, we list and discuss core design patterns of hardware, which form the elements of this model, and how they relate to performance and efficiency.

6.5.1 Spatio-Temporal Computing

To build an FPGA model, we must first identify the practical high level goal for designing hardware. Typically, FPGAs are considered to be spatial computers. The goal for spatial computing is to define processing elements and the connections between them; information used by the compiler to then generate specialized architectures, capable of exploiting arbitrary levels of data, task and instruction parallelism. Specifying custom circuits with dedicated logic for each function is one of the most efficient approaches to computing, since it eliminates a number of inherent bottlenecks of traditional temporal computers (such as CPUs). That being said, spatial computing is not without its own set of constraints. Specifically, a purely spatial computer is built on the pretense of infinite resource; a concept that is certainly not practical. Not only is resource a finite quantity, but it is also an important factor considered when estimating performance. Consuming a lot of certain resources, such as BRAMs, can result in lower operating frequencies since larger MUXs may need to be implemented. Consuming too much may result in failure to compile altogether, since there would be insufficient resources available to generate pipelines, e.g. not enough functional units (e.g. DSPs) or interconnect fabric. In certain cases, the control plane generated can also end up having substantial overhead. Therefore, practical circuit design has to be mindful of a device's resources and performance by incorporating concepts of both spatial and temporal computing. That is, developers should express computations spatially, but also attempt to reuse logic whenever doing so does not significantly re-

duce performance (especially in the case of critical resources such DSPs). This is why we believe that the high level goal in designing hardware is to build a spatio-temporal computer, and as such we include elements of both spatial and temporal computing in our model.

6.5.2 Spatial Elements

Here, we present some of the core spatial elements of the FPGA model (Figure 6·2). The high level goal of these elements is to maximize parallelism for both computations and memory accesses. For each element, we describe i) why that particular element is a core design pattern, and ii) possible optimization blockers that can prevent this design pattern from being reliably inferred.

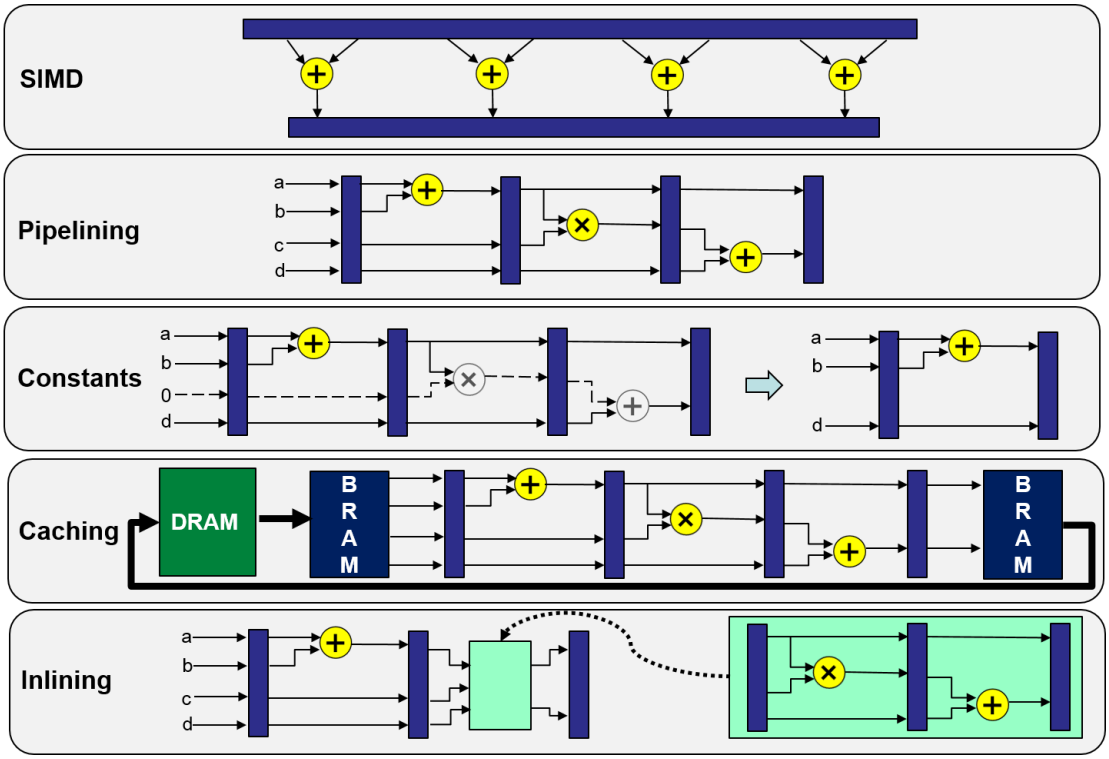


Figure 6·2: Set of spatial design patterns targeted in this dissertation

SIMD

The manifestation of SIMD is HLS code typically corresponds to loops with finite limits, static increment sizes and no loop-carried dependencies. In such a case, it is expected that the compilers identify the hardware needed to execute a single loop iteration, and make multiple copies of it so that all iterations can execute the same instructions at the same time. The advantages of SIMD are that it: i) reduces the time taken to execute a loop, and ii) eliminates overhead of loop control logic, as well as stall logic at the entry of the loop (if the loop is nested). While previous work, such as (Karrenberg et al., 2013), has shown SMT solvers can be used to efficiently detect SIMD, a compiler might still preserve the loop due to potential global resource starvation. This means that the compiler is not certain that assigning more resources to a loop will leave sufficient chip fabric for implementing the rest of the algorithm. Moreover, even if SIMD is exploited in a loop, another bottleneck could be present (perhaps another loop), which would result in limited performance improvements. Overall, knowing how much resource to assign to each loop is a non trivial optimization, since the entire algorithm has to fit on chip whilst maximizing performance. And therefore, without appropriate guarantees or manual tuning, certain compilers may not do any SIMD (despite knowing the opportunity for doing so exists) since they are unable to determine the optimal SIMD parameters for each loop.

Pipelining

Pipelining refers to the exploitation of instruction level parallelism in an algorithm. It enables dependent instructions to execute concurrently by generating separate hardware for them, and connecting outputs of preceding instructions with inputs of subsequent ones. If the subsequent instruction is not evaluated in the next cycle, a delay is added to the data (typically using shift registers). Several pipelines can also

be constructed in parallel, corresponding to one or more tasks. The advantages of pipelining are that it: i) reduces the critical path in the RTL and allows concurrent execution for operations such as floating point arithmetic, ii) increases throughput by allowing multiple iterations of a loop to execute in parallel, with each iteration evaluating different instructions, and ii) decreases resource needed for implementing loop control and stall logic, similar to SIMD. Note that pipelining all iterations of a loop for concurrent access (by implementing multiple copies of a pipelined loop body in a sequence) is generally only useful if this loop is nested, and these replications can fit on chip; if a loop is not nested or consumes too many resources, it is better to have only one copy of the pipelined loop body.

The challenge with pipelines is that while a compiler may easily detect what pipelines are to be created, it may not be able to infer the best approach to implement them. Specifically, it may not infer that pipeline registers must be implemented as registers and not BRAMs. BRAMs are available as a combination of blocks of fixed sizes (e.g. M20K in Intel FPGAs). These blocks can at best be dual ported, which means that only two pipeline registers can be implemented per block. Consequently, a large number of blocks are needed to implement a deep pipeline, which in turn means very little memory is left over for implementing caches and on-chip databases. Moreover, the utilization of these blocks is also expected to be extremely low. There are two potential optimization blockers that can prevent a compiler from inferring registers over BRAMs. First, if the size of the pipeline register array exceeds a certain threshold, the compiler may be hardwired to store the array in a BRAM in an attempt to maximize resource efficiency. This is because memory available using Logic Elements (e.g. MLABs implemented using ALMs in Intel FPGAs) is far less than memory available using BRAMs. Second, if the compiler cannot guarantee an access pattern e.g. data read from RegArray[i] is always processed and stored in

RegArray[i+1], then it may make the default assumption of random access.

Caching

Off-chip Memory transactions involving individual variables typically result in stalls while the memory access takes place. Not only does an individual access waste available memory bandwidth, but multiple such accesses within the same loop iteration have an additive effect since one off-chip access cannot mask another. The stall may even be large enough to prevent the loop from being pipelined. Caching reads and writes to on-chip memory, even if there is no data reuse, helps remove these off-chip access stalls from the critical path of a loop. Since memory can now be accessed in blocks instead of individual variables, coalescing is also possible which improves bandwidth utilization (and hence performance).

Caching is difficult for a compiler to infer based solely on knowing what off-chip memory transactions occur (and where). Specifically, there are two potential optimization blockers for caching. First, it is possible for caching to change the overall result. For example, the result of the algorithm may change if it has two or more concurrently running computations that share the same off-chip memory space. By caching writes, one computation does not commit data to memory at the instance that it was supposed to, thereby resulting in an incorrect read by another computation (that assumed a write took place). Similarly, caching reads means the value read at the start of execution may be different from the value at the instance that it was actually intended to be read. Second, caching requires additional chip resources that were not explicitly requested by the developer, and could thus result in a global resource starvation problem.

Constants

Constants refer to values that are fixed for the duration of an algorithm. They can be declared both as individual values and as arrays. Use of constants not only reduces off-chip memory access, but can also minimize the resources needed for that computation by pre-computing results if possible. For example, in the case of compile time constants, a guaranteed multiplication with 0 or 1 means that the multiplier may be removed altogether. Moreover, both compile time and run time constants can be directly applied to the inputs of pipelines; this reduces the interconnect used to fetch them from a different location on the chip and improves design routability. While a compiler generally infers and leverages individual constants (both compile and run time) with ease, it may not be able to efficiently implement an array of compile time constants that are being accessed by multiple pipelines (and pipeline stages) concurrently. A naive approach is to always replicate the array for each sink so that pipelines can operate stall free. However, if the array was large enough, having an independent copy per access could saturate board resources and the design will not compile. The optimization that a compiler is expected to do is to create smaller arrays that are a subset of the larger one and are tailored to each sink i.e. these arrays only contain values required by the target sink in the pipeline. This can significantly reduce the overhead of constant memory replication, and even eliminate it altogether if every sink accesses a unique memory space. However, a potential optimization blocker is the compiler's inability to correctly infer access patterns for a large constant array.

Inlining

Inlining refers to replacing a function call in the pipeline with the actual logic of that function. The advantage of inlining is that it tightly integrates hardware, resulting in: i) elimination of redundant computations and memory accesses, ii) better

design routability, iii) lower latency, and iv) the ability to optimize both pipeline and function together (global optimizations), including non-trivial optimizations such as pipeline re-ordering or merging pipeline stages. The degree to which these benefits of inlining are exploited, however, depends on the compiler. While the compiler may easily add function logic to the pipeline, it may be blocked from performing aggressive transformations if it cannot be certain that functional correctness for the entire algorithm is preserved.

6.5.3 Temporal Elements

Unlike spatial elements, the high level goal of temporal elements is not to maximize parallelism. Rather, it is to reuse hardware in a manner that does not reduce performance (Figure 6·3). This is typically required when a problem requires more resources than available for spatial implementation, or if the problem has too many dependencies to be parallelised. Similar to above, for each element we will describe: i) why that particular element is a core design pattern, and ii) possible optimization blockers that can prevent this design pattern from being reliably inferred.

Loops

Loops are perhaps the most common approach to folding a problem. Instead of specifying hardware for every possible instruction in the algorithm, developers define a hardware block (loop body) which is reused multiple times by varying inputs (based on the loop iteration number). As discussed before, the block can also be pipelined, enabling multiple loop iterations to concurrently use the block. Overall, an optimized, pipelined loop can reduce the execution complexity from $O(NM)$ to $O(N + M)$, where N is the number of loop iterations and M is the latency of a single iteration. A possible optimization blocker, however, is nested loops. If loops are nested, then the compiler cannot optimize the outer loops since each iteration of this outerloop

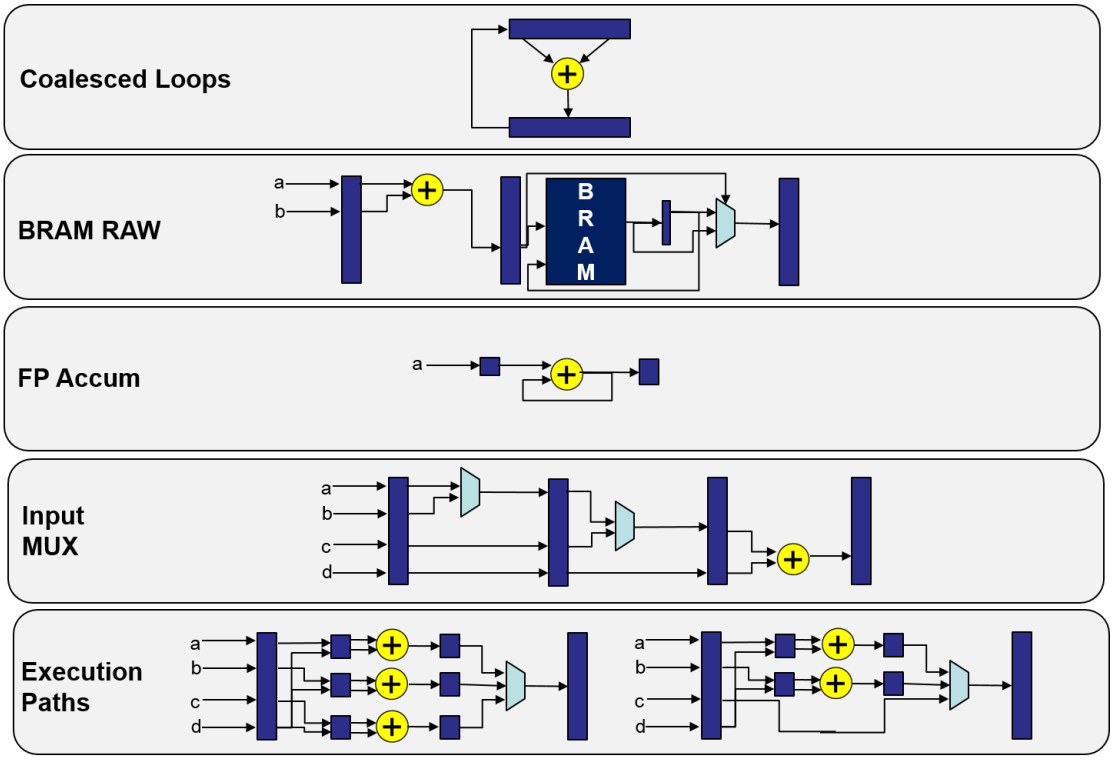


Figure 6·3: Set of temporal design patterns targeted in this dissertation

requires multiple iterations of the inner loop. Since the inner loop hardware block cannot be replicated (global resource starvation), outerloop iterations may not get scheduled till the previous one is complete in order to guarantee functional correctness. In this case, for example for two nested loops, the execution complexity becomes $O(N_{outer} * (N_{inner} + M))$ (assuming there is no hardware block exclusive to the outer loop).

BRAM Read-After-Write Hazard

In the pipelining section above, we discussed how pipeline registers should be inferred as registers. For certain cases, however, BRAMs are actually preferred. A popular example of this exception is network packet processing applications, where meta data of incoming network packets is hashed and used to index into complex data structures.

Since these data structures can be Megabits in size, it is impractical to implement them using registers. Hence, they are stored in single BRAMs which is accessed by every incoming packet (resource reuse). Using BRAMs in the critical path of a pipeline is likely to result in stalls, since the compiler cannot guarantee that a valid read can happen in the same cycle as a write to the BRAM (RAW hazard). A compiler should be able to infer this design pattern and optimize the stall in cases where access patterns guarantee that the hazard will not occur e.g. no read will attempt to index a location that was just written to. It may be blocked from doing so, however, if the BRAM access appears to be random and a valid, "hazard-free" pattern cannot be determined with full certainty.

Execution Paths

Conditional branches in loops, including assignments done as part of loop initialization and comparisons done to detect loop termination, result in multiple possible execution paths for a computation. For temporal computers such as CPUs, the goal is to optimize the trade-off between number of paths and path depth. This is because having more paths increases the probability of branch misprediction (decreases performance), but more paths could also substantially reduce the work done per iteration by eliminating unnecessary computations (increases performance). For computers with spatial elements, such as FPGAs, having multiple paths has no significant benefits. The possibility of multiple paths means that the generated hardware must either support superscalar execution (higher hardware costs) or stall all smaller paths to match the latency of the worst case (potentially no performance improvement). Typically, the latter is preferred due to its lower complexity and overhead. In both cases, the hardware for each path is generated irrespective of its utilization. For example, a path using 99% of the board's DSP resources could end up executing for only 1% of the iterations.

Thus, an optimal hardware design minimizes the number of paths so that the loop can: i) achieve a high aggregate resource utilization by using the same set of resources for every iteration, and ii) tightly integrate paths to reduce the overall resource overhead. One approach to reducing paths is to push everything out of branches to parents as long as there are no conflicts; performing this iteratively may result in emptying all branches, and creating a single path. However, a compiler may be blocked from doing so if the corresponding transformation pass occurs at a late stage of the compilation. For example, if each path has already been optimized independently, then the implementation approach of the paths may diverge substantially. At this point, the compiler may not be able to reliably determine an efficient approach to merging them.

Input Multiplexing

As was highlighted in execution paths above, resources assigned to a computation may not be utilized every iteration of a loop. This is not always because some iterations avoid the computation altogether. In some cases different paths exist because the variables being operated on are dependent on the loop state. For example, if every even iteration of a loop adds 10 to an accumulator and every odd iteration adds 20, then two paths could be generated; each path has its own adder and is hardwired to add either 10 or 20 (but not both). We refer to this as output multiplexing, where final results for all paths are generated, and then one is selected based on the loop iterator value. A possible optimization that compilers can perform is to move MUXs to the inputs of resource common to paths. This allows the same resource to be used for multiple paths, especially if the only difference in a path is the inputs. In the example above, if we first select between 10 and 20 based on the loop state, then we only need one adder which accumulates the result of this selection. Potential optimization blockers here are that such a transformation may not be visible to the

compiler, or would require algorithmic changes beyond the scope of the compilers capabilities.

Floating Point Accumulator

Workloads such as matrix multiplication require results of loop iterations to accumulate to a sum variable. Since the loop body is generally well pipelined, it is expected that a new result is available for accumulation every cycle. It is thus important for the hardware to perform the accumulation every cycle as well, or else the pipeline will stall and result in a significantly lower throughput. The challenge here is if the data type is floating point, then addition can take multiple cycles to compute. If native support for single cycle floating point accumulation is not available (e.g. hard accumulation unit on the FPGA), the compiler may be unable to address this challenge and choose to stall the loop for a few cycles at every iteration. A possible reason, similar to input multiplexing, is that supporting high throughput floating point accumulation would require algorithmic changes beyond the scope of the compilers' capabilities. Another reason is that these changes can result in reordering of the computation.

6.6 Compiler Instrumentation using Static Profiler

6.6.1 Method

Figure 6·4 illustrates our static profiler based method for generating high quality hardware. We begin by extracting the Single Static Assignment (SSA) IR representation for the input HLL code after the compiler has performed optimizing transforms. Then, we extract functions from the IR for evaluation. For each function, we run a number of algorithms which determine what the compiler is inferring and use this information to ID optimization blockers. Then, using the results of code evaluation, we run another algorithm to estimate the performance of compiler output and com-

pare it with previous revisions of the input code. Based on the results of both code evaluation and performance estimation, we pre-process the input HLL code to remove optimization blockers and improve performance. This revised HLL code is then input to static profiler again for verifying the result of pre-processing.

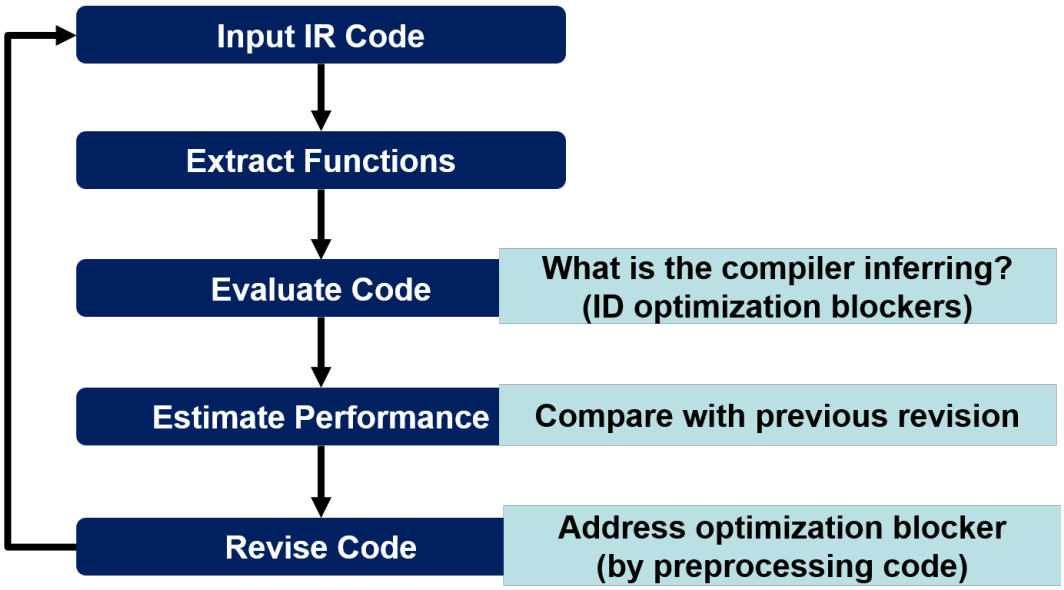


Figure 6·4: Method: Compiler instrumentation using static profiler

6.6.2 Metrics (Eigenvectors)

Here, we propose a set of eigenvectors that must be profiled to evaluate the impact of compiler code transformations. This set includes:

1. **Floating Point Operations Per Cycle**: The maximum floating point throughput that the code is capable of achieving. It is dependent on how many DSP blocks are being used for floating point arithmetic, and the type of operation implemented by each block.

2. **Fixed Point Operations Per Cycle** : Similar to (1), this is also a measure of the maximum throughput, but for fixed point arithmetic operations.

3. **Inlining Efficiency**: Effectiveness of detection and tight integration of two pipelines expressed in separate functions or instruction sequences in the high level representation.

4. **Concurrency**: The amount of parallelism that a code can employ based on the dependencies between instructions i.e. how many instructions can execute in the same pipeline stage, and what is an estimate of the number of pipeline stages?

5. **Device Utilization**: The number of unique execution paths that will be generated, and the depth of these paths.

6. **Stalls**: Occurrences of stalls due to BRAM RAW hazard, off-chip memory access and floating point accumulation.

7. **Loop Efficiency**: How efficiently can loops be implemented based on detected stalls, trip counts and nesting.

8. **Local Memory Efficiency**: Size of local memories and replication required to meet stall-free execution.

9. **Global Memory Efficiency**: Size of global memories (typically constants) and replication required to meet stall-free execution.

6.6.3 Algorithms

Overall Profiler Flow

Figure 6·5 illustrates the overall profiler flow. We first read the file containing assembly code by calling the "ll_read" routine. As the raw data is being read in, we process each line in the file to decode it using the "decode_instruction routine, and store it as an array of instruction "structs". As shown below, an instruction is composed of

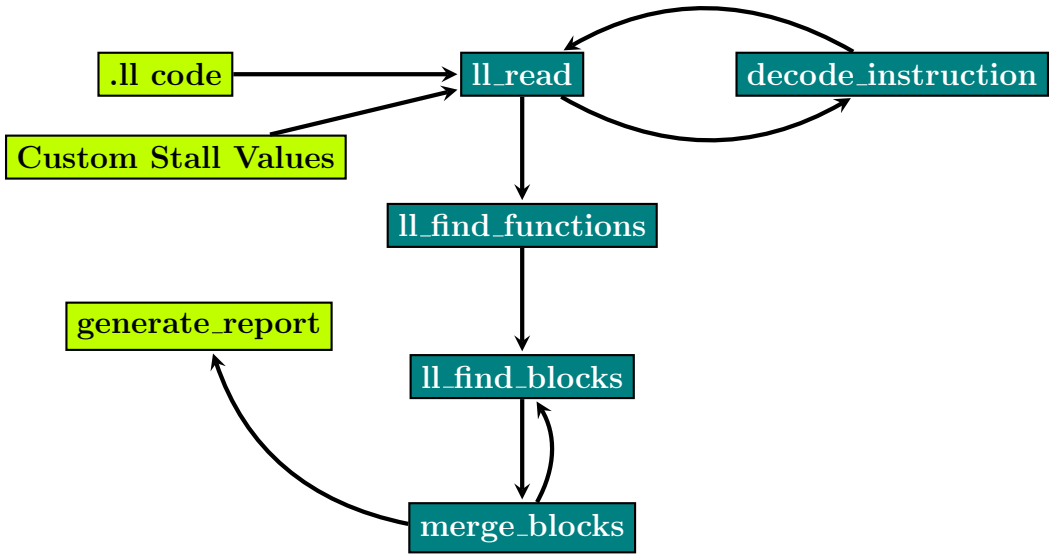


Figure 6·5: Overview of Profiler Flow

the operation name, a single output stored as a <name, type> pair, and an array of inputs. Next, we perform multiple passes over this array of instructions in order to generate a compatible (wrt our analysis algorithms) hierarchy of data structures. These are shown below as the "block" and "function" data structures.

Each file generates an array of functions, with each function containing an array of Phi blocks (referred to as simply "blocks" in this chapter), and each block containing an array of instructions. This hierarchy is created top down. The first pass identifies functions defined within the input file by searching for appropriate keywords. Data stored for each function in this pass includes: i) starting line, ii) ending line, iii) function name, and iv) an array of arguments for the function. The second pass identifies blocks in each function body; these are segments of code bounded by branch instructions. For each block we store: i) starting line, ii) ending line, iii) block name i.e. label used to identify the block as a branch source/destination, iv) parent function name, and v) a copy of all instructions in this block. We also create a single block structure by merging individual blocks in a function, called "AllBlocks"; this is done

using the "merge_block" routine. Finally, we run our algorithms over each function and generate the corresponding report.

struct *instruction* (4.7)

```
string name;
vector <string> inputs
pair <string name, string type> output;
```

---

struct *block* (4.8)

```
int start;
int end;
string block_name;
string function_name;
vector <instruction> lines;
```

---

struct function (4.9)

```
int start;
int end;
string name;
vector <string> args;
vector <block> blocks;
block AllBlocks;
report rep;
```

We also allow the user to define custom stall values for different types of hazards (listed later in this section). The goal behind customizing these stall values is twofold. First, the magnitude of a stall depends on the target hardware. For example, the number of cycles taken to read data from off-chip will vary depending on whether the board supports DDR3, DDR4 or HBM. By being able to tune the stall values appropriately, we can get better estimates of performance. Second, larger stalls can mask smaller ones, resulting in a flawed analysis if a particular model element is being tested in isolation. For example off-chip access can have a stall of hundreds of cycles, while floating point accumulation takes orders of magnitude less. Thus, if a loop has

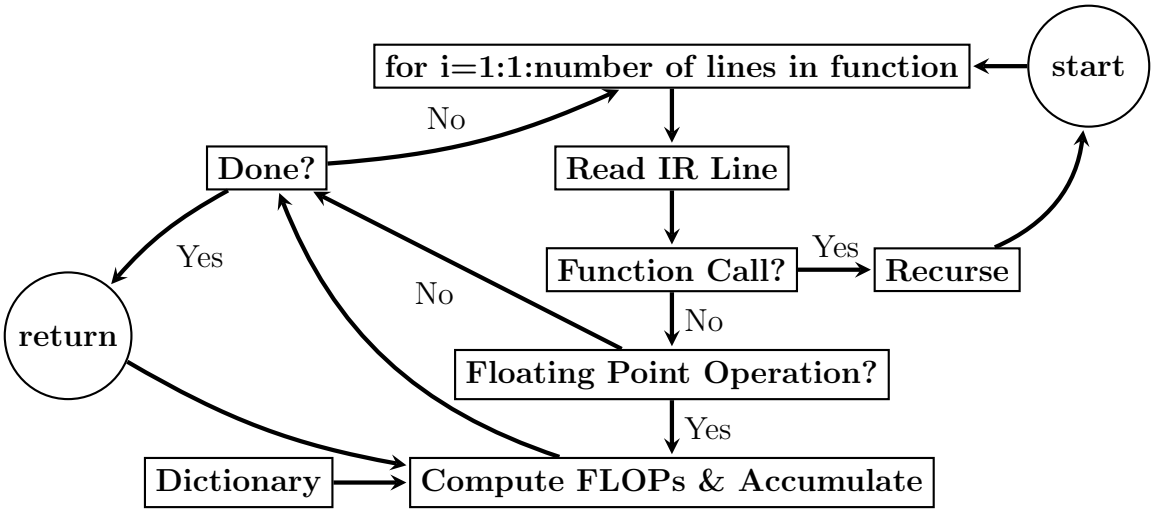


Figure 6·6: Floating Point Operations per Cycle

both hazards, we would only see the effects of off-chip access in the report. However, by suppressing the off-chip access stall, we can enable the floating point accumulation element of our model to be tested in isolation.

Floating Point Operations per Cycle

Figure 6·6 illustrates our algorithm for identifying the number of floating point operations in a function. For each instruction, we check the operation type to determine if it is either performing a floating point operation or a call to another function defined in the input file. For the latter, we recurse and accumulate the result of this recursion into the final count. For the former, we lookup this operation in a dictionary which maps operations to FLOPs/cycle. For example, FP_Add will be 1 FLOPs/cycle while FP_MulAdd (multiply-accumulate) will be 2 FLOPs/cycle.

Fixed Point Operations per Cycle

We reuse the algorithm for floating point operations, except now the algorithm searches for fixed point operations in each instruction. We include arithmetic operations and integer comparisons in this search.

Inlining

Inlining analysis is composed of two results: i) number of user function calls and ii) integration effectiveness. Evaluating (i) gives a high level measure of a compiler's ability to replace a functions call with the corresponding function body. This value is generally expected to be zero since all functions are synthesized on the same chip and inlining, in the context of (i), simply means wiring two pipelines together. The implied requirement here is that the latency of each function is deterministic, and the compiler is able to use this latency value to add appropriate delays to all paths without a particular function call. The second result extends inlining requirements of (i) by further investigating the effectiveness with which inlined pipelines are tightly integrated. Since an exhaustive evaluation in this regard is beyond the scope of this work, we evaluate a single case as proof of concept. Specifically, we evaluate the DSP Efficiency of a hardware design i.e. how effectively can a compiler detect the connectivity between floating point additions and multiplications expressed in different functions, and replace them with floating point "Mul-Add" units. "Mul-Add" units can be more efficient than individual operations since they get better utilization from a DSP block (Intel and Xilinx have both multipliers and adders in the same DSP block).

Figure 6·7 illustrates our algorithm for determining DSP efficiency. We begin by scanning the code for all floating point "Mul-Add" units, as well as individual floating point multipliers and adders. Then, we iterate over these lists and search for unique pairs of multipliers that connect to the same adder, and keep track of this count. Finally, DSP efficiency is computed as a ratio of the number of such multiplier and adder pairs versus the number of "Mul-Add" units plus the pair count above. Ideally this value should be 1.

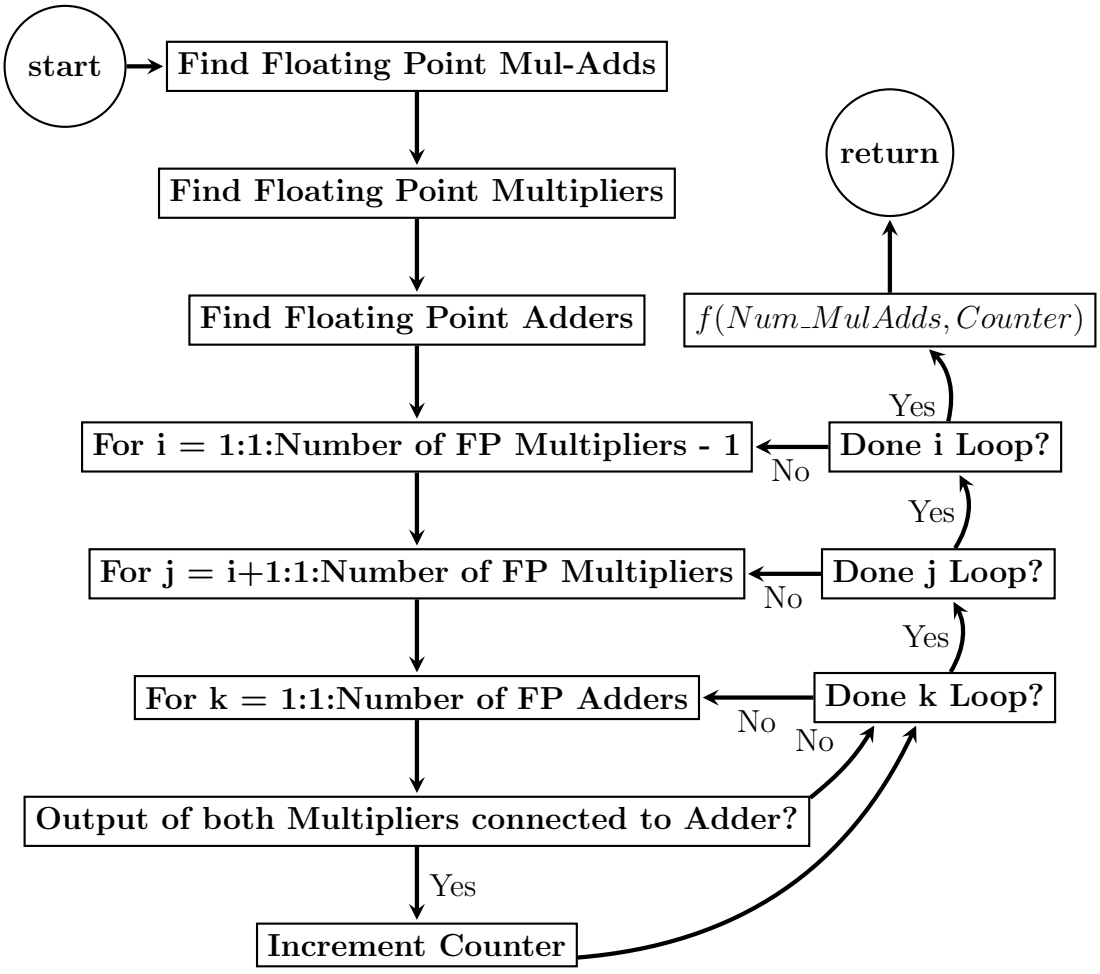


Figure 6·7: DSP Efficiency

Concurrency

We evaluate concurrency as a measure of the number of pipeline stages and the amount of data/task parallelism per stage in a given function. This uses three separate routines: i) building a global block by inlining all function calls (different from All_Blocks which only contains blocks local to a function), ii) building a pipeline based on dependencies between instructions, and iii) evaluating the number of floating point, fixed point and memory point operations being done at each pipeline stage. In the above analysis, latency of all instructions is assumed to be the same and each stage only performs instructions that have no inter-dependency i.e. combination path size of 1. We expect that the actually pipeline depth can vary if delay stages are added or if multiple stages are combined by having a longer combinational path. However, our dependency analysis still provides useful insights when compared in a relative manner i.e. analysis of different optimization iterations of the same application.

Figure 6·8 illustrates our algorithm for building a global block. We start by scanning through all instructions in the function of interest (FOI) and looking for instructions that call other user functions (OUFs). If found, we recurse and run the algorithm for the detected OUF first. Before recursing, an argument map is built which maps the defined function arguments of the OUF to the variables used by the FOI when providing values for these argument. This map, as well as the name of the FOI, is also passed as an input to our algorithm. If no OUF call is found, we read the argument map and the function name passed to this FOI. The map is first used to replace standard function argument names in the FOI with variables from the calling function. Then the calling function name is appended as a prefix to all variable name in the FOI. The All_Blocks data structure of the FOI is then returned to the calling function, which inlines it. This ensures that all variables have unique names and dependencies across function calls are accurately represented in the global

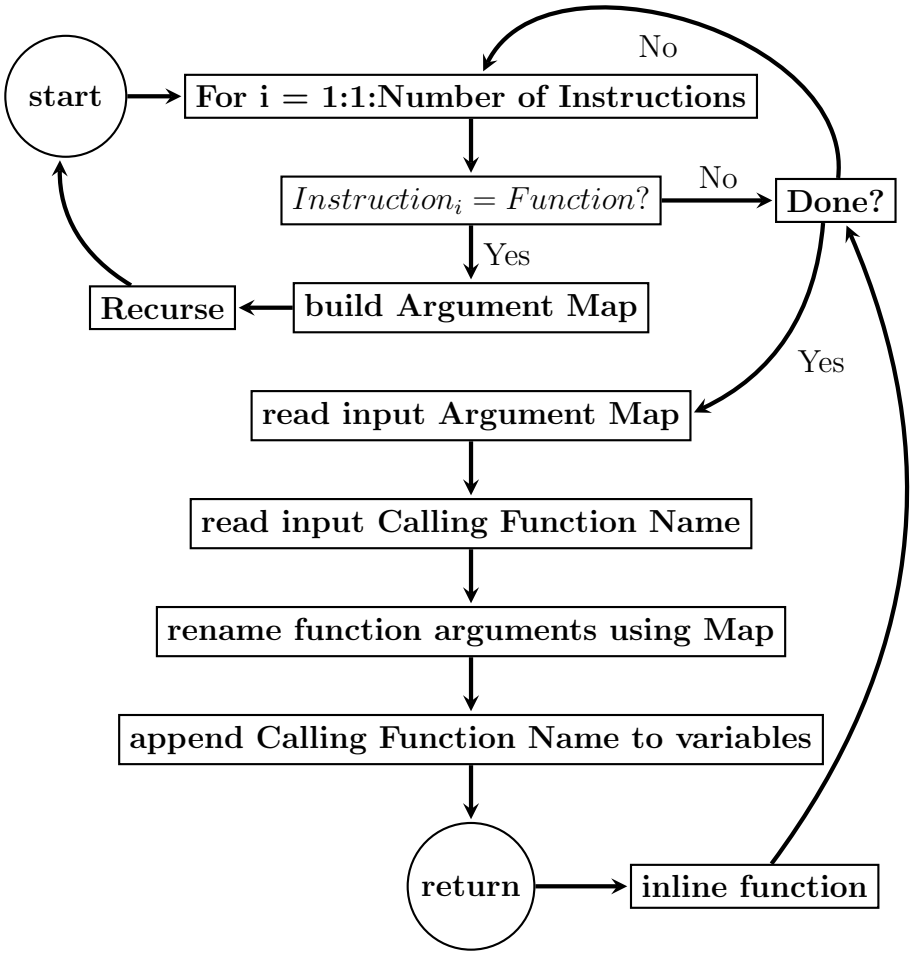


Figure 6·8: Build Global Block

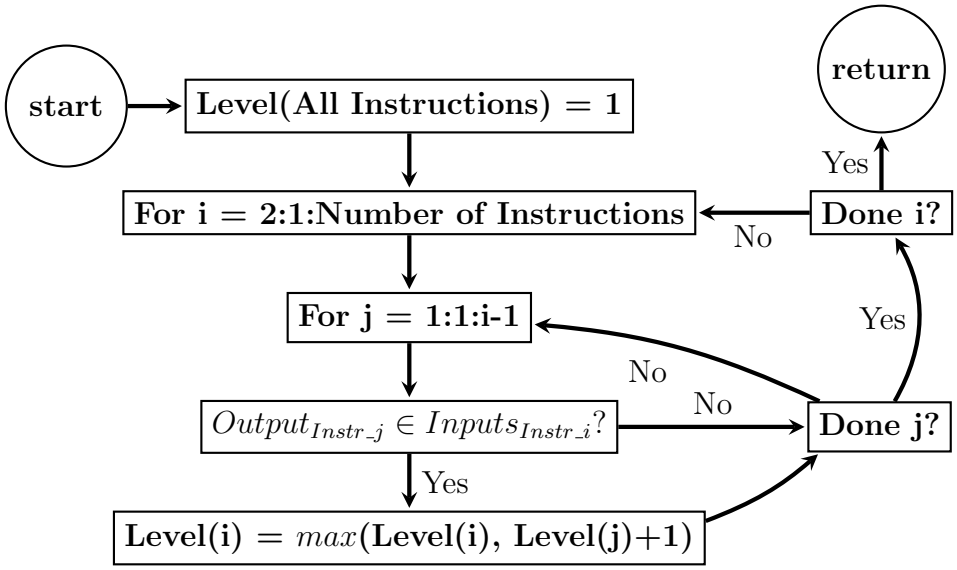


Figure 6·9: Build Pipeline

block.

Figure 6·9 illustrates our algorithm for building a pipeline for the global block. We begin by defining a "Level" array which will contain the pipeline stage number of each instruction. This array is initialized to 1. Starting for the second global block instruction, we scan all instructions prior to it and identify cases where the output of a preceding instruction is an input of the current instruction. For each match, we compare the "Level" value of the preceding instruction with the value of the current instruction, and store the maximum of the two. Once all prior instructions are completed, we move to the next instruction and repeat the process. If an instruction has no dependencies, it is evaluated with a separate set of rules. For example, all "function return" instructions are placed in the last pipeline stage.

Figure 6·10 illustrates our algorithm for performing the concurrency analysis of a function. We begin building the global block and determining the "Levels" data structure for it as described above. The pipeline depth is evaluated as the maximum value in this array. Then, for each instruction in the global block, we check to see if

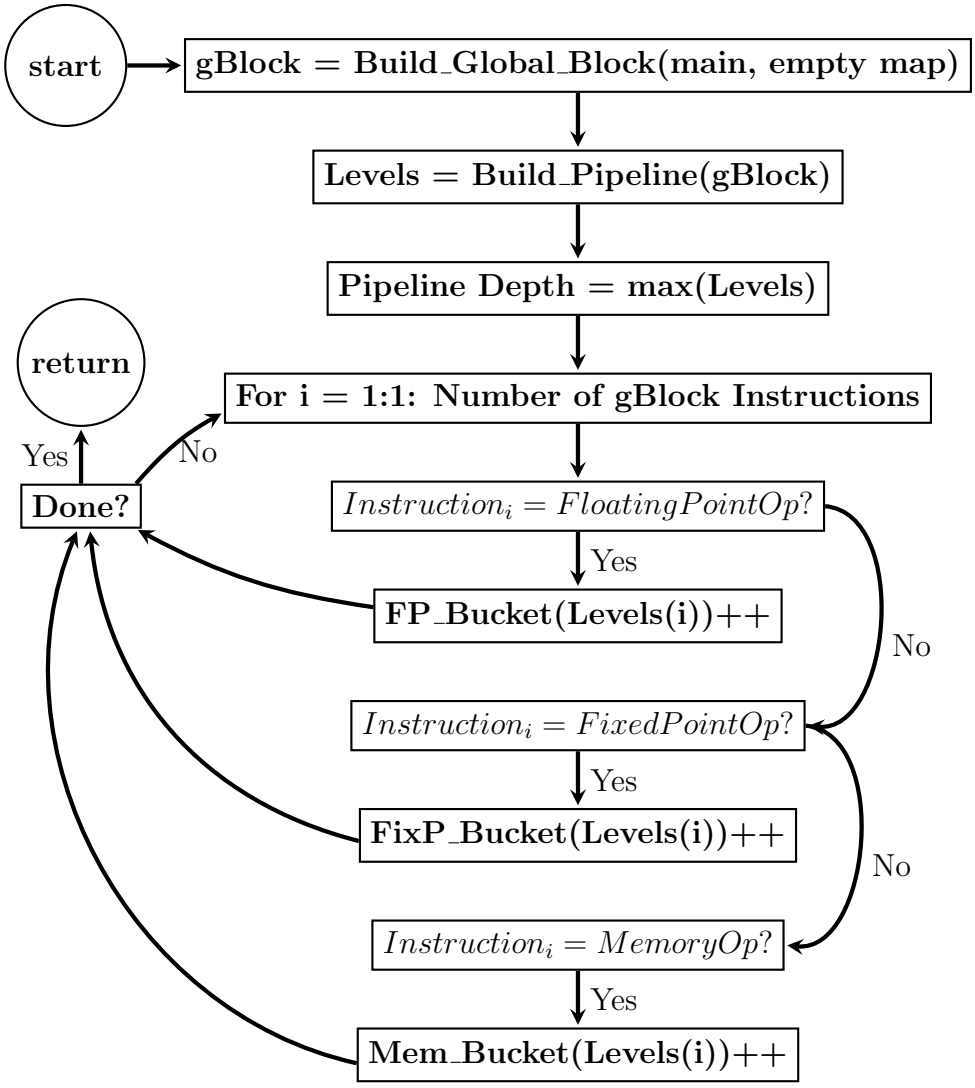


Figure 6·10: Concurrency Analysis

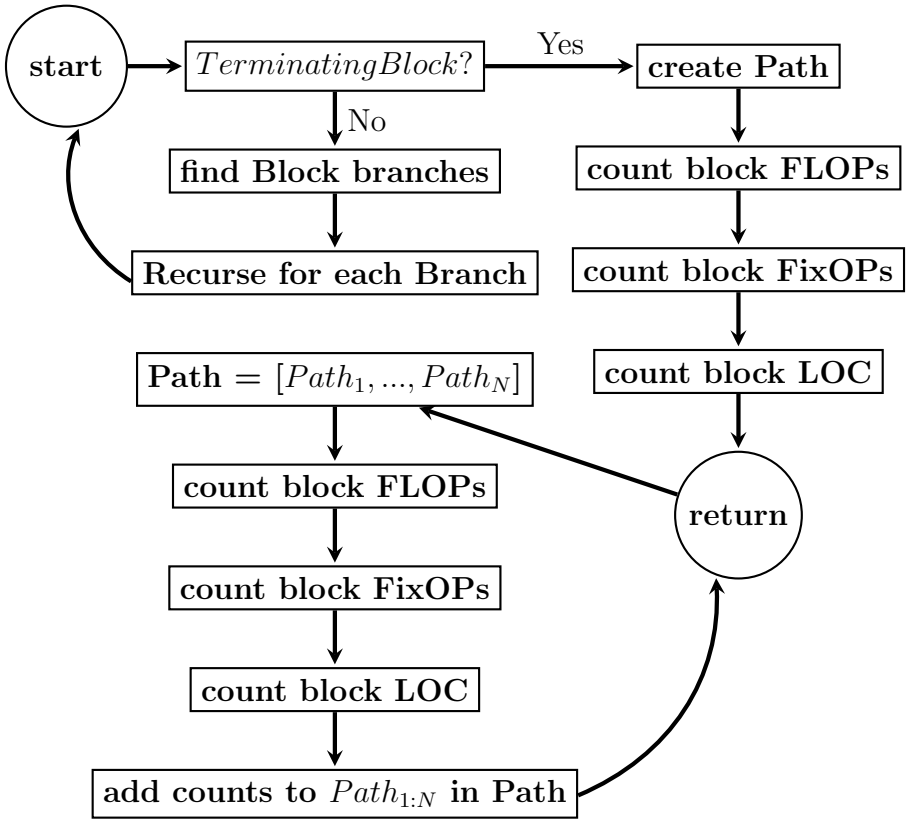


Figure 6·11: Dev Utilization Analysis

it a floating point operation, a fixed point operation or a memory transaction. If any of the three is found, we add a record of this instruction to the appropriate bucket. The specific location of a bucket updated is equal to the value in the "Levels" array corresponding to the instruction. Once the entire global block has been scanned, the algorithm returns all three buckets, as well as the pipeline depth value.

Device Utilization

Figure 6·11 illustrates our algorithm for profiling the unique paths in a function. Unlike other algorithms, we operate on function blocks individually and not the All_Blocks data structure. We begin by analyzing the first block in the function. If it is not the terminating block i.e. does not contain a return instruction, we find all locations where this block can branch to, and recursively run the analysis on those

destination blocks. If a return instruction is found, we create a Path data structure which contains the floating point operations, fixed point operations and lines of code in the block. This Path structure is then returned. If the routine was called by a non-terminating block, then all data returned is merged into a single array, and information for this block computed and added to each element.

Stalls

We analyze three common types of stalls in our profiler: i) BRAM Read-After-Write hazards, ii) floating point accumulation, and iii) off-chip memory accesses.

Figure 6·12 illustrates our algorithm for identifying BRAM based RAW stalls in a function. We first identify all local memories by searching for allocation operations. Then for each memory, we identify all loads and stores corresponding to it. If the variable used to index the memory is the loop iterator, then the operation is considered to be "loop-tied" and discarded. Loop-tied operations are guaranteed to not cause RAW hazard since they always read from or write to a unique location every cycle. If the index is indirectly computed, e.g. by masking bits, then the operation is not considered to be loop-tied since uniqueness for every memory access cannot be guaranteed. For all operations not discarded, we classify them as either definitely causing a hazard (i.e. RAW Known) or potentially causing one (i.e. RAW Warning). The criteria for creating these buckets is: if there is at least one execution path with a load and at least one execution path with a store (to the same memory blocks), and there are no paths without either a load or store operation to the memory block, then it is a known hazard. If there exists at least one path in a loop where there are no loads and stores, or if all memory transactions on all paths are of the same type, then a warning is generated instead.

Figure 6·13 illustrates our algorithm for identifying floating point accumulation hazards. We begin by identifying all floating point addition operations in a function.

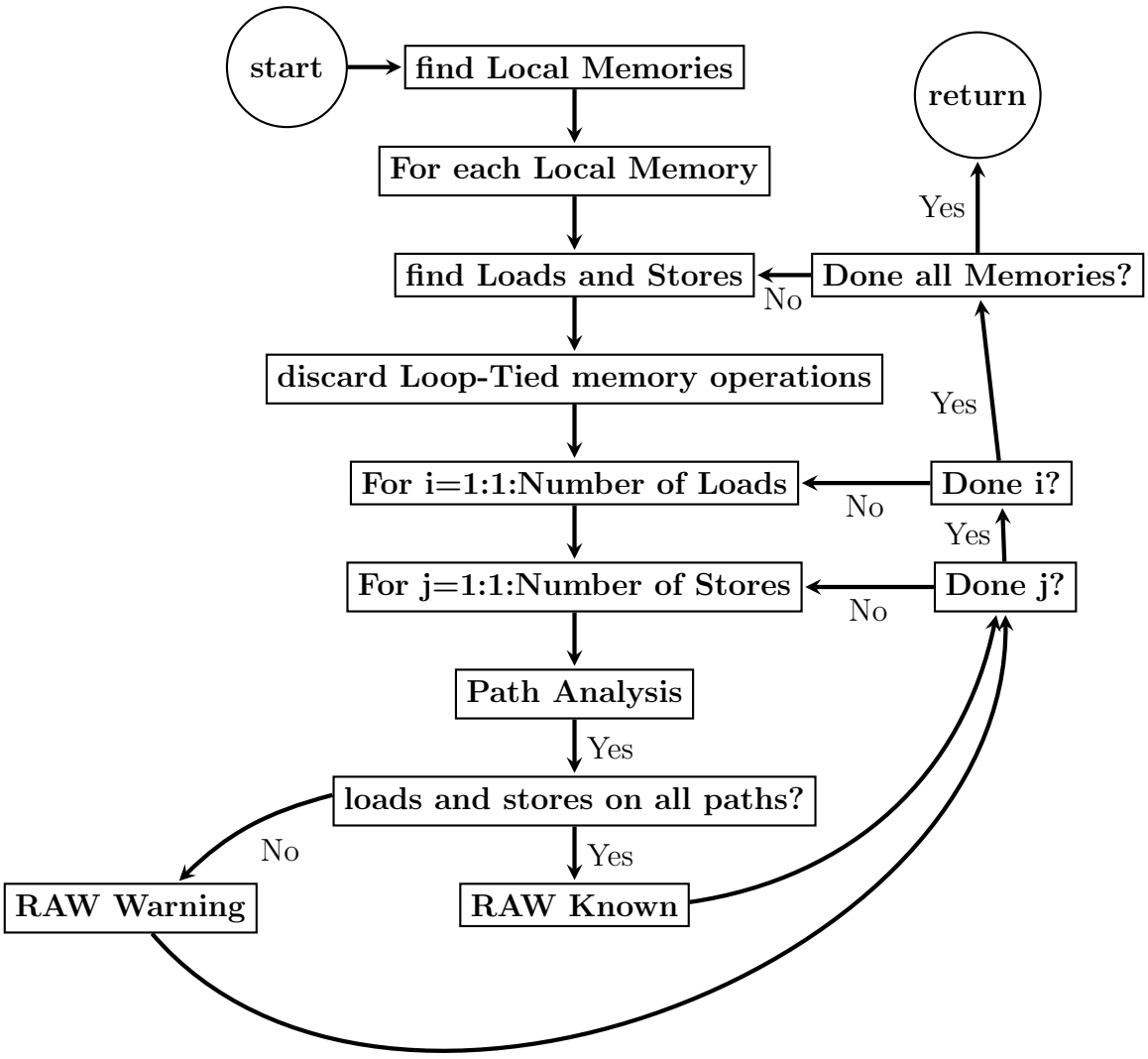


Figure 6·12: BRAM RAW Stall Analysis

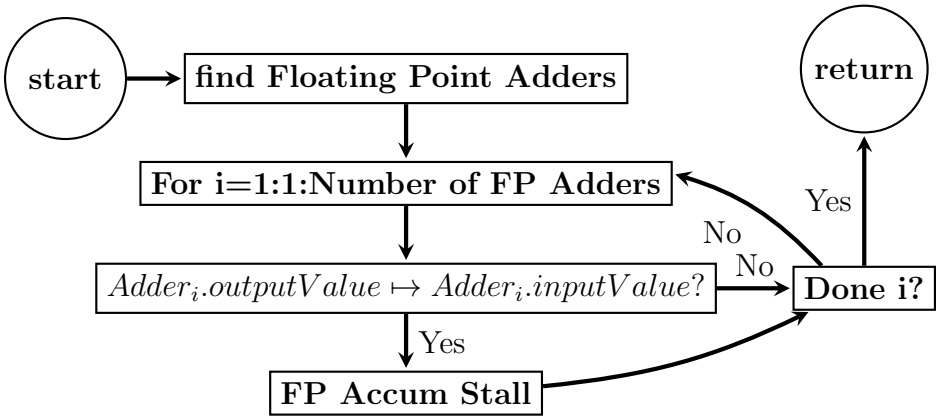


Figure 6·13: FP Accum Stall Analysis

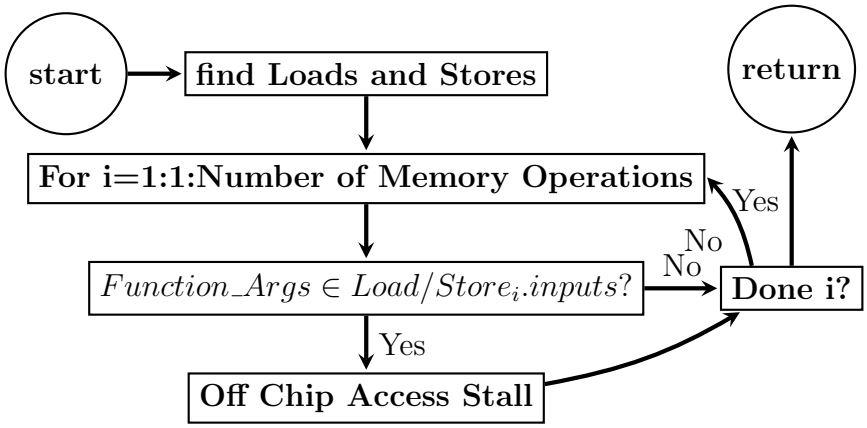


Figure 6·14: Off Chip Memory Stall Analysis

Then, we trace the outputs of each adder to check if they map to one of its inputs. If this is true, then it is recorded as a hazard.

Figure 6·14 illustrates our algorithm for detecting off chip access stalls in a function. This simply involves scanning instructions to find memory operations, and then determining potential off-chip accesses by identifying which indexes are derived from pointers in the function's arguments.

Loop Efficiency

The goal of loop profiling is to identify all loops in a given function, store corresponding loop parameters, organize these loops into hierarchies and then assign stalls,

detected using algorithms discussed above, to each loop. Similar to concurrency analysis, we have three routines for implementing loop profiling.

Figure 6·15 shows the overall algorithm for profiling loops in a function. To simplify the analysis, We assume that user function calls, if any, do not have any loops in them. We begin by searching the All_Blocks structure for all instructions containing keywords corresponding to loop iterator. This indicates the number of loops that must be analyzed. We also initialize an All_Loops array to keep a record of each loop's parameters. These parameters include: i) start line, ii) end line, trip count, iii) array of children (in the case of nested loops), and iv) stall value. Of these, (iii) and (iv) cannot be determined at this stage and are thus left empty. Once the All_Loops structure has been populated with every loop, we then call two routines called Sort_Loops and Assign_Stalls. The first routine takes in the array of unsorted loops and returns a hierarchical data structure by placing loops within the children array of their immediate out loop. For example, given the set of nested loops (i, j, k) such that j is nested in i and k is nested in j, the input to Sort_Loops will be the array $[ijk]$ and its output will be the array $[i]$, where $i.children(1) = j$ and $j.children(1) = k$. The second routine then finds stalls that are unique to each loop (i.e. not shared by its children or siblings) and determines the effective stall seen by loops.

Figure 6·16 illustrates our algorithm for the Sort_Loops routine. We begin by initializing an array of empty Parent loops. If an empty set of unsorted loops is passed as an input to the routine, the Parent array is returned (an empty Children array indicates that it is an innermost loop). Otherwise, we scan the unsorted loops to find all Parent loops i.e. which are not contained in any other loop. Then, for each Parent, we create an array of all loops that are contained in it. This unsorted array is then recursively passed to the Sort_Loop routine and the result stored within the

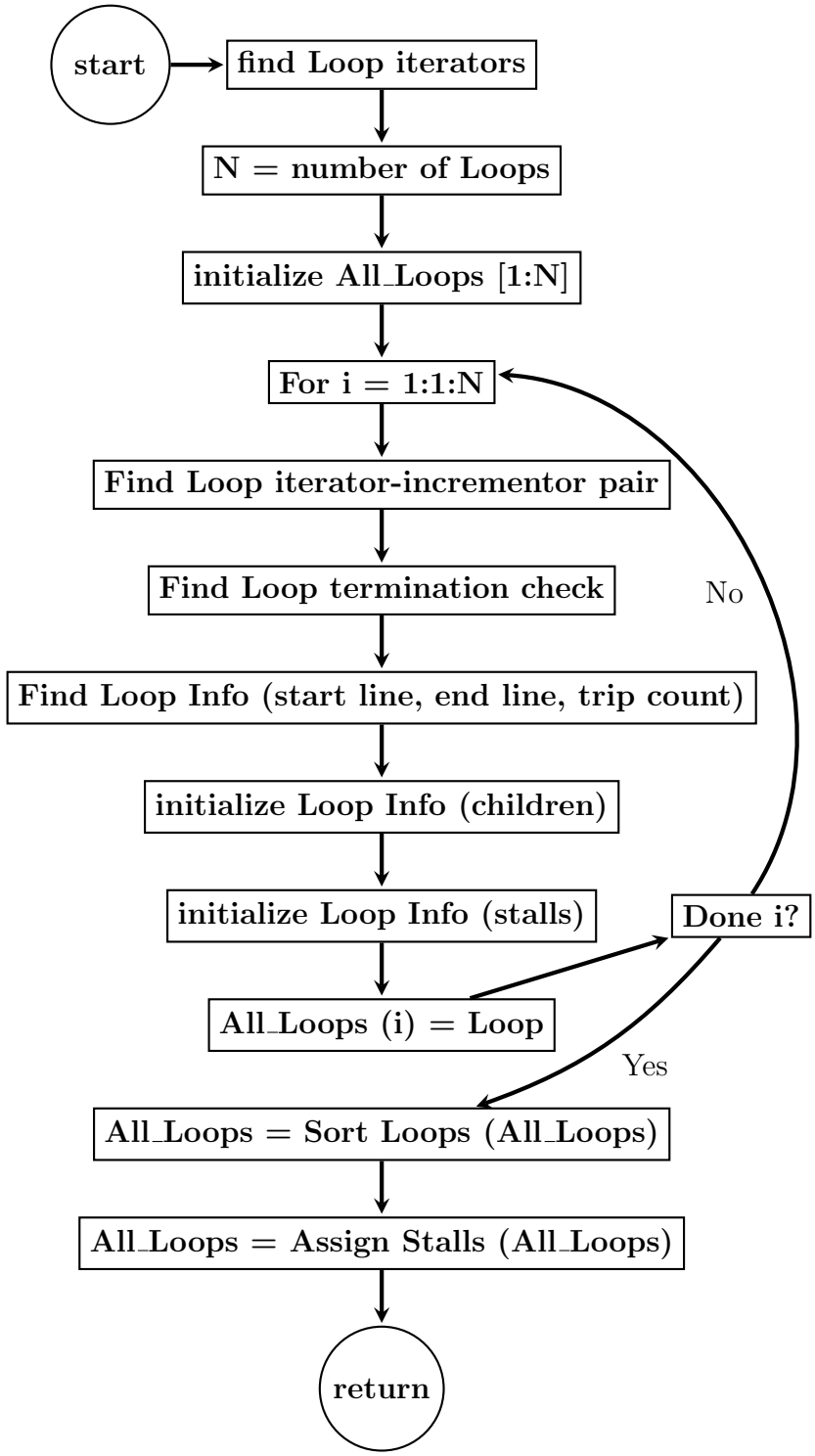


Figure 6·15: Loop Analysis

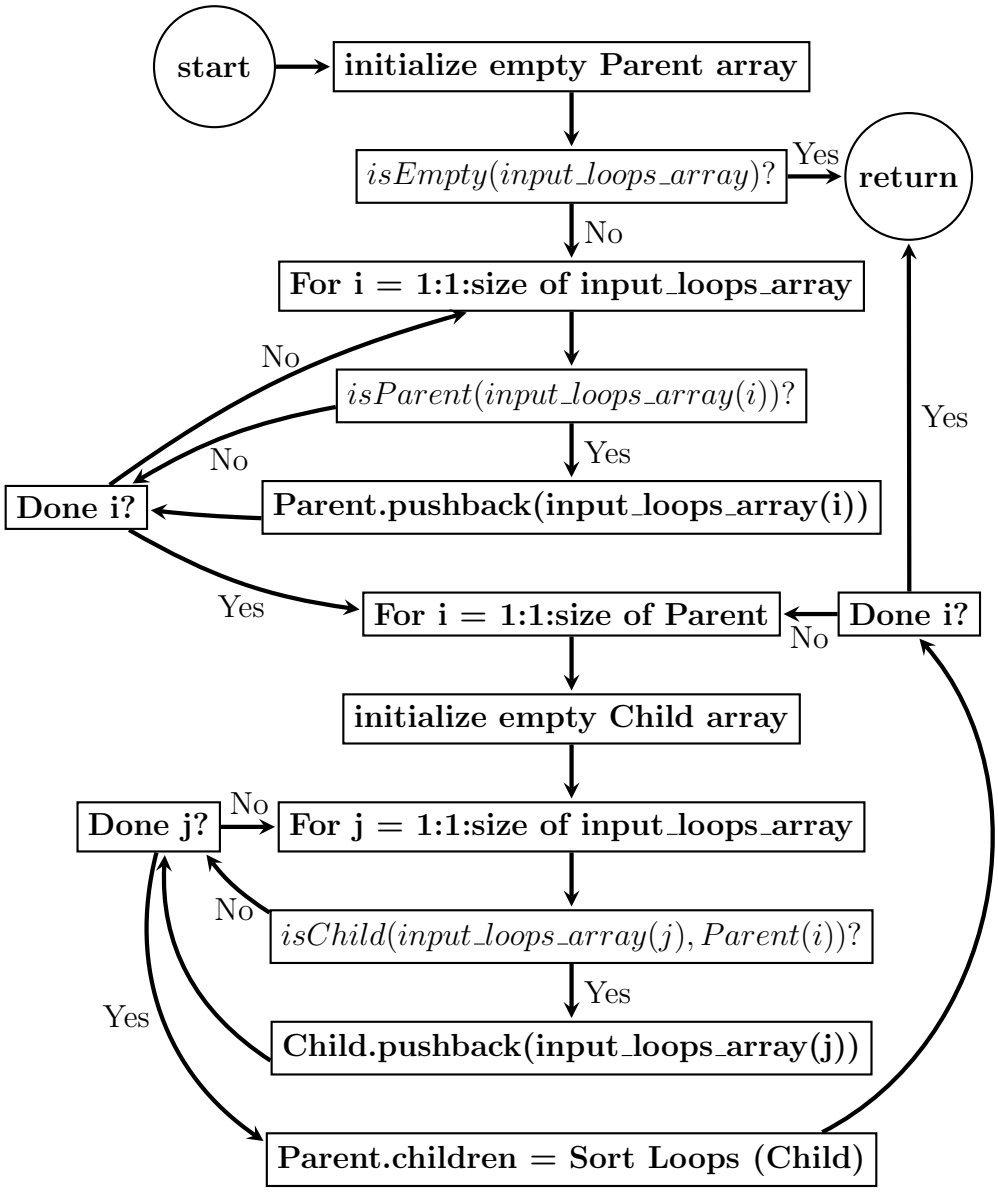


Figure 6·16: Sort Loop Children

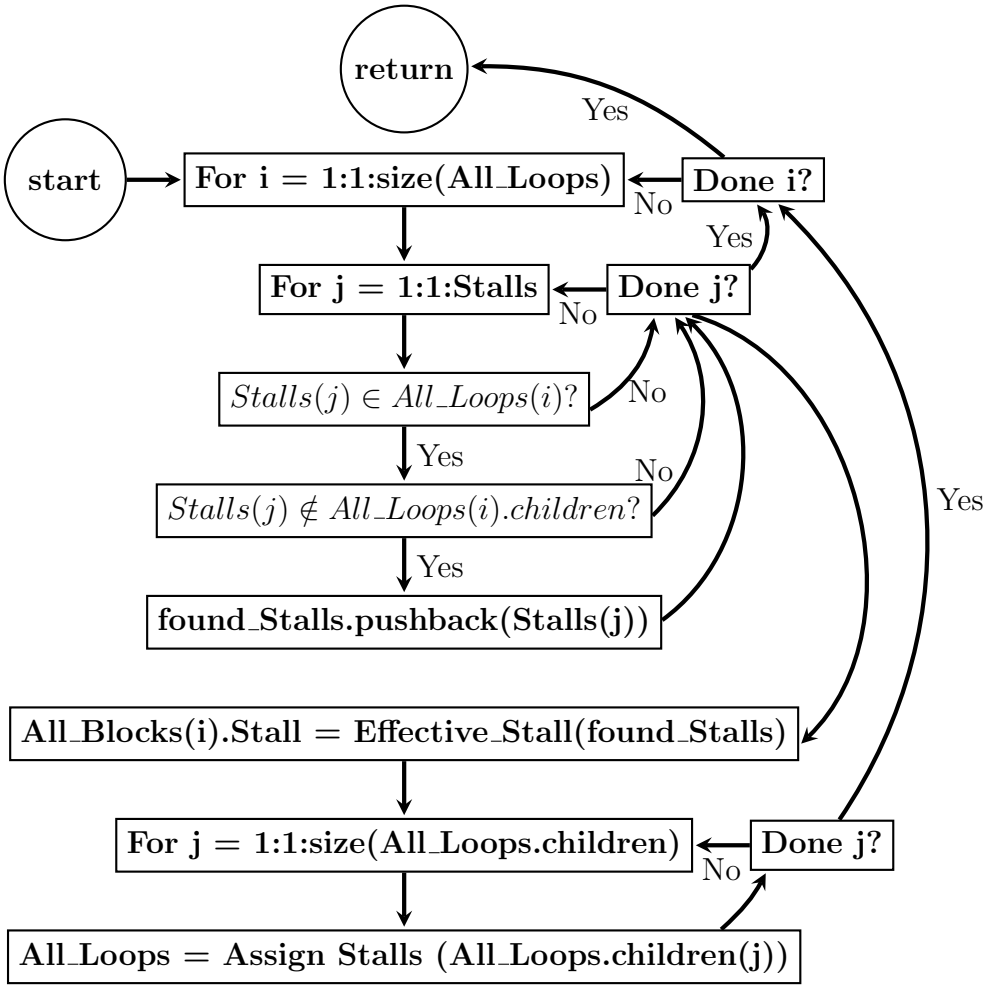


Figure 6·17: Sort Loop Stalls

Parent's structure.

Figure 6·17 illustrates our algorithm for assigning stalls to loops. Given an All_Loops array that has been sorted based on hierarchies, we identify all stalls that are found only within each loop and not its children. Then, based on the magnitude and type of each stall, we compute the effective stall cycles for the loop. To reduce the complexity of this analysis, we assume that stalls from sibling loops are zero. Finally, for each Child or each element in the All_Loops array, we recursively call the Assign_Stalls routine again.

Local Memory Efficiency

The goal of profiling local memory efficiency is to determine the size of memory needed to implement a given function. This size is a function of the allocation size requested when the memory is declared, and the number of times this memory has to be replicated to support stall-free operation.

Figure 6·18 illustrates our algorithm for analysing local memories in a function. We begin by creating an array of all memories declared inside a function. Each element of this array contains the requested memory size and a copy of instructions which access this memory for loads and stores. This array is then passed onto the recursive Loop_Mem_Analysis routine for determining the replication amount. Figure 6·19 illustrates the algorithm for this routine. The computation done here is in the context of loops i.e. if a memory is used in different loops, then accesses to the memory from different instructions may not be concurrent. Similar to loop stall above, we make the simplifying assumption that loop siblings do not affect each other's analysis results. Moreover, we also assume that if a memory is replicated due to a Parent loop, then the additional memory is not shared by any other loop (sibling or child). For each memory, we initialize counters for loads and stores. Then we scan uses of this memory to determine if they are exclusive to the Parent loop, and update counters appropriately if true. After all applicable uses have been evaluated, we compute the replication amount based on the final values of loads and stores. The Loop_Mem_Analysis routine is then called recursively to determine replication contributions by the Child loops.

Global Memory Efficiency

The algorithm for global memory follows the same approach as local memories. The only difference is that we analyze memories declared outside all functions (which

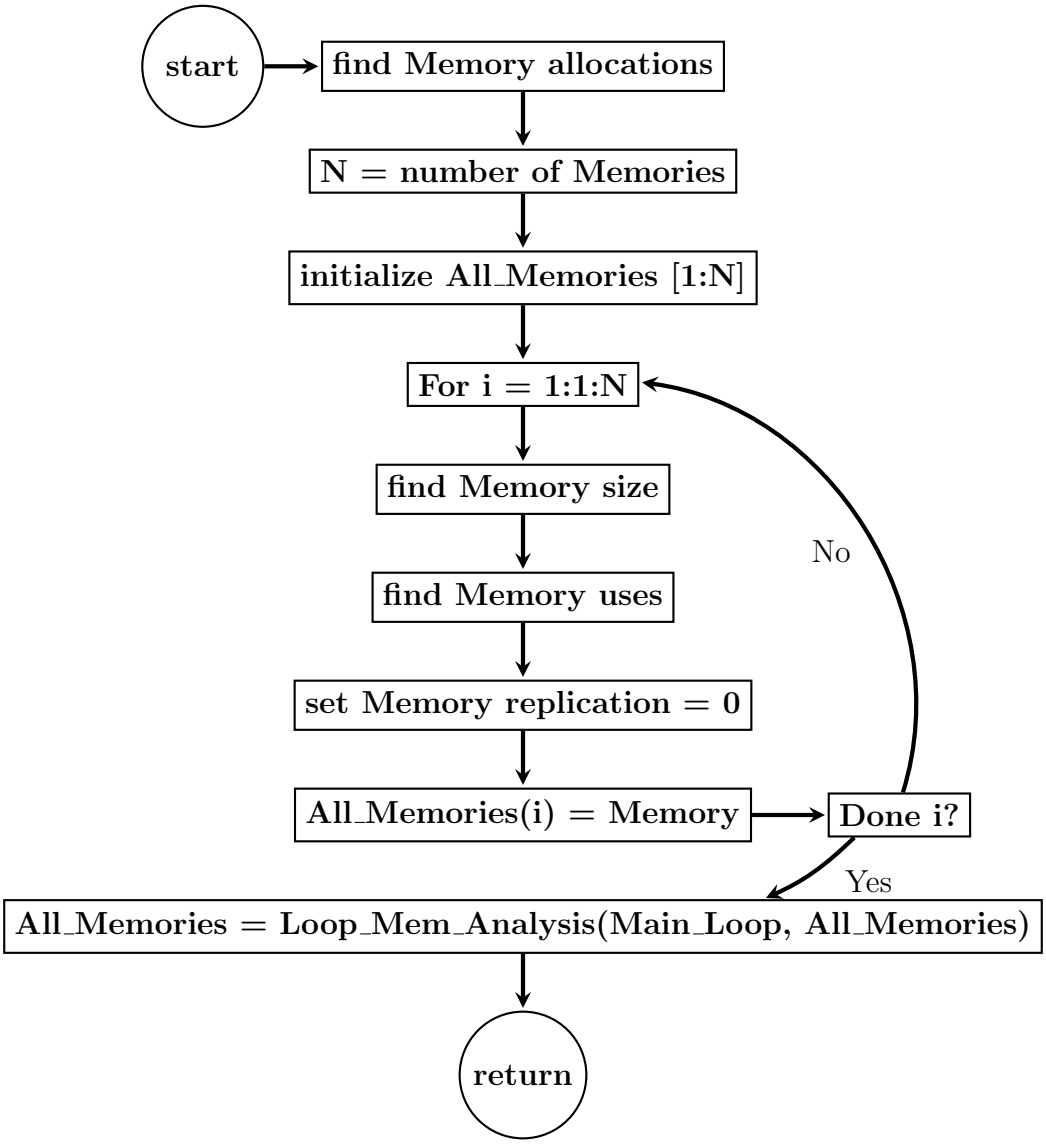


Figure 6·18: Local Memory Efficiency

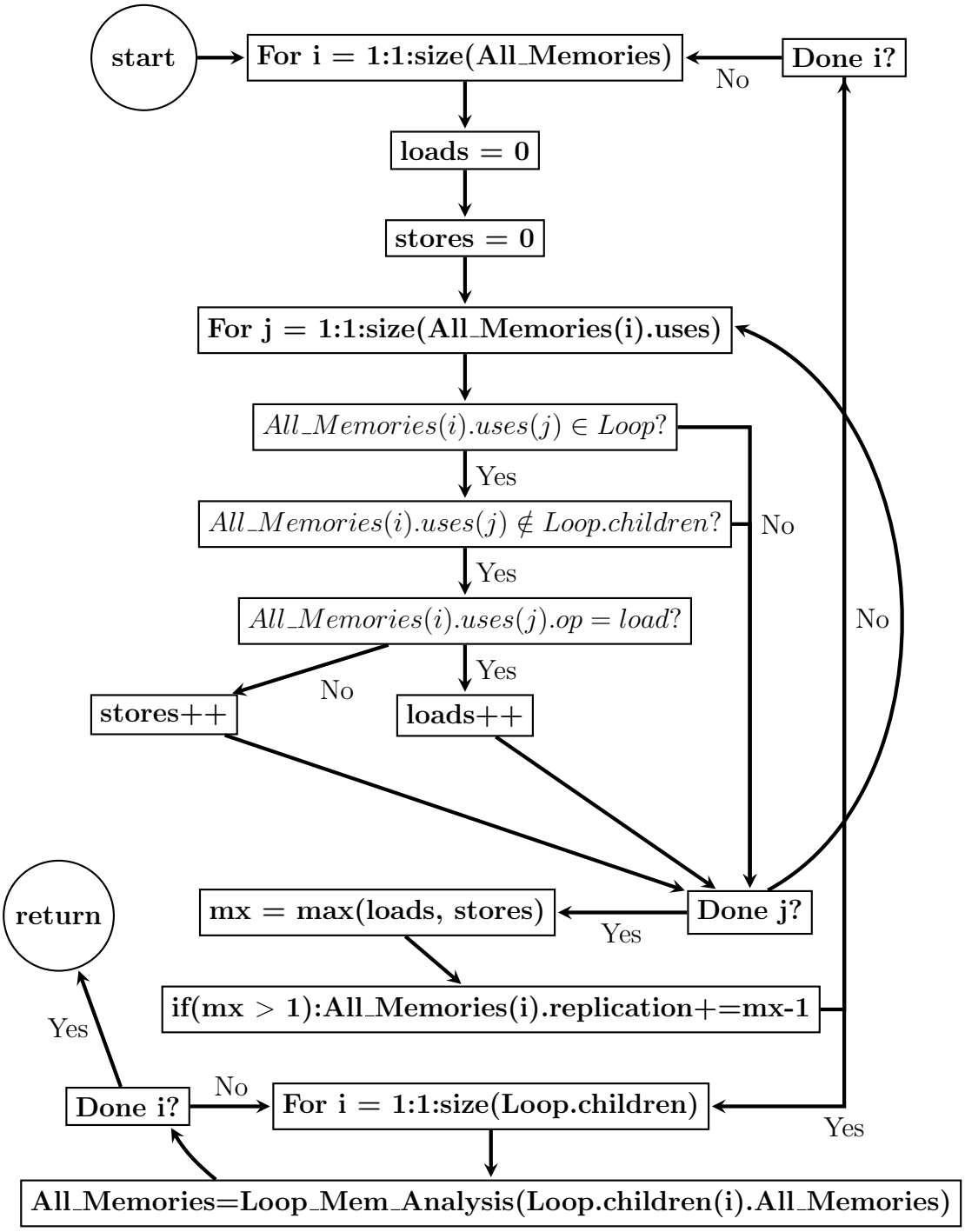


Figure 6·19: Loop Memory Analysis

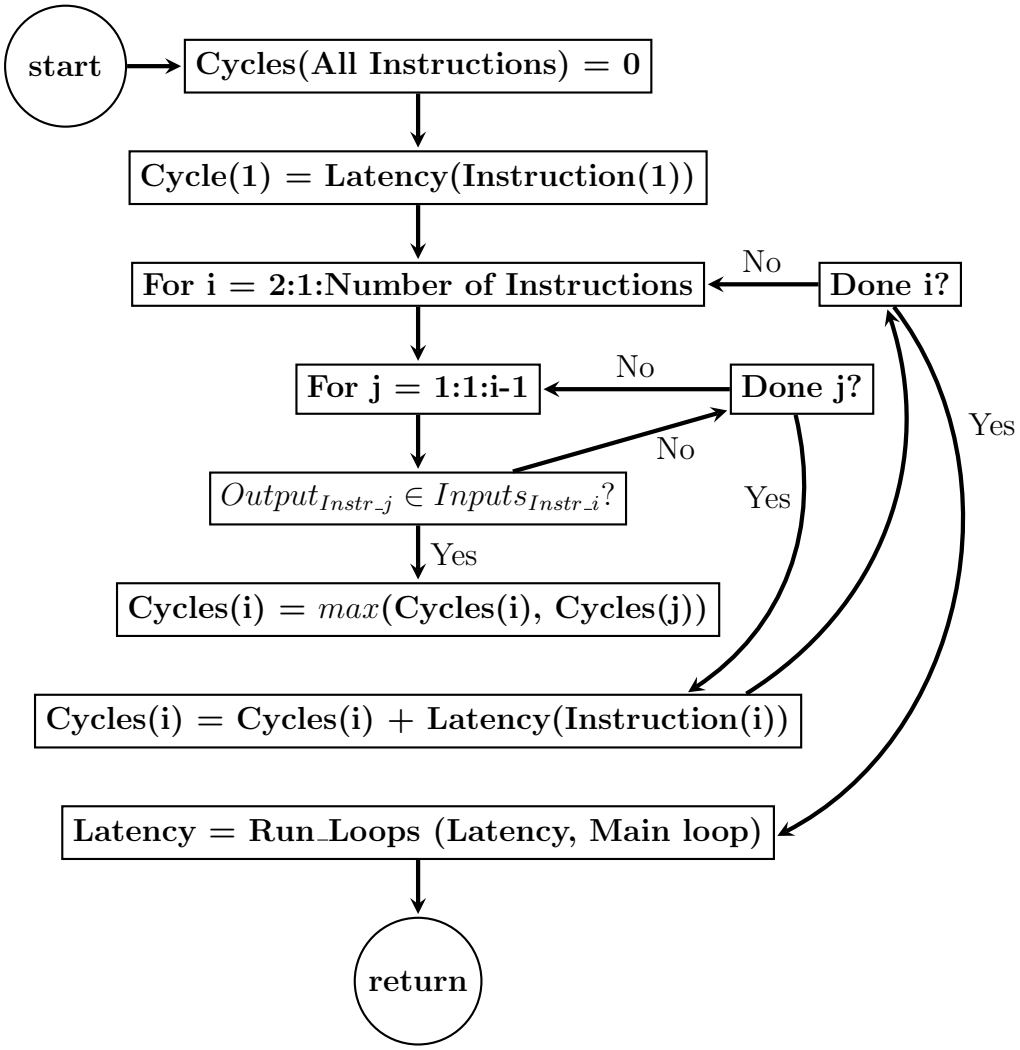


Figure 6·20: Run Time

makes them globally available to every instruction).

Run Time

Finally, Figure 6·20 illustrates our algorithm for estimating function runtime. This is similar to the Build_Pipeline algorithm discussed earlier with two exceptions. First, instead of storing pipeline stage numbers, we store the effective number of cycles taken before an instruction is executed. This is estimated as the largest cycle count for computing one of the instruction's inputs, plus the latency of the instruction's

operation itself. Second, once the "Cycle" array has been generated, we call the Run_Loops routine which recursively emulates the execution of each loop by using trip count values, cycle value of first instruction, cycle value of last instruction, loop stalls, and results of nested loop executions. This routine returns an updated array of cycle counts, with the runtime being the maximum value in this array if the routine was called for the Main loop.

6.7 Probes

Given a profiler, we now propose a set of probes that are aimed at profiling each spatio-temporal model element with a high degree of isolation. By running each probe through a compiler and profiling the resulting IR, we can identify if the corresponding design pattern is being blocked. Below, we provide details of these probes, which are small code fragments written as C functions. To validate the effectiveness of our probes, we compile OpenCL code (without any FPGA specific semantics such as unroll pragmas or channels) into Static Single Assignment (SSA) IR using the Clang front end and "-03" optimization flags. Since only standard code optimizations are applied, we expect that FPGA specific optimizations will be blocked. Custom stall values, unless otherwise stated, are 100, 4 and 1 for off-chip access, floating point accumulation and BRAM RAW hazard respectively.

6.7.1 SIMD

Algorithm 16 gives our probe function for evaluating SIMD. It is a simple vector add loop, with no dependency between iterations, and restrict keywords (to indicate that memory spaces do not overlap). Thus, a compiler should ideally be able to perform the additions in parallel. Based on the relevant profiler results shown in Figure 6·21, we observe that the compiler is blocked from unrolling the loop since there is only 1 floating point operation per cycle.

Algorithm 16 Probe Function for SIMD

```
kernel void vecAdd(global float* restrict const A, global float* restrict const B, global
float* restrict C){
        for (int i = 0 ; i ¡ SIZE; i++)
                C[i] = A[i] + B[i];
}
```

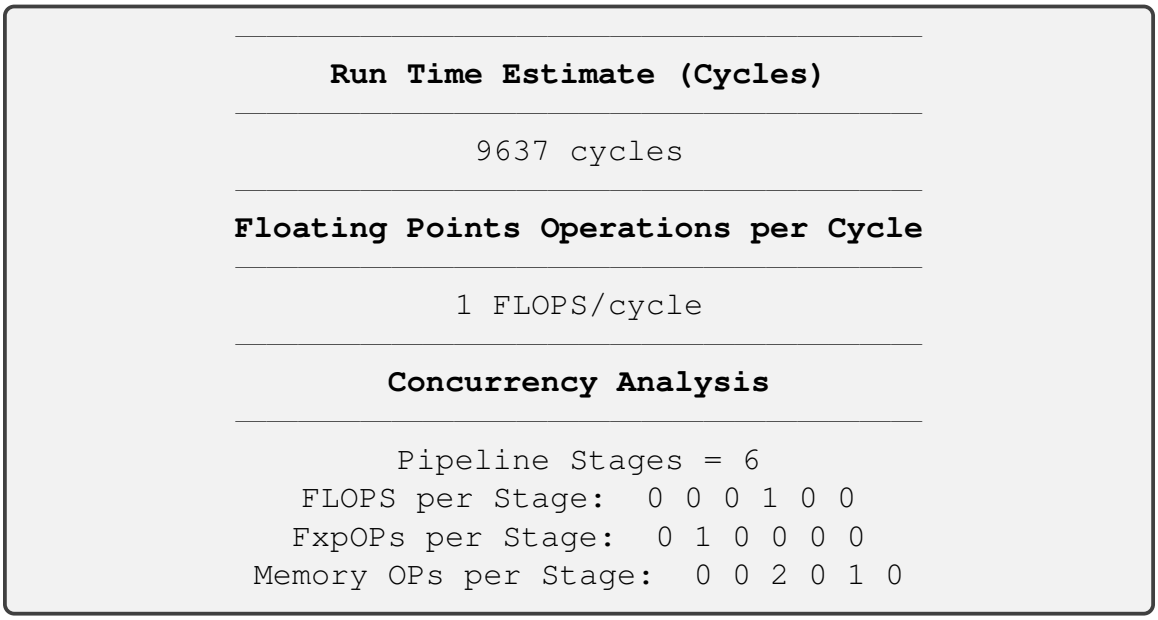


Figure 6·21: Relevant profiler results for the SIMD probe showing a blocked optimization

6.7.2 Pipelining

Algorithm 17 gives our probe function for evaluating Pipelining. It is a simple systolic array implementation, expressed as two nested loops. The inner loop computes results for each systolic array element based on the results of the preceding ones, while the outer loop iterates over the external inputs to the array. Since there is a loop carried dependency in the inner loop, and since it is nested within an outer loop, a compiler should ideally be able to: i) infer instruction level parallelism and replace this inner loop with a pipeline, and ii) implement the integer arrays "systol" and "systol_1" as a set of individual variables, as opposed to a single memory block. The latter

corresponds to inferring pipeline registers as registers. Based on the relevant profiler results shown in Figure 6·22, we observe that the compiler is blocked from creating the loop and does not infer the integer arrays as individual variables.

Algorithm 17 Probe Function for Pipelining

```
int maval (int a, int b, int c){
        if ((a >= b) && (a >= c))
                return a;
        if ((b >= a) && (b >= c))
                return b;
        return c;
}

kernel void ram_array(global int* restrict const A, global int* restrict C){
        int systol[SIZE];
        int systol_1[SIZE];
        for (int i = 0; i < SIZE; i++){
                systol[i] = 0;
                systol_1[i] = 0;
        }
        for (int i = 0; i < ITERS; i++){
                for (int j = 0; j < SIZE; j++){
                        systol_1[j] = systol[j];
                        if (j == 0)
                                systol[j] = maval(0 , A[i] , systol_1[j]+1);
                        else
                                systol[j] = maval(systol_1[j-1] , systol[j-1] , systol_1[j]+1);
                }
                C[i] = systol[SIZE-1];
        }
 }
```

6.7.3 Caching

Algorithm 18 gives our probe function for evaluating caching. It is a simple integer matrix multiplication implementation which reads data from off-chip memory, processes it, and then stores it back off-chip. A compiler should ideally be able to implement caches for inputs and outputs in order to coalesce memory accesses, and prevent off-chip access of individual variable in the critical path of the computation.

```
Run Time Estimate (Cycles)

2680750080 cycles

Local Memory Analysis

 Name:  systol
Size:  512 bits
Replication:  1
   Uses:  5

Name:  systol_1
Size:  512 bits
Replication:  0
   Uses:  3
```

Figure 6·22: Relevant profiler results for the Pipelining probe showing blocked optimizations i.e. unable to infer i) pipelines and ii) pipeline registers as registers

Based on the relevant profiler results shown in Figure 6·23, we observe that the compiler is blocked from inferring these caches, since the innermost loops both have 200 cycle stalls due to off-chip access.

6.7.4 Constants

Algorithm 19 gives our probe function for evaluating the implementation efficiency of constant arrays. It is similar to the systolic array code in Algorithm 17 above, but with two key differences. First, we replace the inner loop with equivalent manually written code. Second, we modify the computation to read from a constant array when evaluating the new value of a systolic array element (as opposed to the "+1" done previously). The memory space accessed does not overlap, and hence a compiler should ideally split the single constant array into multiple smaller ones. Based on the relevant profiler results shown in Figure 6·24, we observe that the compiler is blocked from doing so, and instead replicates the constant array.

Algorithm 18 Probe Function for Caching

```
kernel void no_cache(global int* restrict const A, global int* restrict const B, global int*
restrict C){
        for (int i = 0 ; i < SIZE; i++){
                for (int j = 0 ; j < SIZE; j++){
                        C[i*SIZE + j] = 0;
                        for (int k = 0; k ¡ SIZE; k++){
                                C[i*SIZE + j] += A[i*SIZE + k] * B[k*SIZE + j];
                        }
                }
        }
}
```

Algorithm 19 Probe Function for Constants

```
int maval (int a, int b, int c);

constant int lookup [16*ITERS] = ....

kernel void single_const_array(global int* restrict const A, global int* restrict C){
        int systol_0 = 0; int systol_1 = 0; ... int systol_SIZE-1=0;
        int systol_1_0 = 0; int systol_1_1 = 0; ... int systol_1_SIZE-1=0;
        for (int i = 0; i < ITERS; i++){
                systol_1_0 = systol_0;
                systol_0 = maval(0, A[i], systol_1_0 + lookup [ITERS*0 + i]);
                systol_1_1 = systol_1;
                systol_1 = maval(systol_1_0, systol_0, systol_1_1 + lookup [ITERS*1 + i]);
                                                ⋮
                systol_1_SIZE-1 = systol_SIZE-1;
                systol_SIZE-1 = maval(systol_1_SIZE-2, systol_SIZE-2, systol_1_SIZE-1 +
                lookup [ITERS*(SIZE-1) + i]);
                C[i] = systol_SIZE-1;
        }
 }
```

Figure 6·23: Relevant profiler results for the Caching probe showing that the compiler is blocked from inferring caches, and thus causes off-chip access stalls in the innermost loops.

6.7.5 Inlining

Algorithm 20 gives our probe function for evaluating the inlining efficiency. It is similar to the systolic array code in Algorithm 19 above, but without the use of

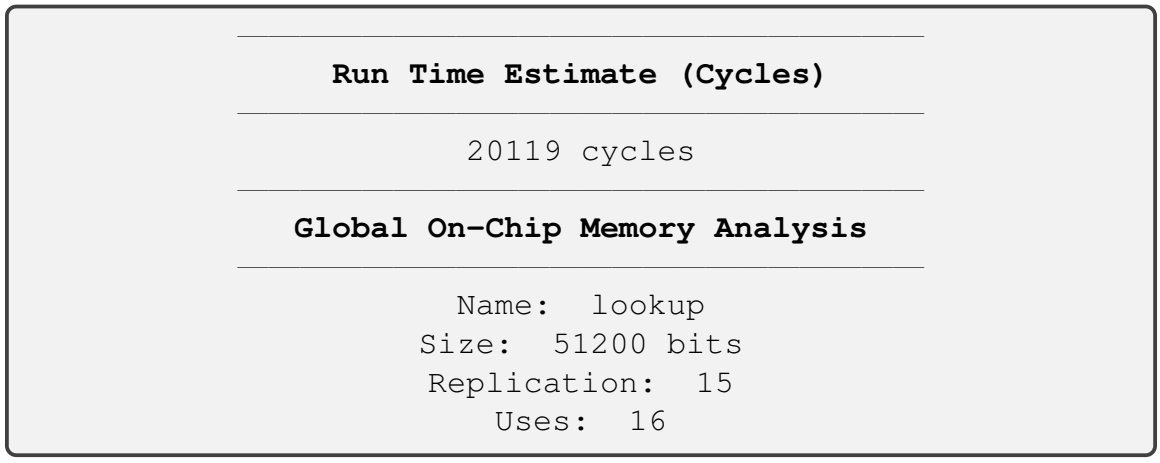


Figure 6·24: Relevant profiler results for the Constants probe showing that the compiler is blocked from breaking the larger constant array into smaller ones, and instead would need to replicate the memory

a constant array. Based on the relevant profiler results shown in Figure 6·25, we observe that the compiler is able to inline the "maval" function, but is blocked from performing a tight integration of pipelines. As a result, there are a very large number of paths generated as the code explicitly branches between main function and "maval" instructions.

6.7.6 Loops

We reuse the code in Algorithm 18 for evaluating loop implementation efficiency. As shown by results in Figure 6·23, the compiler is blocked from removing nested loops, and hence the outer loops cannot be pipelined.

6.7.7 BRAM Read-After-Write Hazard

Algorithm 21 gives our probe function for evaluating compiler effectiveness in resolving RAW hazards for on-chip memory. It is a simple hash table implementation, which computes a key as a function of table size and the value of A[i]. Based on the value of B[i], we either perform a store or load transaction on the hash table. This results in a potential RAW hazard since: i) the hash table is not indexed directly using a

Algorithm 20 Probe Function for Inlining

```
int maval (int a, int b, int c);

kernel void reg_array_func(global int* restrict const A, global int* restrict C){
        int systol_0 = 0; int systol_1 = 0; ... int systol_SIZE-1=0;
        int systol_1_0 = 0; int systol_1_1 = 0; ... int systol_1_SIZE-1=0;
        for (int i = 0; i < ITERS; i++){
                systol_1_0 = systol_0;
                systol_0 = maval(0, A[i], systol_1_0 + 1);
                systol_1_1 = systol_1;
                systol_1 = maval(systol_1_0, systol_0, systol_1_1 + 1);
                                                ⋮
                systol_1_SIZE-1 = systol_SIZE-1;
                systol_SIZE-1 = maval(systol_1_SIZE-2, systol_SIZE-2, systol_1_SIZE-1 +
                1);
                C[i] = systol_SIZE-1;
        }
}
```

loop iterator, and is hence not loop-tied, and ii) the value of B[i] is non-deterministic, which could result in a case where table[x] = C[y] is followed by C[y+1] = table[x] (reading a table location immediately after it is written to). The compiler should ideally infer this potential hazard, and implement hardware such that the loop does not stall. Based on the relevant profiler results shown in Figure 6·26, we observe that the compiler is blocked from doing so. This means that each path either does a load or a store, and therefore doing a read after write requires stalling to ensure that the write is complete.

6.7.8 Execution Paths

Algorithm 22 gives our probe function for evaluating compiler effectiveness in reducing the number of execution paths. It is a variant on the vector add function, where the actual addition done is based on the loop iteration number and the value of A[i]. Since the code does not contain branches, a compiler should ideally be able to separate the conditional assignment operations, and cascade them in a manner that generates a

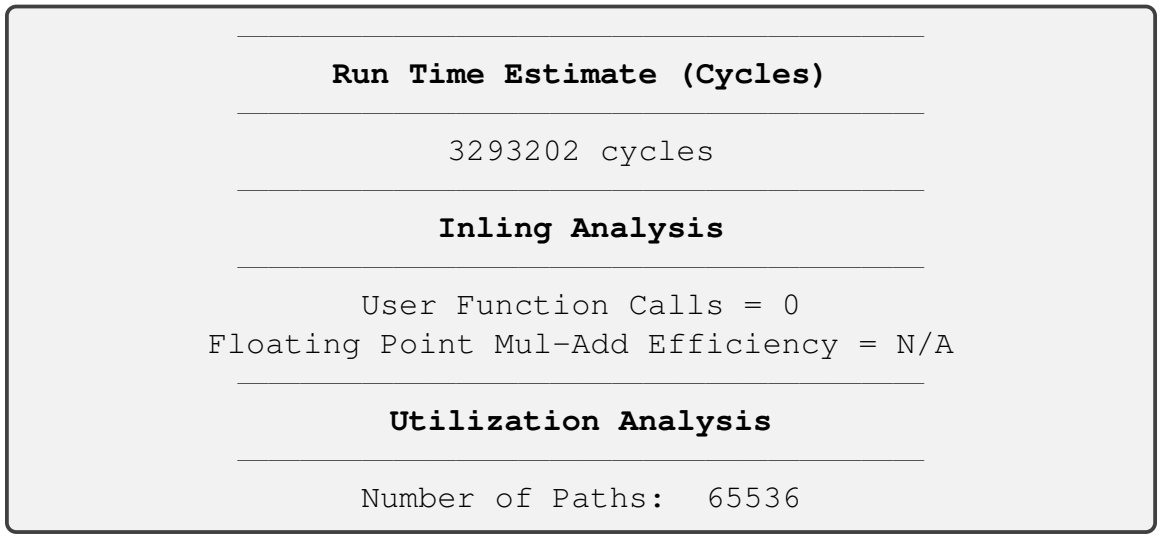


Figure 6·25: Relevant profiler results for the Constants probe showing that compiler is: i) able to replace the function called with function body, and ii) blocked from tightly integrating the function body into the main pipeline

single path. However, based on the relevant profiler results shown in Figure 6·27, we observe that the compiler is blocked from doing so since 3 paths are generated, each corresponding to evaluating "in1", "in2" and "in3" respectively.

6.7.9 Input Multiplexing

Algorithm 23 gives our probe function for evaluating compiler effectiveness inferring input multiplexing. It is similar to Algorithm 22, except that conditional branches are used here instead of conditional assignments. A compiler should ideally be able to infer that the effective operation is a single floating point addition, and creates paths such that only one adder is required. However, based on the relevant profiler results shown in Figure 6·28, we observe that the compiler is blocked from doing so and instead generates three different adders (3 FLOPs/cycle).

Algorithm 21 Probe Function for BRAM based Read-After-Write Hazard

```
kernel void raw_hazard(global int* restrict const A, global int* restrict const B, global
int* restrict const C){
        int table[SIZE];
        for (int i = 0 ; i < ITERS; i++){
                int hash = ((A[i] >> 5) & (A[i] << 7)) & (SIZE-1);
                if (B[i] == 0){
                        table[hash] = C[i];
                }
                else {
                        C[i] = table[hash];
                }
        }
}
```

Algorithm 22 Probe Function for Execution Paths

```
kernel void multi_paths(global float* restrict const A , global float* restrict const B,global
float* restrict C){
        for (int i = 0 ; i < SIZE; i++){
                float in1 = A[i] + B[i];
                float in2 = A[i] + SIZE;
                float in3 = B[i] + SIZE;
                float in4 = C[i];
                C[i] = (i>0) ? in1 : ((A[i] < SIZE) ? in2 : ((A[i] == SIZE) ? in3 : in4 ));
        }
}
```

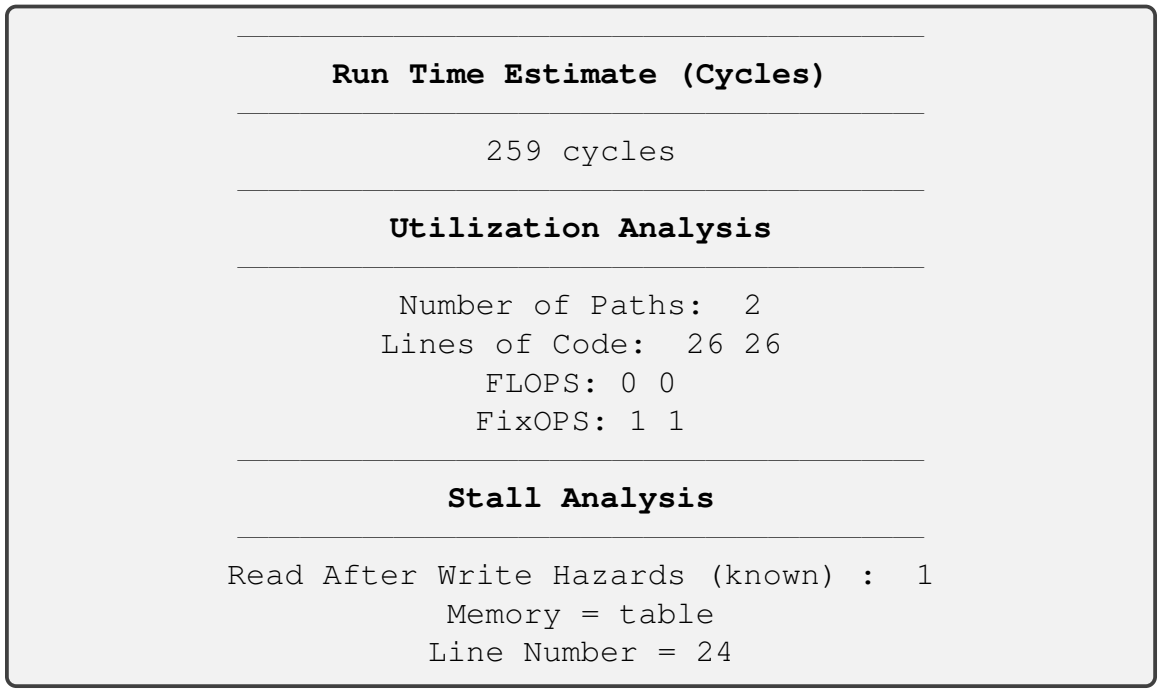


Figure 6·26: Relevant profiler results for the BRAM RAW probe showing that the compiler is blocked from resolving the read after write hazard

6.7.10 Floating Point Accumulator

We reuse Algorithm 18 here, with the exception that matrices A, B and C contain floating point values. Moreover, we set off-chip stall values to 0 in order to prevent it from masking stalls due to floating point accumulation. As shown by the results in Figure 6·29, the compiler is blocked from removing the stall due to accumulating multiplication results in the innermost loop.

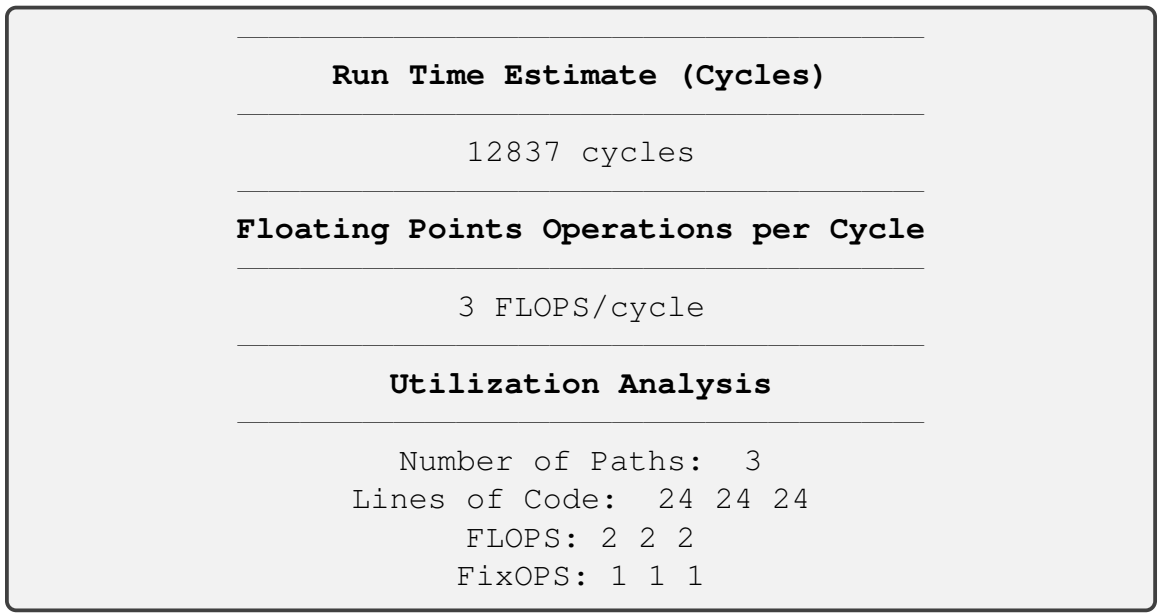


Figure 6·27: Relevant profiler results for the Executions Paths probe showing that the compiler is blocked from implementing the computation as a single path.

Algorithm 23 Probe Function for Input Multiplexing

```
kernel void branch(global float* restrict const A , global float* restrict const B, global float* restrict C){
    for (int i = 0 ; i < SIZE; i++){
        if (i == 0){
            if(A[i] < SIZE){
                C[i] = A[i]+SIZE;
            }
            else if (A[i] == SIZE){
                C[i] = B[i]+SIZE;
            }
        }
        else{
            C[i] = A[i] + B[i];
        }
    }
}
```

```
Run Time Estimate (Cycles)

19237 cycles

Floating Points Operations per Cycle

3 FLOPS/cycle

Concurrency Analysis

Pipeline Stages = 6
FLOPS per Stage:  0 0 0 3 0 0
FxpOPs per Stage:  0 1 0 0 0 0
Memory OPs per Stage:  0 0 3 0 3 0
```

Figure 6·28: Relevant profiler results for the Input Multiplexing probe showing that the compiler is blocked from reducing logic requirements

```
Run Time Estimate (Cycles)

173056 cycles

Stall Analysis

Read After Write Hazards (known) :  0
Read After Write Hazards (warning):  0
   Floating Point Accumulator :  1
     Off Chip Memory Access :  4
```

Figure 6·29: Relevant profiler results for the FP Accum Probe showing that the compiler has detected a floating point accumulator

6.8 Resolving Optimization Blockers

In the previous section, we presented a model for FPGAs which represents efficient design patterns that a compiler must be able to effectively target in order to generate high performance hardware. We then demonstrated, using our static profiler and probe suite, that a compiler may be blocked from inferring and optimizing these design patterns. In our example results using the Clang front end and -03 optimizations, we saw that all design patterns in the model are faced with optimization blockers. In this section, we demonstrate that by using certain code transformations, we can remove these optimization blockers. Moreover, these code transformations can potentially be implemented as compiler passes to automate the process. It is important to note that this is not an exhaustive list of potential transforms; rather, our focus is on providing a representative example for each design pattern.

6.8.1 SIMD

We address the SIMD optimization blocker by manually unrolling the loop, as shown in Algorithm 24. Based on the relevant profiler results shown in Figure 6·30, we can see that the code can achieve 32 FLOPs/cycle throughput. Moreover, as shown by the concurrency analysis, the compiler can also establish lack of dependency between additions, and thus places nearly all adders within the same pipeline stage. The overall runtime improvement is a modest 1.03x, but this is primarily due to the off-chip memory stalls.

6.8.2 Pipelining

To help the compiler infer pipelines and register based pipeline registers, we manually unroll the loop and use individual variables to represent each systolic array element. The code for this is the same as the inlining probe (Algorithm 20). As shown by Figure 6·31, this has two benefits. First, no local memories are used and the pipeline

Algorithm 24 SIMD Probe

```
kernel void vecAdd(global float* restrict const A, global float* restrict const B, global
float* restrict C){
        for (int i = 0 ; i ¡ SIZE; i+=32){
                C[i+0] = A[i+0] + B[i+0];
                C[i+1] = A[i+1] + B[i+1];
                        ⋮
                C[i+31] = A[i+31] + B[i+31];
        }
}
```

is composed solely of registers. Second, since manual unroll removes the nested loop and the resulting code is well pipelined, we get an estimated performance improvement of 814x.

6.8.3 Caching

We manually add caches for each off-chip data structure, as shown in Algorithm 25. The code first populates these local memories, and then uses them to performs reads/writes during the computation. This eliminates off-chip access from the critical path. Moreover, the off-chip memory transactions are now coalesced, which results in higher utilization of memory bandwidth (as compared to single value accesses). Figure 6·32 shows that stalls are removed from the nested loops, and as a result we get an estimated 3500x improvement in run time. While our profiler currently does not analyze block memory copy routines, such as "llvm.memcpy", it is unlikely that the overhead of such operations for populating local memories will significantly increase the run time shown in Figure 6·32 .

6.8.4 Constants

To help the compiler create separate memory blocks, we manually create smaller constant arrays. These arrays are derived from the larger one shown in Algorithm 19 and their contents are based on the application's access patterns. Algorithm 26 shows

Run Time Estimate (Cycles)

9306 cycles

Floating Points Operations per Cycle

32 FLOPS/cycle

Concurrency Analysis

```
Pipeline Stages = 5
FLOPS per Stage:  0 1 31 0 0
FLOPS per Stage:  0 1 31 0 0
Memory OPs per Stage:  2 62 1 31 0
```

Figure 6·30: Relevant profiler results for the SIMD probe showing that 32 floating point adders have been inferred.

the resulting code. From Figure 6·33, we can see that using smaller constant arrays reduces overall memory requirements from $51200 \times 16 = 800Kb$ to $3200 \times 16 = 50Kb$, a difference of 16x. We also observe that since this particular code transformation was aimed at resource optimization, the run time estimate does not change.

6.8.5 Inlining

Algorithm 27 gives our code for manually inlining the systolic array. Based on Figure 6·34, we can see that while the estimated runtime increases slightly due to more pipeline stages added, the number of paths is reduced from 65536 to 1.

Loops

Algorithm 28 shows an implementation of matrix multiplication with coalesced loops. Effectively, the three nested loops of trip counts = 32 are replaced with a single loop of trip count 32^3, while the indexes i, j an k are derived from this loop's iterator. From Figure 6·35, we can see that coalescing reduces the number of loops and, while

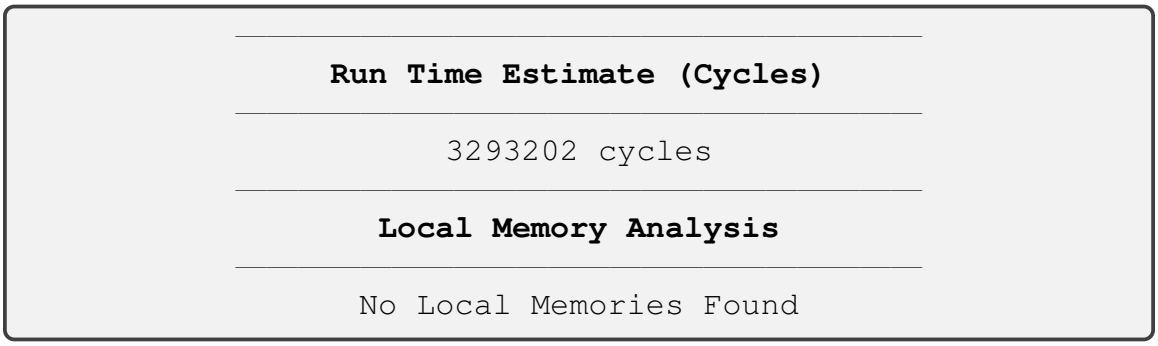


Figure 6·31: Relevant profiler results for the Pipelining probe showing that no local memories are inferred i.e. pipeline registers have been inferred as registers.

it increases the stall amount per loop iteration, the estimated run time improves by 80x.

6.8.6 BRAM Read-After-Write Hazard

We address RAW hazards for local memories using the code shown in Algorithm 29. Instead of storing values directly in the table, we place the result (as well as the destination index) in a register. This register is represented as individual variables (as opposed to an array). Unlike BRAMs, registers can be read immediately after being written to. Therefore, when a request for load is received, we first check the register to see if there is a match. In the event that it is a read-after-write, this check would be true and data from the register is read out instead of the table. Otherwise, we are guaranteed that valid data is available in the table, and is thus read directly. Simultaneously, while the memory operation for an iteration is taking place, the register stores the result of the previous iteration in the table. Figure 6·36 shows that this approach removes the RAW hazard, and improves estimated run time by 2x.

Algorithm 25 Caching Probe

```
kernel void all_cache(global int* restrict const A, global int* restrict const B, global int*
restrict C){
        int a[SIZE*SIZE]; int b[SIZE*SIZE]; int c[SIZE*SIZE];
        for (int i = 0; i < SIZE*SIZE; i++)
                a[i] = A[i];
        for (int i =0; i < SIZE*SIZE; i++)
                b[i] = B[i];
        for (int i = 0; i < SIZE*SIZE; i++)
                c[i] = 0;
        for (int i = 0 ; i < SIZE; i++){
                for (int j = 0 ; j < SIZE; j++){
                        for (int k = 0; k ¡ SIZE; k++){
                                c[i*SIZE + j] += a[i*SIZE + k] * b[k*SIZE + j];
                        }
                }
        }
        for (int i = 0; i < SIZE*SIZE; i++)
                C[i] = c[i];
}
```

6.8.7 Execution Paths

Algorithm 30 shows our approach for reducing execution paths. We manually reduce the amount of calculations done per instruction. This is aimed at allowing the compiler to better infer potential pipelines in the code by simplifying its view of the computation. Figure 6·37 shows that this approach reduces the number of paths from 3 to 1.

6.8.8 Input Multiplexing

To improve logic reuse, we restructure code such that instead of selecting the output of a valid add operation (given values of i and A[i]), we select the inputs of the add operation (Algorithm 31). Consequently, as shown in Figure 6·38, the number of floating point adders needed is reduced from 3 to 1 (as shown by the FLOPS/cycle value and concurrency analysis), and the estimated run time improves by 1.2x.

Algorithm 26 Constants Probe

```
int maval (int a, int b, int c);

constant int lookup0 [ITERS] = ....
constant int lookup1 [ITERS] = ....
        ⋮
constant int lookupSIZE-1 [ITERS] = ....

kernel void single_const_array(global int* restrict const A, global int* restrict C){
        int systol_0 = 0; int systol_1 = 0; ... int systol_SIZE-1=0;
        int systol_1_0 = 0; int systol_1_1 = 0; ... int systol_1_SIZE-1=0;
        for (int i = 0; i < ITERS; i++){
                systol_1_0 = systol_0;
                systol_0 = maval(0, A[i], systol_1_0 + lookup0 [i]);
                systol_1_1 = systol_1;
                systol_1 = maval(systol_1_0, systol_0, systol_1_1 + lookup1 [i]);
                                                ⋮
                systol_1_SIZE-1 = systol_SIZE-1;
                systol_SIZE-1 = maval(systol_1_SIZE-2, systol_SIZE-2, systol_1_SIZE-1 +
                lookupSIZE-1 [i]);
                C[i] = systol_SIZE-1;
        }
 }
```

Algorithm 27 Inlining Probe

```
int maval (int a, int b, int c);

kernel void reg_array_func(global int* restrict const A, global int* restrict C){
        int systol_0 = 0; int systol_1 = 0; ... int systol_SIZE-1=0;
        int systol_1_0 = 0; int systol_1_1 = 0; ... int systol_1_SIZE-1=0;
        for (int i = 0; i < ITERS; i++){
                systol_1_0 = systol_0;
                int u0 = A[i]; int v0 = 0; int s0 = (u0 > v0) ? u0 : v0;
                int t0 = systol_1_0 + 1; systol_0 = (t0 > s0) ? t0 : s0;
                                                ⋮
                C[i] = systol_SIZE-1;
        }
 }
```

Run Time Estimate (Cycles)

37888 cycles

Stall Analysis

Off Chip Memory Access : 0
Read After Write Hazards (warning): 3

Loop Analysis

Loop: main
Iterator:
Limit: 1
Stall: 0
Children: 1

Loop: main_1
Iterator: indvars.iv21
Limit: 32
Stall: 0
Children: 1

Loop: main_1_1
Iterator: indvars.iv18
Limit: 32
Stall: 0
Children: 1

Loop: main_1_1_1
Iterator: indvars.iv
Limit: 32
Stall: 0
Children: 0

Figure 6·32: Relevant profiler results for the Caching probe showing that there are no off-chip memory accesses within the critical path, and as a result loops operate stall free.

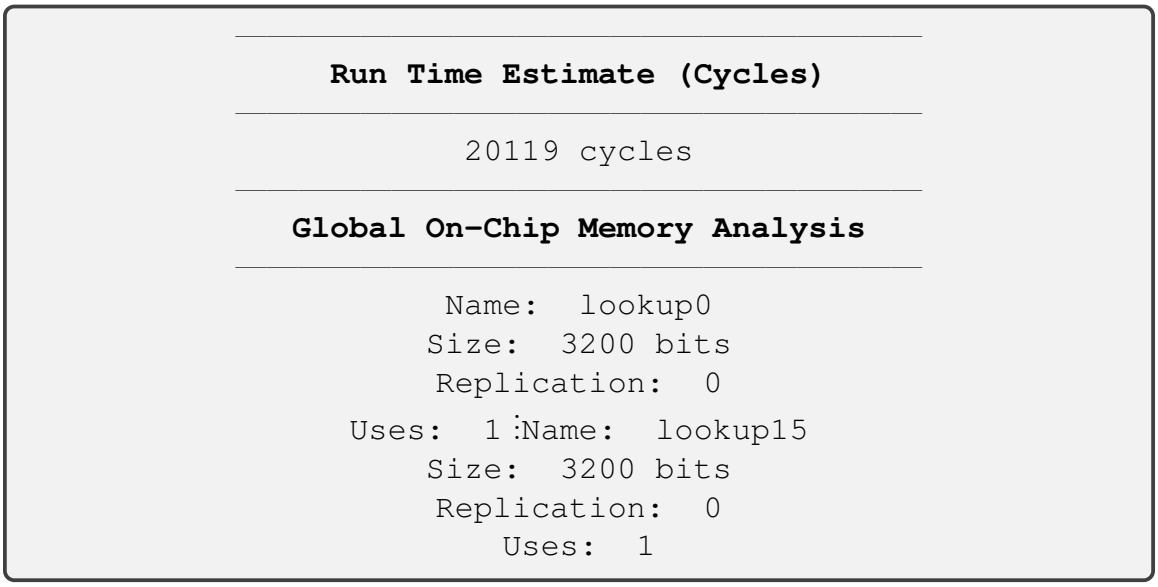


Figure 6·33: Relevant profiler results for the Constants probe showing that resource usage has been minimized i.e. each sink has its own on-chip memory block containing only required data.

6.8.9 Floating Point Accumulator

Since floating point accumulation is difficult to resolve without hardware support, we use an aggressive code transformation. Specifically, we unroll the inner loop such that the need for accumulation is removed altogether. Moreover, we do not create an adder tree (by adding brackets around additions) to ensure the computation is in order. It is important to note that this approach is only useful if there are sufficient resources available. Figure 6·39 shows that our approach removes the floating point accumulation stall, and improves estimated run time by 100×.

```
Run Time Estimate (Cycles)

3293218 cycles

Inling Analysis

User Function Calls = 0
Floating Point Mul-Add Efficiency = N/A

Utilization Analysis

Number of Paths:  1
```

Figure 6·34: Relevant profiler results for the Inlining probe showing that the number of paths have been reduced to 1, which indicated a tigheter integration of pipelines.

```
Run Time Estimate (Cycles)

16580608 cycles

Loop Analysis

Loop:  main
 Iterator:
 Limit:  1
 Stall:  0
Children:  1

Loop:  main_1
Iterator:  m.04
Limit:  32768
 Stall:  500
 Children:  0
```

Figure 6·35: Relevant profiler results for the Loops probe showing a single coalesced loop.

Algorithm 28 Loops Probe

```
kernel void coalesced_matMul(global int* restrict const A, global int* restrict const B,
global int* restrict C){
        int i,j,k;
        i = 0; j = 0 ; k = 0;
        for (int m = 0; m < SIZE*SIZE*SIZE; m++){
                if (k == 0)
                        C[i*SIZE+j] = 0;
                C[i*SIZE + j] += A[i*SIZE + k] * B[k*SIZE + j];
                k++;
                if (k == SIZE){
                        k = 0;
                        j++;
                        if (j == SIZE){
                                j = 0;
                                i++;
                        }
                }
        }
}
```

```
Run Time Estimate (Cycles)

131 cycles

Utilization Analysis

Number of Paths:  3
Lines of Code:  29 32 36
FLOPS: 0 0 0
FixOPS: 1 1 1

Stall Analysis

Read After Write Hazards (warning) :  1
Memory = table
Line Number = 25
```

Figure 6·36: Relevant profiler results for the BRAM RAW probe showing that the known RAW Hazard has been removed

Algorithm 29 BRAM Read-After-Write Hazard Probe

```
kernel void reg_store(global int* restrict const A, global int* restrict const B, global int*
restrict const C){
        int table[SIZE];
        int temp_data = 0;
        int temp_hash = 0;
        for (int i = 0 ; i < ITERS; i++){
                int local_temp_data = temp_data;
                int local_temp_hash = temp_hash;
                int hash = ((A[i] >> 5) & (A[i] << 7)) & (SIZE-1);
                if (B[i] == 0){
                        temp_data = C[i];
                        temp_hash = hash;
                }
                else{
                        int read;
                        if (hash == temp_hash)
                                read = local_temp_data;
                        else
                                read = table[hash];
                        temp_data = read;
                        temp_hash = hash;
                        C[i] = read;
                        }
                table[local_temp_hash] = local_temp_data;
        }
}
```

Algorithm 30 Execution Paths Probe

```
kernel void single_path(global float* restrict const A , global float* restrict const B,global
float* restrict C){
        for (int i = 0 ; i ¡ SIZE; i++){
                float in1 = A[i] + B[i];
                float in2 = A[i] + SIZE;
                float in3 = B[i] + SIZE;
                float out1 = C[i];
                float out2 = (A[i] == SIZE) ? in3 : out1;
                float out3 = (A[i] < SIZE) ? in2 : out2;
                float out4 = (i > 0) ? in1 : out3;
                C[i] = out4;
        }
}
```

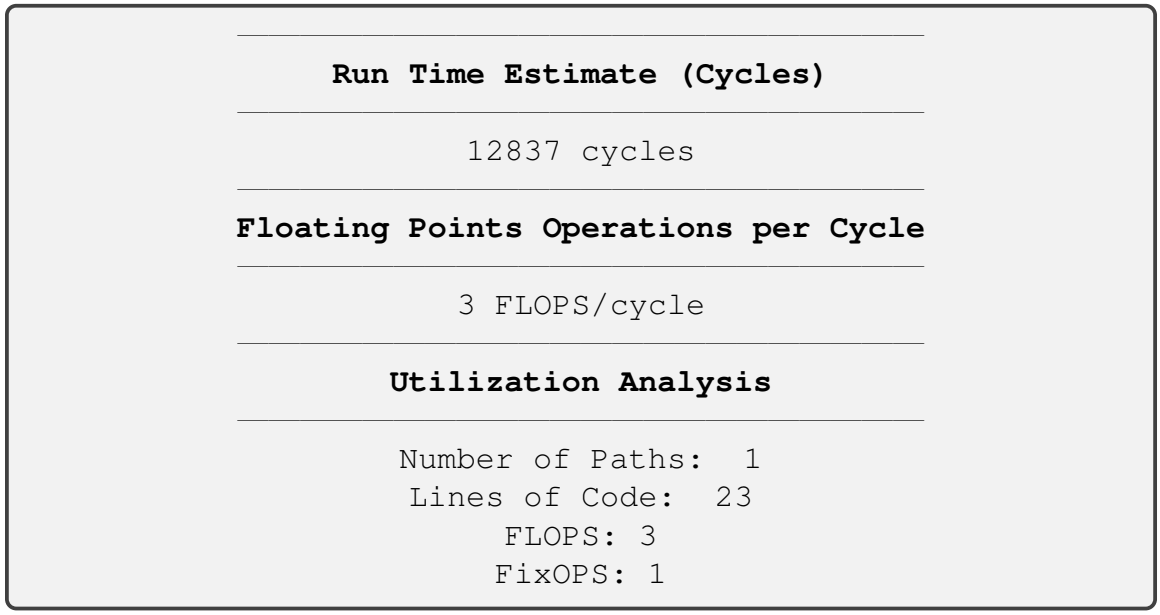


Figure 6·37: Relevant profiler results for the Execution Paths probe showing that the number of paths have been reduced to 1 while maintaining the same number of floating point operations.

Algorithm 31 Input Multiplexing Probe

```
kernel void predication(global float* restrict const A , global float* restrict const B,
global float* restrict C){
        for (int i = 0 ; i < SIZE; i++){
                float in1 = ((i == 0) && (A[i] == SIZE)) ? B[i] :
                                (((i == 0) && (A[i] > SIZE)) ? C[i] : A[i]);
                float in2 = ((i == 0) && (A[i] <= SIZE)) ? SIZE :
                                (((i == 0) && (A[i] > SIZE)) ? 0 : B[i]);
                C[i] = in1 + in2;
                }
        }
}
```

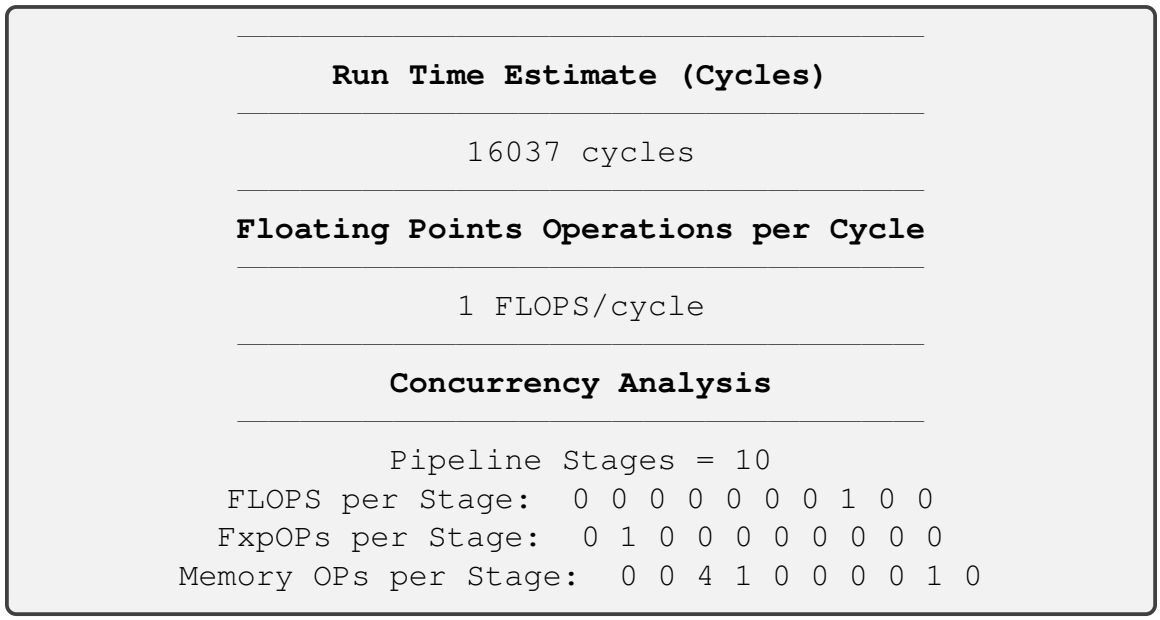


Figure 6·38: Relevant profiler results for the Input Multiplexing probe showing that the number of floating point units needed have been reduced to 1.

Algorithm 32 FP Accum Probe

```
kernel void no_cache(global float* restrict const A, global float* restrict const B, global
float* restrict C){
    for (int i = 0 ; i < SIZE; i++){
        for (int j = 0 ; j < SIZE; j++){
            C[i*SIZE + j] =
                        A[i*SIZE + 0] * B[0*SIZE + j]
                        + A[i*SIZE + 1] * B[1*SIZE + j]
                        + A[i*SIZE + 2] * B[2*SIZE + j]
                        + A[i*SIZE + 3] * B[3*SIZE + j]
                                        ⋮
                        + A[i*SIZE + 29] * B[29*SIZE + j]
                        + A[i*SIZE + 30] * B[30*SIZE + j]
                        + A[i*SIZE + 31] * B[31*SIZE + j];
        }
    }
}
```

Run Time Estimate (Cycles)

5056 cycles

Stall Analysis

```
Read After Write Hazards (known) :  0
Read After Write Hazards (warning):  0
   Floating Point Accumulator :  0
    Off Chip Memory Access :  63
```

Figure 6·39: Relevant profiler results for the FP Accum probe showing that there floating point accumulator has been removed from the design.

6.9 Profiling Intel OpenCL SDK for FPGAs

We now apply our profiler to Intel OpenCL and evaluate compiler effectiveness. The binary code for each stage of the compilation in the Intel OpenCL toolflow can be obtained by setting appropriate flags in the compilation script. We use the final binary code generated by "aocl-llc" which is a result of performing all compiler passes; in the normal flow, this binary code is subsequently used by the "system_generator" routine to create the kernel system. To obtain the ".ll" assembly file, we run the binary code through the LLVM disassembler. Table 6.1 shows the results of compiling our probes using the Intel OpenCL v16.0.2 compiler, and analysing the resulting .ll assembly files. From the table, we see that a majority of key optimizing transforms are blocked. The compiler is only successfully able to implement a single execution path, input multiplexing and floating point accumulation. Execution path was the most consistent optimizating transform, since compilation results for all probes had a single path. On the other hand, floating point accumulation was the least reliable since it used a single cycle accumulator which is only available in Intel Arria 10 FPGAs. Moreover, while the compiler was able to inline a user function call, the number of instructions taken was greater than if we used the probe from Algorithm 27. This indicates that the compiler has limited freedom when integrating pipelines.

Table 6.1: Summary of Profiling Intel OpenCL Compiler using Probes

Probe	**Results**
SIMD	**Blocked**
Pipelining	**Blocked**
Caching	**Blocked**
Constant	**Blocked**
Inlining	Inefficient
Loops	**Blocked**
BRAM RAW Hazard	**Blocked**
Execution Paths	Efficient
Input Multiplexing	Efficient
Floating Point Accumulation	Efficient

We compare the results of our profiler with the set of systematic transformations

given in chapter 5. From Table 6.2, we see that all our empirically guided transformations are indeed needed since they address one or more optimizations being blocked. Moreover, for optimizations that were efficiently performed by the compiler, no pre-processing transform was needed.

Table 6.2: Matching System Code Transformations with Profiler Results

Ver.	**Transformations**	**Blocked Design Pattern**
1	Single thread code with cache optimization	Baseline Model
2	Implement task parallel computations in separate kernels and connect them using channels	Pipelining
	Unroll loops using w/ *#pragma unroll*	SIMD,Pipelining
	Minimize variable declaration outside compute loops – use temps where possible	Pipelining
	Use constants to reduce spatial footprint and reduce replicated array sizes based on access patterns	Constants
	Coalesce memory operations and loops	Caching,Loops
3	Inline kernels and express computation within a single kernel	Inlining
4	Reduce array sizes to infer pipeline registers as registers, or add support for removing BRAM RAW hazards	Pipelining,RAW
5	Perform computations in detail to improve the compiler's dependency analysis	Pipelining
6	Use predication instead of conditional branch statements when defining forks in the data path	Inlining

6.10 Conclusion

In this chapter, we provided a framework for instrumenting HLS compilers. This allows us to get greater insights into pre-processing transformations, i.e. what works, how it works and why it works. The projected result of our contribution is that transforms will become more robust, which in turn addresses limitations of work presented in chapter 5. Moreover, the compiler will improve since we can rapidly

explore the impact of existing and new passes, as well as their ordering. Finally, we can also address the limitation of grammar i.e. we can identify pragmas needed if the HLL does not have sufficient expressability for certain design patterns.

Chapter 7

Formalization and Generalization of Hardware Operating Systems

7.1 Motivation

Hardware operating systems are effectively any logic on the FPGA that is not part of the application. They are responsible for partitioning device fabric between multiple entities, data flow management and interfaces, and hardware modifications. They also manage the flow of data between different components in the FPGAs by defining a number of specifications such as APIs, protocols, bus widths, clock domains, FIFO depths etc. With regards to external connectivity, hardware operating systems implement interfaces through controllers and abstractions; the latter is useful in masking the complexity of the former and making protocols/interfaces coherent with intra-FPGA data and control planes.

For elastic and shared FPGAs in particular, a hardware operating systems supports simultaneous deployment of application logic (through application support discussed previously) and system administrator logic. System administrator logic in FPGAs enables implementation of important, often performance critical, services and functions. This includes crypto, firewalls/isolation, packet processing, control/management planes, event logs/counters, firmware attestation services etc. A hardware operating system also enables modification of deployed logic in a number of forms with varying overhead, including i) hotfixes (run-time parameter updates), ii)

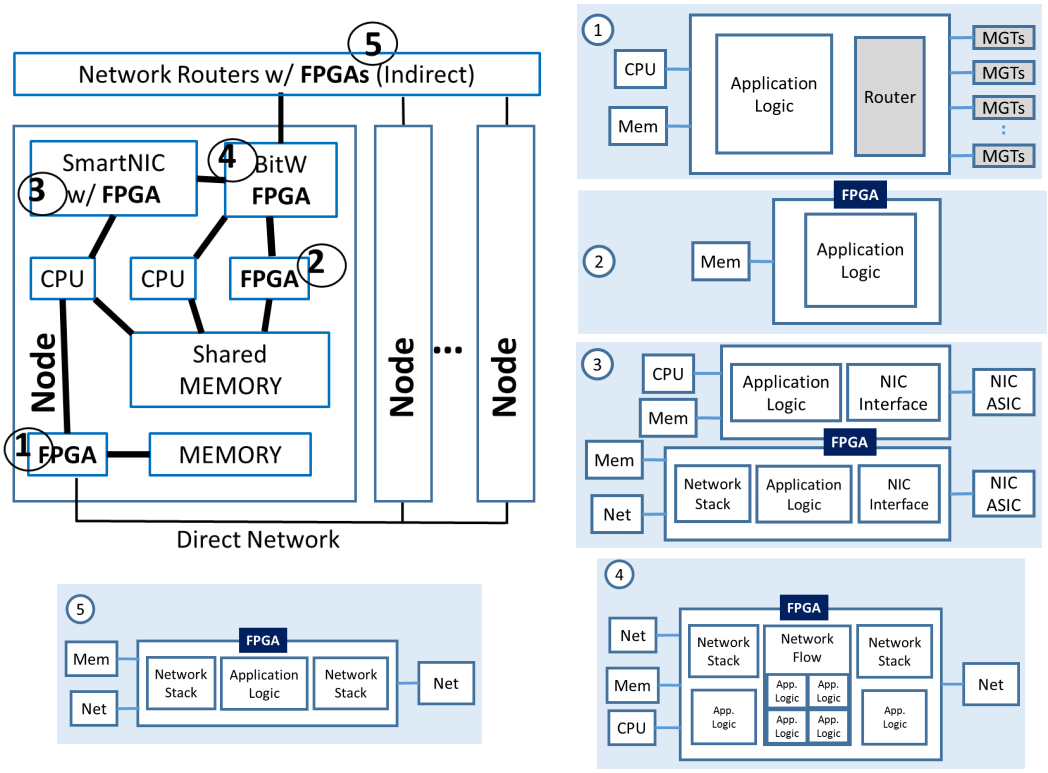


Figure 7·1: Partial RHOS Taxonomy

partial reconfiguration (modifications to a specific pre-defined region of the FPGA), or iii) full reconfiguration (updating the entire fabric).

The drawback of the current state of hardware operating systems, as shown in chapter 2, is that they are ad hoc, i.e. developed for every system. Since our target is the "CPU-ization" of these FPGA systems, we present a framework to formalize and generalize these hardware operating systems. We call the resulting uniform OS a Reconfigurable Hardware Operating System (RHOS).

7.2 Partial RHOS Taxonomy

To be able to generalize hardware operating systems, we need a taxonomy. In our work, this was done by looking a the different places the FPGA appears in a system and what the RHOS needs to be for each. Figure 7·1 gives our partial RHOS taxonomy based on our analysis of previous work discussed in chapter 2. While this is not an exhaustive list, it covers some of the most common methods of deploying FPGAs in Data Centers.

7.3 Major Components Types of a RHOS

We broadly categorize the major component types of a RHOS into four sets. These are:

1. **C1**: This refers to components for which we have prior knowledge of their connectivity and functionality. For example, controllers for reconfiguration, memory, and ntwork will likely be connected to appropriate pins for corresponding external devices. While the functionality of this set can change, the average timeframe after which it may need to be done is typically orders of magnitude higher than other sets. Therefore, these can be assumed as static.

2. **C2**: This refers to components for which we have a prior knowledge of their connectivity and a baseline functionality. System administrators can make small modifications to the functionality through updates to the control plane; all operational scenarios for the data path are already implemented in these components. For example, a switch typically implements a crossbar of fixed dimensionality since it depends on the number of sources and destinations. However, the rules of this crossbar depend on arbitration algorithms, which can be changed dynamically.

3. **C3**: This refers to components for which we have no prior knowledge of their functionality and connectivity. As such, reconfigurable space is allocated in order for these components to be deployed at run time. Moreover, these are connected via internal crossbars to all other components, as well as developer logic. An example of this is Network Flow in configuration (4) in the previous section. Given a BitW FPGA, we cannot assume a particular form of network flow i.e. it could be as simple as a wire that just connects the two network stacks together, or it could implement complex packet processing. It also cannot be assumed that the network flow logic will be directly connected to network stacks. Therefore, we do not know of the functionality or connectivity of this network flow, unless it is explicitly specified (which could happen at run time).

4. **APIs**: This refers to the intra-FPGA interfaces between components. Having standardized interfaces is critical, since it ensures that components can be added and removed without changes needed to other parts of the RHOS. This in turn means that we do not need to build the a new hardware operating system for the entire taxonomy above; instead, we can reuse a substantial amount of logic and only build components unique to a configuration.

7.4 Method for Building a RHOS Generator

We build the RHOS generator as a custom Board Support Package (BSP) for the Intel FPGA OpenCL SDK. As discussed earlier, a BSP is set of hardware and software components, as well as Partial Reconfiguration regions and pin-out definitions, required to compile and deploy user applications on a specific FPGA. Specifically, it has existing implementations for the C1 set identified earlier in this section. Having a BSP based implementation ensures compatibility across the stack, going from high level code to hardware execution on the FPGA. Therefore, we package all RHOS

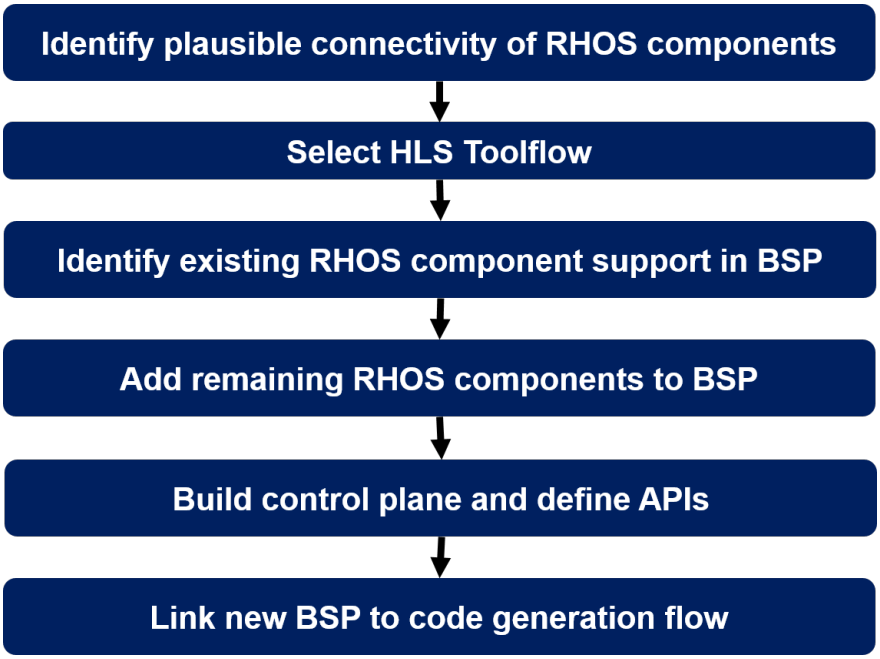


Figure 7·2: Method for building a RHOS generator

components (including system administrator logic) into an OpenCL wrapper and add that to a baseline BSP (containing DRAM/PCIe/PR controllers). Intel OpenCL also supplies the drivers needed for programming the FPGA, DMA operations and start/stop triggers for a workload.

The specific process is listed in Figure 7·2. We first identify a plausible connectivity and hierarchy of RHOS components. Then, for a given HLS toolflow, we identify what RHOS components present in the BSP can be reused. For the remaining components, we add the logic to this BSP using defined APIs. A control plane is also built to manage the augmentations made to the BSP. Finally, this new BSP is linked to the code generation flow to ensure compatibility and automatic transition between the code generation and deployment generation parts of HaaRNESS.

7.5 Proposed Organization of RHOS Components

Figure 7·3 shows our proposed model for a RHOS. Here we list component types and discuss their composition.

- **C1**
  - PCIe Controller: Used by the CPU for DMA operations, interrupts, control signalling etc.
  - Reconfiguration Controller: Interfaces the host for modifying developer and lookaside logic.
  - Mem. Controller: Provides off chip memory access.
- **C2**
  - Lightweight Network Stack: The Lightweight Network Stack implements up to Layer 2 networking support. Example of baseline functionality here is sending and receiving packets at line rates for supported network protocols.
  - In-line Logic: In-line processing here refers to functions that operate on network packets at line rates. It is connected to the network stack and the Smart Switch. Examples of baseline functionality here are encryption/decryption and bypass switches to enable network traffic to selectively bypass one or more modules within the in-line logic to reduce latency.
  - Smart Switch: At the heart of the FPGA is the Smart Switch which implements a number of features that manage the flow of ingress and egress data, enabling connectivity between different components within the FPGA. An example of baseline functionality are a crossbar with support for Layer 3 of the OSI model, which analyzes incoming meta data to switch packets between different endpoints. An example is resolving egress port contention

based on defined Quality of Service algorithms e.g. round robin, priority first.

- Host Interface: We include the host interface in C2 since the FPGA may not exclusively connect to the CPU via PCIe. Similar to Microsoft Catapult 2, a second network stack can also be implemented for supporting bypass operations i.e. data moves directly from Data Center network to host NIC and vice versa using a wire-like connection inside the FPGA.

- **C3**

  - Lookaside Logic: This is used for implementing components which are only useful for very specific contexts. An example is Key Value Store, which is an important Data Center application in its own right, but has fairly limited benefits to workloads in general. It is important to note that some partitions of the Lookaside fabric can also be used as C2 components.

- **API**

  - Since we have used the Intel OpenCL SDK, we leverage its APIs for interfacing FPGA logic from CPU.
  - With regards to interfacing the RHOS, we use blocking OpenCL channel calls to implement FIFOs between the Smart Switch and developer logic.
  - Support is also provided for accessing the FPGA using the network interface for modifying parameters, updating algorithms and reading internal counters/states. By using the appropriate network addresses, system administrators can remotely access, modify, monitor and control logic on FPGAs.

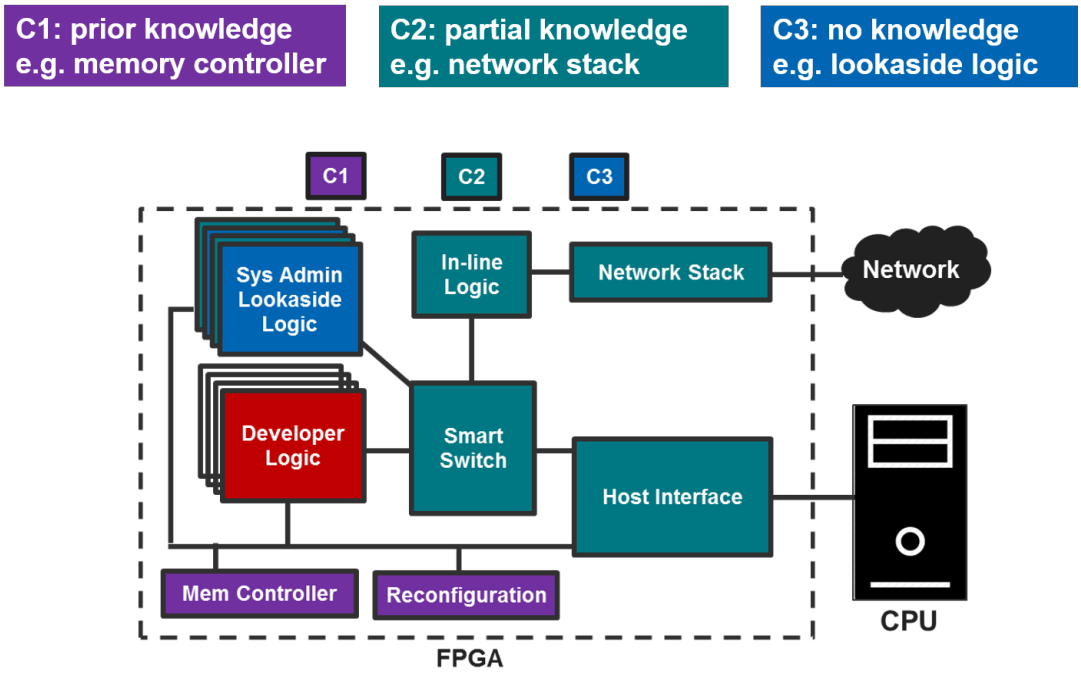


Figure 7·3: Proposed connectivity and hierarchy of RHOS component types

7.6 Application of RHOS Framework to Taxonomies

Creating a new RHOS is a significant effort that would require many dissertations. To demonstrate the viability and usefulness of our approach, we instead provide implementation details for one taxonomy i.e. BitW FPGAs. For the other taxonomies shown in Figure 7·1, we demonstrate that the RHOS framework can be applied to them as well.

7.6.1 BitW FPGAs

Here we describe the design of a RHOS for BitW FPGAs, called Morpheus. We provide details of four important aspects of the deployment i.e. Network Stack, In line logic, Smart Switch and Control Plane.

Network Stack

Since the baseline BSP does not contain a network stack, we provide a lightweight modular implementation in Morpheus. Currently, we do not provide support for FPGA bypass i.e. direct network to CPU connectivity over a low latency connection inside the FPGA. Figure 7·4 shows the design of our network stack:

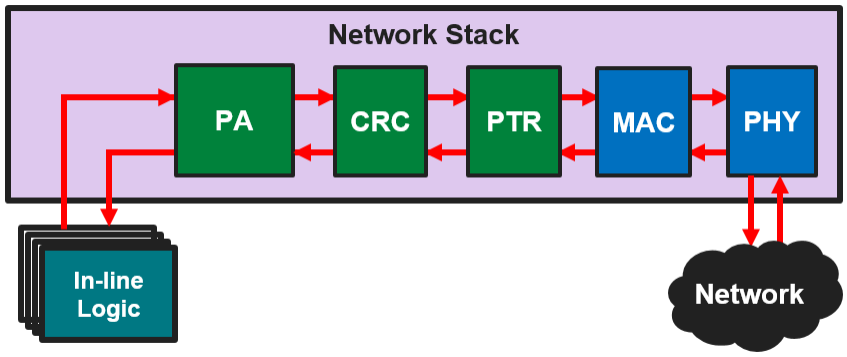


Figure 7·4: Overview of the modular Morpheus Network Stack

PHY: This is implemented using vendor IP to ensure optimal resource usage, latency and operating frequency. An example of this is the Arria 10 Transceiver Native PHY IP block (Intel, 2019).

MAC: This is implemented using vendor IP to leverage the benefits stated above. An example of this is the Low Latency Ethernet 10G MAC IP block (Intel, 2019).

PTR (Packet Transmitter Receiver): Figure 7·5 shows an overview of the Packet Transmitter Receiver (PTR) module. PTR is used to convert ingress sequential packet data from the MAC to a full packet which can move within the FPGA in parallel. Similarly, it also converts full egress packets to sequential data and sends them to the MAC. Its interfaces are streaming on both the MAC and CRC side (the CRC side has fewer wires since it is not required to support the extra ones e.g. start of packet (sop)). Both transmitter and receiver components of PTR operate independently and concurrently as state machines. When sending a packet, the transmitter state machine

buffers the packet in an input register, and then locks the input ports from the CRC module. This is done to stall other egress packets till the process is complete. When receiving a packet, a trigger is used to buffer the full packet in the output register, after which it is sent to the CRC module.

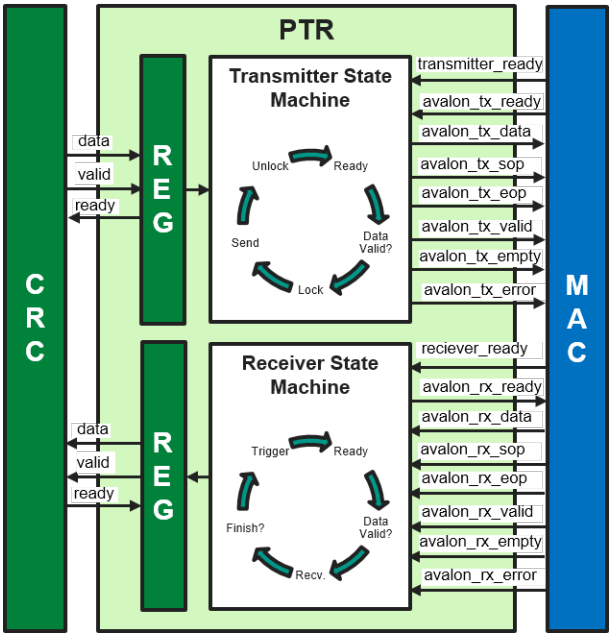


Figure 7·5: Packet Transmitter Receiver: Interfaces the MAC to send and receive packets using state machines

CRC (Cyclic Redundancy Check): The CRC module is used to generate and verify appropriate checksums, based on the packet protocol, in order to ensure that ingress packets are received correctly. These checksums are also added to egress packets so that the same is possible on remote nodes.

PA (Payload Alignment): Ingress payload data from the MAC to the PTR module, received sequentially in 32 bit blocks, is trivially converted to a full vector by shift operations. Since data arrives MSB first, we simply left shift MAC outputs into a register. As a result valid payload data, that is less than the maximum possible size, is right aligned. This is shown in Figure 7·6a. This right aligned format is maintained throughout the FPGA. The problem here is that right aligned egress payload data

cannot be trivially shifted out from the PTR module to the MAC. This is because start of valid data can occur in any of hundreds of potential locations. A large MUX is needed to select between these possibilities, which can significantly increase resource consumption. This is shown in Figure 7·6b. The red line here shows the first valid byte. The PA module left aligns payload data, so that the PTR module can trivially left shift it out to the MAC module as shown in Figure 7·6c. The latency of PA module can potentially be masked by the stall between transmissions since the MAC has a fixed 32 bit interface and requires a significant number of cycles to transmit a packet.

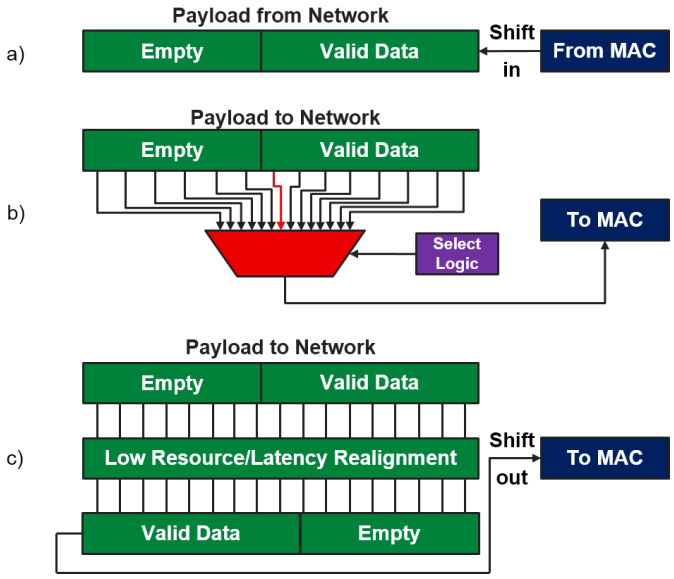


Figure 7·6: Conversion between 32 bit MAC interface and full payload vector. a) Ingress data is trivially handled through left shifts. b) For egress data, since the actual payload size can vary, using a MUX to select an arbitrary 32 bit value in the payload has a high hardware overhead. c) Alternative approach for egress data, where the payload is left aligned using the PA module, so that data can be simply be shifted similar to (a).

In line logic

We implement AES-CTR-128 as an example of In line logic. We use counter based AES block mode for line rate encryption since it is easy to parallelize. The "nonce" value is hard coded in our design.

Smart Switch

Similar to In line logic, we implement a simple Smart Switch. It is composed of a cross bar which assigns priority based on values embedded in incoming data.

Control Plane

An important design aspect that we highlight here is the use of Softcores for the control plane. This has the following benefits:

- Softcores are programmed in Embedded C. Not only can this facilitate greater and more effective adoption of this technology, but a number of legacy algorithms are written in high level languages, and can potentially be translated to Embedded C with little effort.

- The resource overhead of a Softcores is typically very low. We can potentially generate a separate Softcore for each high level functionality deployed by the system administrator. This means we can create a MPSoC with highly specialized and specific algorithms running on each Softcore, which in turn could result in a decrease in implementation complexity and a faster response to event flags.

- Softcores can be reprogrammed at runtime simply by updating the instruction/data memories and resetting the processor. This means that entire algorithms can be modified over the network within a few microseconds. Moreover,

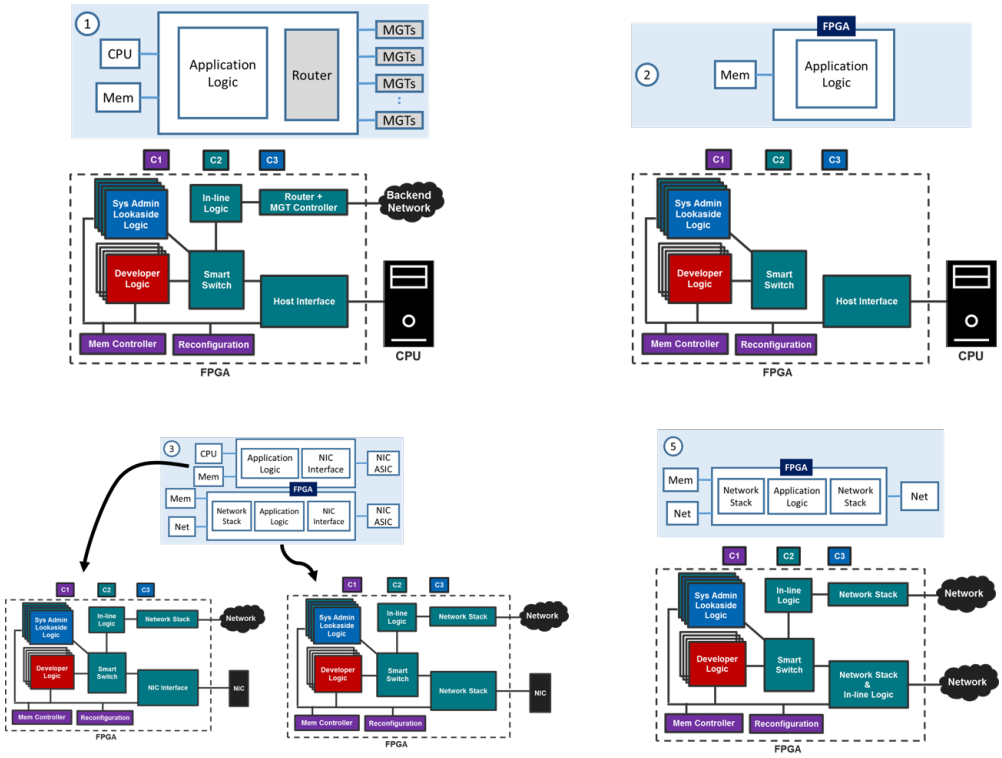


Figure 7·7: Application of RHOS generator framework to Back-end, storage attached, SmartNIC and TOR switch FPGAs.

the data needed to reprogram the Softcore will typically be in the order of a few kilobytes, and hence is unlikely to generate large amounts of network traffic if done remotely.

7.6.2 Other Taxonomies

Figure 7·7 illustrates how the RHOS framework can be applied to (1) Co-Processor FPGAs with a back-end network, (2) storage attached FPGAs, (4) SmartNICs, and (5) FPGAs embedded in TOR Switch. (3) is a BitW FPGA which is alredy discussed above. In all designs, only components unique to the taxonomy are modified, while the remaining components are reused. Moreover, the standard APIs ensure that neither Developer application logic nor other RHOS components need to be modified if changes are made.

7.7 Conclusion

Removing ad hoc'ness in runtime support can substantially reduce developer effort. In this chapter, we presented the design of a RHOS generator which addresses this ad hoc'ness in hardware operating systems. We started by building a partial RHOS taxonomy. Then we identified the major component types of a RHOS, as well as their connectivity and hierarchy. We also presented our approach for building the RHOS generator. Finally, we demonstrated the validity and usefulness of our proposed RHOS generator by showing that it can be applied to a number of taxonomies.

Chapter 8

Conclusion

In this dissertation, we demonstrated that it is possible to "CPU-ize" FPGAs by providing capability of fast, high quality code generation and uniform deployment generation within the same framework.

Fast code generation is achieved by building a framework for performing RTL simulations of kernel logic and application logic, which significantly reduces design space exploration overhead for HLS. The framework was also shown to be useful for generating custom IP, which can outperform vendor IP and ASICs due to greater application specificity.

High quality code generation is achieved by applying our proposed systematic set of code transformations to HLL code. These transformations remove optimization blockers in order to bridge the HLS/HDL gap. We also provide a framework for compiler instrumentation and static profiling of HLS IR code, which enables automatic identification of these blockers and integration of pre-processing transformations into the HLS compiler.

Finally, uniform deployment generation is achieved by making the runtime support uniform and vendor agnostic. In this regards, we presented our design of a RHOS generator and its application to multiple common configurations of Data Center FPGAs.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1704.04760*.

Abedalmuhdi, A., Wells, B. E., and Nishikawa, K.-I. (2017). Efficient Particle-Grid Space Interpolation of an FPGA-Accelerated Particle-in-Cell Plasma Simulation. In *Proc Field-Programmable Custom Computing Machines*, pages 76–79.

Abel, F., Weerasinghe, J., Hagleitner, C., Weiss, B., and Paredes, S. (2017). An FPGA Platform for Hyperscalers. In *2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI)*, pages 29–32. IEEE.

Accolade (2018). FPGA-Based 1-100GE NICs / Platform Enable Host CPU Offload and Application Acceleration . `accoladetechnology.com/products/`. Accessed 2018-10-16.

Algo-Logic (2019). Algorithms in Logic. `www.algo-logic.com`. Accessed: 2019-05-25.

Alibaba (2019). Compute Optimized and FPGA-Equipped Instance Type Families. `www.alibabacloud.com/help/doc-detail/108504.htm`. Accessed 2019-09-12.

Amazon (2018). Amazon EC2 F1 Instances - Run Customizable FPGAs in the AWS Cloud. `aws.amazon.com/ec2/instance-types/f1/`. Accessed 2018-10-16.

Amazon (2019). Amazon EC2 F1 Instances. `aws.amazon.com/ec2/instance-types/f1/`. Accessed 2019-01-03.

Amazon (2019a). AWS Marketplace. `aws.amazon.com/marketplace`. Accessed: 2019-05-25.

Amazon (2019b). AWS Shell Interface Specification. `github.com/aws/aws-fpga/blob/master/hdk/docs/AWS_Shell_Interface_Specification.md`. Accessed 2019-01-05.

Anand, P. A. et al. (2016). Design of High Speed CRC Algorithm for Ethernet on FPGA using Reduced Lookup Table Algorithm. In *2016 IEEE Annual India Conference (INDICON)*, pages 1–6. IEEE.

Argonne (2017). Exascale Deep Learning and Simulation Enabled Precision Medicine for Cancer. http://candle.cels.anl.gov.

Arista (2019). 7130 FPGA-enabled Network Switches. www.arista.com/en/products/7130-fpga-enabled-network-switches-quick-look. Accessed: 2019-09-10.

Avnet (2019). IP Cores Parts by Avnet. www.avnet.com/shop/us/c/programmable-logic/ip-cores/. Accessed: 2019-05-25.

Baidu (2019). FPGA Cloud Server. cloud.baidu.com/product/fpga.html. Accessed 2019-01-03.

Balaprakash, P., Alexeev, Y., Mickelson, S. A., Leyffer, S., Jacob, R., and Craig, A. (2014). Machine-Learning-Based Load Balancing for Community Ice CodE Component in CESM. In *International Conference on High Performance Computing for Computational Science*, pages 79–91. Springer.

Ben-Nun, T., Licht, J. d. F., Ziogas, A. N., Schneider, T., and Hoefler, T. (2019). Stateful Dataflow Multigraphs: A Data-Centric Model for High-Performance Parallel Programs. *arXiv preprint arXiv:1902.10345*.

Benkrid, K., Liu, Y., and Benkrid, A. (2009). A Highly Parameterized and Efficient FPGA-Based Skeleton for Pairwise Biological Sequence Alignment. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(4):561–570.

BittWare (2018). BittWare OpenCL Board Support Packages. www.bittware.com/wp-content/uploads/datasheets/ds-OpenCL_BSP.pdf. Accessed: 2018-01-16.

Broadcom (2019). Stingray SmartNIC Adapters and IC. www.broadcom.com/products/ethernet-connectivity/stingray-smartnic/. Accessed 2019-01-03.

Brumme, S. (2018). Fast CRC32. http://create.stephan-brumme.com/crc32/.

Büyükkeçeci, F., Awile, O., and Sbalzarini, I. F. (2013). A Portable OpenCL Implementation of Generic Particle–Mesh and Mesh–Particle Interpolation in 2D and 3D. *Parallel Computing*, 39(2):94–111.

Byma, S., Steffan, J. G., Bannazadeh, H., Garcia, A. L., and Chow, P. (2014). FPGA in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 109–116. IEEE.

Byma, S. A. (2014). *Virtualizing FPGAs for Cloud Computing Applications*. PhD thesis, University of Toronto (Canada).

Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., and Czajkowski, T. (2011). LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM.

Caulfield, A. M., Chung, E. S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., et al. (2016). A Cloud-Scale Acceleration Architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 7. IEEE Press.

Chameleon (2019). A Configurable Experimental Environment for Large-Scale Cloud Research. `www.chameleoncloud.org/`. Accessed 2019-01-03.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A Benchmark Suite for Heterogeneous Computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54.

Chellappa, S., Franchetti, F., and Pueschel, M. (2008). How To Write Fast Numerical Code: A Small Introduction. In *Generative and Transformational Techniques in Software Engineering II, Lecture Notes in Computer Science v5235*, pages 196–259.

Chiu, M. and Herbordt, M. (2009). Efficient filtering for molecular dynamics simulations. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*.

Chiu, M. and Herbordt, M. (2010a). Towards production FPGA-accelerated molecular dynamics: Progress and challenges. In *Proceedings of High Performance Reconfigurable Technology and Applications (HPRCTA)*.

Chiu, M., Herbordt, M., and Langhammer, M. (2008). Performance potential of molecular dynamics simulations on high performance reconfigurable computing systems. In *Proceedings of High Performance Reconfigurable Technology and Applications (HPRCTA)*.

Chiu, M. and Herbordt, M. C. (2010b). Molecular Dynamics Simulations on High-Performance Reconfigurable Computing Systems. *ACM Transactions on Reconfigurable Technology and Systems*, 3(4):23.

Chiu, M., Khan, M., and Herbordt, M. (2011). Efficient calculation of pairwise nonbonded forces. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Chung, E., Fowers, J., Ovtcharov, K., Papamichael, M., Caulfield, A., Massengil, T., Burger, D., et al. Accelerating Persistent Neural Networks at Datacenter Scale. `https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/`.

Chung, E., Fowers, J., Ovtcharov, K., Papamichael, M., Caulfield, A., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., et al. (2018). Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38(2):8–20.

Cisco (2017). Forecast and Methodology, 2016–2021; White Paper; Cisco Systems. *Inc.: San Jose, CA, USA*.

Conti, A., VanCourt, T., and Herbordt, M. (2004). Flexible FPGA acceleration of dynamic programming string processing. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*.

DataCenterResearch (2019). Data Center Map. `www.datacentermap.com`. Accessed: 2019-08-29.

Digital Blocks (2019). Verilog / VHDL IP Cores for SoC, ASSP, ASICs and FPGAs. `www.digitalblocks.com`. Accessed: 2019-05-25.

Eastman, P. and Pande, V. (2010). OpenMM: a Hardware-Independent Framework for Molecular Simulations. *Computing in Science & Engineering*, 12(4):34–39.

ECP. Benchmarks. `https://github.com/ECP-CANDLE/Benchmarks`.

Endace (2019). DAG Packet Capture Cards. `www.endace.com/endace-high-speed-packet-capture-solutions/oem/dag/`. Accessed 2019-01-02.

EthernityNetworks (2019). SmartNICs. `www.ethernitynet.com/products/smartnics/`. Accessed 2019-01-03.

Forbes (2017). With The Public Clouds Of Amazon, Microsoft And Google, Big Data Is The Proverbial Big Deal. `www.forbes.com/sites/johnsonpierr/2017/06/15/with-the-public-clouds-of-amazon-microsoft-and-google-big-data-is-the-proverbial-big-deal`. Accessed: 2019-08-29.

Gautier, Q., Althoff, A., Meng, P., and Kastner, R. (2016). Spector: An OpenCL FPGA Benchmark Suite. In *Proceedings of the 2016 International Conference on Field-Programmable Technology (FPT)*.

Geng, T., Wang, T., Sanaullah, A., Yang, C., Xuy, R., Patel, R., and Herbordt, M. (2018a). FPDeep: A Framework for CNN Training Acceleration on FPGA Clusters. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*.

Geng, T., Wang, T., Sanaullah, A., Yang, C., Xuy, R., Patel, R., and Herbordt, M. (2018b). FPDeep: Acceleration and Load Balancing of CNN Training on FPGA Clusters. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Geng, T., Wang, T., Wu, C., Yang, C., Li, A., Song, S., and Herbordt, M. (2019a). LP-BNN: Ultra-low-Latency BNN Inference with Layer Parallelism. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors (ASAP)*.

Geng, T., Wang, T., Wu, C., Yang, C., Wu, W., Li, A., and Herbordt, M. (2019b). O3BNN: An Out-Of-Order Architecture for High-Performance Binarized Neural Network Inference with Fine-Grained Pruning. In *Proceedings of the International Conference on Supercomputing (ICS)*.

George, A. D., Herbordt, M. C., Lam, H., Lawande, A. G., Sheng, J., and Yang, C. (2016). Novo-G#: Large-Scale Reconfigurable Computing with Direct and Programmable Interconnects. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE.

Gidel (2018). Development Tools and IP. `http://gidel.com/development-tools-and-ip/`. Accessed: 2018-01-16.

Gidel (2019). Proc10A PCIe x8 (Gen.3) FPGA Computation Accelerators. `gidel.com/acceleration-platforms-2/proc10a-pcie-x8-gen-3-fpga-computation-accelerators/`. Accessed: 2019-05-24.

Gomperts, A., Ukil, A., and Zurfluh, F. (2011). Development and Implementation of Parameterized FPGA-based General Purpose Neural Networks for Online Applications. *IEEE Transactions on Industrial Informatics (IINF)*, 7(1):78–89.

Gu, Y. and Herbordt, M. (2007). FPGA-based multigrid computations for molecular dynamics simulations. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 117–126.

Gu, Y., VanCourt, T., and Herbordt, M. (2006a). Accelerating Molecular Dynamics Simulations with Configurable Circuits. *IEEE Proceedings on Computers and Digital Technology*, 153(3):189–195.

Gu, Y., VanCourt, T., and Herbordt, M. (2006b). Improved interpolation and system integration for FPGA-based molecular dynamics simulations. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 21–28.

Gu, Y., VanCourt, T., and Herbordt, M. (2006c). Integrating FPGA acceleration into the ProtoMol molecular dynamics code: Preliminary report. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Gu, Y., VanCourt, T., and Herbordt, M. (2008). Explicit design of FPGA-based coprocessors for short-range force computation in molecular dynamics simulations. *Parallel Computing (PC)*, 34(4-5):261–271.

Herbordt, M. (2013). Architecture/algorithm codesign of molecular dynamics processors. In *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, pages 1442–1446.

Herbordt, M., Gu, Y., VanCourt, T., Model, J., Sukhwani, B., and Chiu, M. (2008a). Computing models for FPGA-based accelerators with case studies in molecular modeling. *Computing in Science and Engineering*, 10(6):35–45.

Herbordt, M., Kosie, F., and Model, J. (2008b). An efficient $O(1)$ priority queue for large FPGA-based discrete event simulations of molecular dynamics. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 248–257.

Herbordt, M., Model, J., Sukhwani, B., Gu, Y., and VanCourt, T. (2006). Single pass, BLAST-like, approximate string matching on FPGAs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Herbordt, M., Model, J., Sukhwani, B., Gu, Y., and VanCourt, T. (2007a). Single pass streaming BLAST on FPGAs. *Parallel Computing (PC)*, 33(10-11):741–756.

Herbordt, M. and VanCourt, T. (2005). System and method for programmable logic acceleration of data processing applications and compiler therefore. United States Patent Application Publication. Pub. no.: US2007/0277161 A1.

Herbordt, M., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J., and DiSabello, D. (2007b). Achieving high performance with FPGA-based computing. *IEEE Computer*, 40(3):42–49.

Huang, Q., Lian, R., Canis, A., Choi, J., Xi, R., Brown, S., and Anderson, J. (2013). The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 89–96. IEEE.

Huawei (2019). FPGA-Accelerated Cloud Server. `www.huaweicloud.com/en-us/product/fcs.html`. Accessed 2019-01-03.

Humphries, B., Zhang, H., Sheng, J., Landaverde, R., and Herbordt, M. C. (2014). 3D FFTs on a Single FPGA. In *IEEE 22nd International Symposium Field-Programmable Custom Computing Machines*, pages 68–71.

Intel (2010). FFT IP Core User Guide. `www.altera.com/documentation/hco1419012539637.html`. Accessed: 2010-08-29.

Intel (2018a). Computing an FFT. https://software.intel.com/en-us/mkl-developer-reference-c-computing-an-fft.

Intel (2018b). FFT (1D) Design Example. www.altera.com/support/support-resources/design-examples/design-software/opencl/fft-1d.html. Accessed: 2018-01-16.

Intel (2018c). Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide. www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html. Accessed: 2018-07-30.

Intel (2019). Intel Arria 10 Transceiver PHY User Guide. www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/ug_arria10_xcvr_phy.pdf. Accessed: 2019-01-09.

Intel (2019). Intel FPGA SDK for OpenCL. www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html. Accessed: 2019-05-24.

Intel (2019). Intel High Level Synthesis Compiler. www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html. Accessed: 2019-09-10.

Intel (2019). Introduction to Intel FPGA IP Cores. www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_intro_to_megafunctions.pdf. Accessed: 2019-05-25.

Intel (2019). Low Latency Ethernet 10G MAC Intel FPGA IP User Guide. www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_32b_10g_ethernet_mac.pdf. Accessed: 2019-01-09.

Intel (2019). Platform Designer (formerly Qsys) Support. www.intel.com/content/www/us/en/programmable/support/support-resources/design-software/qsys.html. Accessed: 2019-08-18.

Jin, Z., Yoshii, K., Finkel, H., and Cappello, F. (2017a). Evaluation of the OpenCL AES Kernel using the Intel FPGA SDK for OpenCL. Technical report, Argonne National Laboratory.

Jin, Z., Yoshii, K., Finkel, H., and Cappello, F. (2017b). Evaluation of the Single-precision Floating Point Vector Add Kernel Using the Intel FPGA SDK for OpenCL . Technical report, Argonne National Laboratory.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-Datacenter Performance Analysis of a Tensor Processing Unit. *ACM/IEEE International Symposium on Computer Architecture*.

Karrenberg, R., Košta, M., and Sturm, T. (2013). Presburger Arithmetic in Memory Access Optimization for Data-Parallel Languages. In *International Symposium on Frontiers of Combining Systems*, pages 56–70. Springer.

Khan, M., Chiu, M., and Herbordt, M. (2013). Fpga-accelerated molecular dynamics. In Benkrid, K. and Vanderbauwhede, W., editors, *High Performance Computing Using FPGAs*, pages 105–135. Springer Verlag.

Khan, M. and Herbordt, M. (2012). Communication requirements for FPGA-centric molecular dynamics. In *Symposium on Application Accelerators for High Performance Computing*.

Koeplinger, D., Feldman, M., Prabhakar, R., Zhang, Y., Hadjis, S., Fiszel, R., Zhao, T., Nardi, L., Pedram, A., Kozyrakis, C., et al. (2018). Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 296–311, New York, NY, USA. ACM.

Krommydas, K., Helal, A. E., Verma, A., and Feng, W.-C. (2016). Bridging the Performance-Programmability Gap for FPGAs via OpenCL: A Case Study with Opendwarfs. Technical report, Department of Computer Science, Virginia Polytechnic Institute & State University.

LANL (2019). Los Alamos National Laboratory. `www.lanl.gov/`. Accessed: 2019-09-10.

Li, B., Tan, K., Luo, L. L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. (2016a). ClickNP-Bojie-Slides. `conferences.sigcomm.org/sigcomm/2016/files\\/program/sigcomm/Session01-Paper01-ClickNP-Bojie-Slides.pdf`. Accessed: 2019-05-25.

Li, B., Tan, K., Luo, L. L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. (2016b). ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14. ACM.

Mahram, A. and Herbordt, M. (2010). Fast and Accurate NCBI BLASTP: Acceleration with Multiphase FPGA-Based Prefiltering. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS)*, pages 73–82.

Mahram, A. and Herbordt, M. (2012a). CAAD BLASTP 2.0: NCBI BLASTP accelerated with pipelined filters. In *Proceedings 22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 217–223.

Mahram, A. and Herbordt, M. (2012b). FMSA: FPGA-Accelerated ClustalW-Based Multiple Sequence Alignment through Pipelined Prefiltering. In *Proceedings of the 20th International Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 177–183.

Mahram, A. and Herbordt, M. C. (2015). NCBI BLASTP on High-Performance Reconfigurable Computing Systems. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(4):33.

MathWorks (2018). HDL Coder. `www.mathworks.com/products/hdl-coder.html`. Accessed: 2018-03-28.

Maxeler (2019). MaxCompiler. `www.maxeler.com/products/software/maxcompiler`. Accessed: 2019-08-16.

Mellanox (2018). Smart Network Adapters - Overview. `http://www.mellanox.com/page/programmable_network_adapters`. Accessed 2018-10-16.

Mentor (2019). Catapult High-Level Synthesis. `www.mentor.com/hls-lp/catapult-high-level-synthesis/`. Accessed: 2019-08-16.

Microsemi (2019). Synphony Model Compiler ME. `www.microsemi.com/product-directory/dev-tools/4899-synphony`. Accessed: 2019-08-16.

Microsoft (2018a). Project Catapult Academic Program. `www.microsoft.com/en-us/research/academic-program/project-catapult-academic-program/#!training`. Accessed: 2018-12-27.

Microsoft (2018b). Project Catapult Academic Tutorial. `www.microsoft.com/en-us/research/video/project-catapult-academic-tutorial/`. Accessed: 2018-12-27.

MiTwell (2019). MiTwell Releases Intel Arria 10 FPGA SmartNIC. `www.mitwell.com.tw/news/news_4.htm`. Accessed 2019-01-03.

Model, J. and Herbordt, M. (2007). Discrete event simulation of molecular dynamics with configurable logic. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 151–158.

Molex (2018). SST DN4 DeviceNet* Network Interface Cards and USB Interface Module. `www.molex.com/molex/products/family?key=sst_dn4_devicenet_network_interface_cards_nics`. Accessed 2018-10-16.

Moura, J., Pueshel, M., Padua, D., and Dongarra, J., editors (2005). *Proceedings of the IEEE Special Issue: Program Generation, Optimization, and Platform Adaptation*. IEEE.

Myricom (2018). The Myricom ARC Series D-Class Network Adapter. A Low-cost, Low-latency Financial Trading Interface. www.cspi.com/ethernet-products/adapters/d-class/. Accessed 2018-10-16.

Nallatech (2018). Nallatech FPGA Accelerators support the Altera SDK for Open Computing Language (OpenCL). http://www.nallatech.com/solutions/fpga-accelerated-computing/opencl-software-bsps/. Accessed: 2018-01-16.

Napatech (2018). Napatech FPGA SmartNICs Enable Smarter Data Delivery to Your Network Management and Security Applications. www.napatech.com/products/napatech-smartnics/. Accessed 2018-10-16.

Netcope (2018). Netcope FPGA Boards. www.netcope.com/en/products/fpga-boards-(1). Accessed 2018-10-16.

NewWaveDV (2018). Programmable Network Interface Cards. newwavedv.com/products/function/programmable-network-interface-cards/. Accessed 2018-10-16.

Nikhil, R. (2004). Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications. In *Proceedings of the 2nd ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004 (MEMOCODE)*, pages 69–70. IEEE.

Nimbix (2019). Accelerate your workflows with Xilinx Alveo Accelerator Cards in the Cloud. www.nimbix.net/alveo/. Accessed 2019-01-03.

NVIDIA (2008). cuBLAS Library. https://docs.nvidia.com/cuda/cublas/index.html.

NVIDIA (2014). cuSparse Library. http://docs.nvidia.com/cuda/cufft.

NVIDIA (2018). cuFFT library. http://docs.nvidia.com/cuda/cufft.

OpenCores (2019). The Reference Community for Free and Open Source Gateware IP cores. opencores.org. Accessed: 2019-05-25.

OpenNetworkingFoundation (2019). OpenFlow Switch Specification. www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf. Accessed: 2019-05-24.

OpenPOWER (2019). SuperVessel OpenPOWER RD Cloud with Operation and Practice Experience Sharing. openpowerfoundation.org/blogs/supervessel-openpower-rd-cloud-with-operation-and-practice-experience-sharing/. Accessed 2019-01-03.

Ortigosa, E. M., Cañas, A., Ros, E., Ortigosa, P. M., Mota, S., and Díaz, J. (2006). Hardware Description of Multi-Layer Perceptrons with Different Abstraction Levels. *Microprocessors and Microsystems*, 30(7):435–444.

OVH (2019). FPGA accelerators on Public Cloud. `labs.ovh.com/fpga-accelerators-on-public-cloud`. Accessed 2019-01-03.

Panicker, M. and Babu, C. (2012). Efficient FPGA Implementation of Sigmoid and Bipolar Sigmoid Activation Functions for Multilayer Perceptrons. *IOSR Journal of Engineering*, pages 1352–1356.

Park, J., Qiu, Y., and Herbordt, M. (2009). CAAD BLASTP: NCBI BLASTP Accelerated with FPGA-Based Pre-Filtering. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*, pages 81–87.

Park, J., Qui, Y., and Herbordt, M. (2010). Caad blastn: Accelerated ncbi blastn with fpga prefiltering. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 3797–3800.

Pascoe, C., Lawande, A., Lam, H., George, A., Sun, Y., Farmerie, W., and Herbordt, M. (2010). Reconfigurable supercomputing with scalable systolic arrays and in-stream control for wavefront genomics processing. In *Proceedings of the Symposium on Application Accelerators for High Performance Computing (SAAHPC)*.

Patterson, D. A. (2004). Latency Lags Bandwith. *Communications of the ACM*, 47(10):71–75.

Pontarelli, S., Bifulco, R., Bonola, M., Cascone, C., Spaziani, M., Bruschi, V., Sanvito, D., Siracusano, G., Capone, A., Honda, M., et al. (2019). FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 531–548.

Portwell (2018). MIES-XHN5A10-NIC - FPGA Accelerator Card. `http://portwell.com/products/detail.php?CUSTCHAR1=MIES-XHN5A10-NIC`. Accessed 2018-10-16.

Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., et al. (2014). A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24.

Richter-Gottfried, F., Hain, S., and Fey, D. (2016). FPGA-aware Transformations of LLVM-IR. In *The First International Conference on Advances in Signal, Image and Video Processing*. ThinkMind.

Rodriguez-Donate, C., Botella, G., Garcia, C., Cabal-Yepez, E., and Prieto-Matías, M. (2015). Early Experiences with OpenCL on FPGAs: Convolution Case Study. In *Field-Programmable Custom Computing Machines*, pages 235–235.

Rucci, E., Garcia, C., Botella, G., De Giusti, A., Naiouf, M., and Prieto-Matias, M. (2017). Accelerating Smith-Waterman Alignment of Long DNA Sequences with OpenCL on FPGA. In *International Conference on Bioinformatics and Biomedical Engineering*, pages 500–511.

Sanaullah, A. and Herbordt, M. (2017). OpenCL for HPC/FPGAs: Case Study with 3D FFT. In *Proceedings of Heterogeneous High Performance Reconfigurable Computing (H2RC)*.

Sanaullah, A. and Herbordt, M. (2018a). FPGA HPC using OpenCL: Case Study in 3D FFT. In *Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*.

Sanaullah, A. and Herbordt, M. (2018b). Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolflow. In *IEEE High Performance Extreme Computing Conference (HPEC)*.

Sanaullah, A., Khoshparvar, A., and Herbordt, M. C. (2016a). FPGA-Accelerated Particle-Grid Mapping. In *International Symposium on Field-Programmable Custom Computing Machines*, pages 192–195. IEEE.

Sanaullah, A., Lewis, K., and Herbordt, M. (2016b). Accelerated Particle-Grid Mapping. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Sanaullah, A., Patel, R., and Herbordt, M. (2018a). An Empirically Guided Optimization Framework for FPGA OpenCL. In *Proceedings of the IEEE Conference on Field Programmable Technology*.

Sanaullah, A., Sachdeva, V., and Herbordt, M. (2018b). SimBSP: Enabling RTL Simulation for Intel FPGA OpenCL Kernels. In *Proceedings of Heterogeneous High Performance Reconfigurable Computing (H2RC)*.

Sanaullah, A., Yang, C., Alexeev, Y., Yoshii, K., and Herbordt, M. (2018c). Real-Time Data Analysis for Medical Diagnosis using FPGA Accelerated Neural Networks. *BMC bioinformatics*, 19(18):490.

SAVI (2019). Smart Applications on Virtual Infrastructure (SAVI). `www.savinetwork.ca/`. Accessed: 2019-08-21.

Sen, F., Hills, S., Kinaci, A., Narayanan, B., Davis, M., Gray, S., Sankaranarayanan, S., and Chan, M. (2017). Combining First Principles Modeling, Experimental Inputs, and Machine Learning for Nanocatalysts Design. *Bulletin of the American Physical Society*, 62.

Sharma, H., Park, J., Mahajan, D., Amaro, E., Kim, J. K., Shao, C., Mishra, A., and Esmaeilzadeh, H. (2016). From High-level Deep Neural Models to FPGAs. *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12.

Shen, J., Qiao, Y., Huang, Y., Wen, M., and Zhang, C. (2018). Towards a Multi-Array Architecture for Accelerating Large-Scale Matrix Multiplication on FPGAs. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE.

Sheng, J., Humphries, B., Zhang, H., and Herbordt, M. C. (2014). Design of 3D FFTs with FPGA clusters. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE.

Sheng, J., Xiong, Q., Yang, C., and Herbordt, M. (2016a). Collective Communication on FPGA Clusters with Static Scheduling. *Computer Architecture News (CAN)*, 44(4).

Sheng, J., Yang, C., Caulfield, A., Papamichael, M., and Herbordt, M. (2017). HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*.

Sheng, J., Yang, C., and Herbordt, M. (2015). Towards Low-Latency Communication on FPGA Clusters with 3D FFT Case Study. In *Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*.

Sheng, J., Yang, C., and Herbordt, M. (2016b). Application-Aware Collective Communication on FPGA Clusters. In *Proceedings IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Sheng, J., Yang, C., and Herbordt, M. (2018a). High Performance Dynamic Communication on Reconfigurable Clusters. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*.

Sheng, J., Yang, C., and Herbordt, M. (2018b). High Performance Dynamic Communication on Reconfigurable Clusters (Extended Abstract). In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Silicom (2018). Ultra Low Latency FPGA Server Adapters. www.silicom-usa.com/pr/server-adapters/programmable-fpga-server-adapter/ultra-low-latency-fpga-server-adapters/fb1cgv7580-fpga-card/. Accessed 2018-10-16.

Solarflare (2019). Ultra Low Latency FPGA Server Adapters. www.solarflare.com/software-defined-nics-adapters. Accessed 2019-01-02.

Stern, J., Xiong, Q., Sheng, J., Skjellum, A., and Herbordt, M. (2017). Accelerating MPI_Reduce with FPGAs in the Network. In *Proceedings of the Workshop on Exascale MPI*.

Stern, J., Xiong, Q., Skjellum, A., and Herbordt, M. (2018). A Novel Approach to Supporting Communicators for In-Switch Processing of MPI Collectives. In *Proceedings of the Workshop on Exascale MPI*.

Sukhwani, B. and Herbordt, M. (2008). Acceleration of a Production Rigid Molecule Docking Code. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 341–346.

Sukhwani, B. and Herbordt, M. (2010). FPGA Acceleration of Rigid Molecule Docking Codes. *IET Computers and Digital Techniques*, 4(3):184–195.

Sultana, N., Galea, S., Greaves, D., Wójcik, M., Shipton, J., Clegg, R., Mai, L., Bressana, P., Soulé, R., Mortier, R., et al. (2017). Emu: Rapid Prototyping of Networking Services. In *2017 USENIX Annual Technical Conference (ATC)*, pages 459–471.

Tarafdar, N., Lin, T., Fukuda, E., Bannazadeh, H., Leon-Garcia, A., and Chow, P. (2017). Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 237–246. ACM.

Taright, Y. and Hubin, M. (1998). FPGA Implementation of a Multilayer Perceptron Neural Network using VHDL. *IEEE International Conference on Signal Processing (ICSP)*, 2:1311–1314.

Telekom (2019). Faster computing: FPGA hardware acceleration for the Open Telekom Cloud. cloud.telekom.de/en/blog/open-telekom-cloud/fpga-closed-beta. Accessed 2019-01-03.

Tencent (2019). FPGA Cloud Server. cloud.tencent.com/product/fpga. Accessed 2019-01-03.

Tensorflow. How to Quantize Neural Networks with TensorFlow. https://www.tensorflow.org/performance/quantization.

Tripp, J. L., Peterson, K. D., Ahrens, C., Poznanovic, J. D., and Gokhale, M. B. (2005). Trident: An FPGA Compiler Framework for Floating-Point Algorithms. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 317–322. IEEE.

UCI (2018a). MNIST Data Set. `https://archive.ics.uci.edu/ml/databases/mnist/`.

UCI (2018b). Poker Hand Data Set. `https://archive.ics.uci.edu/ml/datasets/Poker+Hand`.

Ulutas, U., Tosun, M., Levent, V. E., Büyükaydın, D., Akgün, T., and Ugurdag, H. F. (2017). FPGA implementation of a dense optical flow algorithm using Altera OpenCL SDK. In *International Conference on ICT Innovations*, pages 89–101. Springer.

Vadatech (2018). Network Interface. `www.vadatech.com/category.php?catid_2=39\&catid_1=0`. Accessed 2018-10-16.

VanCourt, T., Gu, Y., and Herbordt, M. (2004). FPGA acceleration of rigid molecule interactions. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*.

VanCourt, T. and Herbordt, M. (2004). Families of FPGA-based algorithms for approximate string matching. In *International Conference on Application Specific Systems, Architectures, and Processors*, pages 354–364.

VanCourt, T. and Herbordt, M. (2005a). LAMP: A tool suite for families of FPGA-based application accelerators. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*.

VanCourt, T. and Herbordt, M. (2005b). Three dimensional template correlation: Object recognition in 3D voxel data. In *Proceedings of Computer Architectures for Machine Perception (CAMP)*, pages 153–158.

VanCourt, T. and Herbordt, M. (2006a). Application-dependent memory interleaving enables high performance in FPGA-based grid computations. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 395–401.

VanCourt, T. and Herbordt, M. (2006b). Rigid molecule docking: FPGA reconfiguration for alternative force laws. *Journal on Applied Signal Processing*, v2006:1–10.

VanCourt, T. and Herbordt, M. (2006c). Sizing of processing arrays for FPGA-based computation. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 755–760.

VanCourt, T. and Herbordt, M. (2007). Families of FPGA-based accelerators for approximate string matching. *Microprocessors and Microsystems*, 31(2):135–145.

VanCourt, T. and Herbordt, M. (2009). Elements of high performance reconfigurable computing. In Zelkowitz, M., editor, *Advances in Computers*, volume v75, pages 113–157. Elsevier.

VanCourt, T., Herbordt, M., and Barton, R. (2003). Case study of a functional genomics application for an FPGA-based coprocessor. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*, pages 365–374.

VanCourt, T., Herbordt, M., and Barton, R. (2004). Microarray data analysis using an FPGA-based coprocessor. *Microprocessors and Microsystems*, 28(4):213–222.

Wang, H., Soulé, R., Dang, H. T., Lee, K. S., Shrivastav, V., Foster, N., and Weatherspoon, H. (2017). P4FPGA: A Rapid Prototyping Framework for P4. In *Proceedings of the Symposium on SDN Research*, pages 122–135. ACM.

Wang, T., Geng, T., Jin, X., and Herbordt, M. (2019a). Accelerating AP3M-Based Computational Astrophysics Simulations with Reconfigurable Clusters. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors (ASAP)*.

Wang, T., Geng, T., Jin, X., and Herbordt, M. (2019b). FP-AMR: A Reconfigurable Fabric Framework for Block-Structured Adaptive Mesh Refinement Applications. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Wang, Z., He, B., Zhang, W., and Jiang, S. (2016). A Performance Analysis Framework for Optimizing OpenCL Applications on FPGAs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 114–125. IEEE.

Weerasinghe, J., Abel, F., Hagleitner, C., and Herkersdorf, A. (2015). Enabling FPGAs in Hyperscale Data Centers. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 1078–1086.

Weerasinghe, J., Abel, F., Hagleitner, C., and Herkersdorf, A. (2016a). Disaggregated FPGAs: Network Performance Comparison against Bare-Metal Servers, Virtual Machines and Linux Containers. In *8th IEEE International Conference on Cloud Computing Technology and Science*.

Weerasinghe, J., Polig, R., Abel, F., and Hagleitner, C. (2016b). Network-attached fpgas for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 36–43. IEEE.

Weller, D., Oboril, F., Lukarski, D., Becker, J., and Tahoori, M. (2017). Energy Efficient Scientific Computing on FPGAs using OpenCL. In *International Symposium on Field-Programmable Gate Arrays*, pages 247–256.

Xilinx (2018). Vivado High-Level Synthesis. `www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`. Accessed: 2018-03-28.

Xilinx (2019). SDNet PX Programming Language. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug1016-px-programming.pdf`. Accessed: 2019-09-12.

Xiong, Q., Bangalore, P., Skjellum, A., and Herbordt, M. (2018a). MPI Derived Datatypes: Performance and Portability Issues. In *Proceedings of the EuroMPI Conference*.

Xiong, Q. and Herbordt, M. C. (2017). Bonded Force Computations on FPGAs. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 72–75. IEEE.

Xiong, Q., Skjellum, A., and Herbordt, M. (2018b). Accelerating MPI Message Matching Through FPGA Offload. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications (FPL)*.

Xiong, Q., Yang, C., Patel, R., Geng, T., Skjellum, A., and Herbordt, M. (2019). GhostSZ: A Transparent SZ Lossy Compression Framework with FPGAs. In *Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM)*.

Yang, C., Geng, T., Wang, T., Patel, R., Xiong, Q., Sanaullah, A., Lin, C., Sachdeva, V., Sherman, W., and Herbordt, M. (2019a). FPGA Molecular Dynamics Simulations. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Yang, C., Geng, T., Wang, T., Sheng, J., Lin, C., Sachdeva, V., Sherman, W., and Herbordt, M. (2019b). Molecular Dynamics Range-Limited Force Evaluation Optimized for FPGA. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors (ASAP)*.

Yang, C., Sheng, J., Patel, R., Sanaullah, A., Sachdeva, V., and Herbordt, M. C. (2017a). OpenCL for HPC with FPGAs: Case Study in Molecular Electrostatics. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*.

Yang, C., Sheng, J., Patel, R., Sanaullah, A., Sachdeva, V., and Herbordt, M. C. (2017b). OpenCL for HPC with FPGAs: Case study in Molecular Electrostatics. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE.

Zhang, J., Zhang, Z., Zhou, S., Tan, M., Liu, X., Cheng, X., and Cong, J. (2010). Bit-Level Optimization for High-Level Synthesis and FPGA-based Acceleration. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 59–68. ACM.

Zhuo, L. and Prasanna, V. K. (2005). Sparse Matrix-Vector Multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*, FPGA '05, pages 63–74, New York, NY, USA. ACM.

Zipcores (2019). Zipcores - IP Cores for FPGA and ASIC platforms. `www.zipcores.com`. Accessed: 2019-05-25.

Zohouri, H. R., Maruyamay, N., Smith, A., Matsuda, M., and Matsuoka, S. (2016). Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 35:1–35:12, Piscataway, NJ, USA. IEEE Press.