Copyright
by
Aman Arora
2023

The Dissertation Committee for Aman Arora
certifies that this is the approved version of the following dissertation:

Optimizing FPGA Architecture for Deep Learning Workloads

Committee:

Lizy Kurian John, Supervisor

Vaughn Betz

Andreas Gerstlauer

Nur Touba

Earl Swartzlander

Alex Settle

Optimizing FPGA Architecture for Deep Learning Workloads

by

Aman Arora

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2023

Dedicated to my nephews and niece - Kanav, Kanisha and Saksham.

Acknowledgments

First and foremost, I must express my gratitude to my advisor, Prof. Lizy Kurian John, for her guidance and advice throughout my graduate school life. She coached me from a novice to an experienced researcher. I am thankful to her for the freedom and flexibility she gave me during my research, and her continuous support and help outside of research as well.

My heartfelt gratitude to Prof. Vaughn Betz from University of Toronto for collaborating with me, guiding me on FPGA architecture concepts, and allowing me to attend his class on FPGA architecture, which was fundamental for my research. I also want to thank professors Jaydeep Kulkarni, Pierre-Emmanuel Gaillardon, and Kenneth Kent, who collaborated with me and guided me in my research.

I would like to thank my committee members - Prof. Vaughn Betz, Prof. Andreas Gerstlauer, Prof. Nur Touba, Prof. Earl Swartzlander, and Dr. Alex Settle (from NVIDIA) - for their helpful comments and productive suggestions. I took classes with many of them and appreciate their lecturing and mentoring.

I am extremely thankful to Andrew Boutros from University of Toronto for his help and contributions in developing the Koios benchmark suite. Special thanks are due to my friend Supreet Jeloka who guided me with compute-in-

memory concepts that led to CoMeFa RAMs. I am also grateful to other graduate students who collaborated with me in my research - Seyed Alireza Damghani, Zhigang Wei, Siyuan Ma, Daniel Rauch, Aishwarya Rajen, Sangram Kate, Mohamed Elgammal, Rishabh Sehgal, and Bagus Hanindhito. Special thanks to undergraduate interns at LCA (Laboratory of Computer Architecture) who helped me with my research - Pragnesh Patel, Samidh Mehta, Moinak Ghosh, Aatman Borda, Tanmay Anand, Karan Mathur, Vedant Mohanty, and Atharva Bhamburkar.

I also want to give a shout-out to Qinzhe Wu from LCA. He helped me with logistics every step of the way (servers, bookings, and the like). I want to thank other LCA students for their help and support as well - Ruihao Li, Zachary Suskind, Ashen Ekanayake, Steffen Jensen, and Mugdha Jadhao.

I would also like to thank the National Science Foundation for funding my research, and the Graduate School at UT Austin for offering me a fellowship.

A lot of thanks are due to the staff at UT ECE - Melanie Gulick, David Korts, Barbara Hiene, Mary Matejka, Andrew Kieschnick, and Gabriel Hernandez.

Lastly, I want to thank my family for supporting me in starting my Ph.D. this late in my life, loving me, encouraging me, and believing in me all throughout.

Abstract

Optimizing FPGA Architecture for Deep Learning Workloads

Publication No. _____

Aman Arora, Ph.D.
The University of Texas at Austin, 2023

Supervisor: Lizy Kurian John

Deep Learning (DL) applications have tremendous computation requirements, making running them on traditional computers (CPUs) very inefficient. Modern computer systems deploy hardware acceleration, which involves offloading compute-intensive and memory-intensive tasks to specialized hardware. In the space of hardware acceleration alternatives, Field Programmable Gate Arrays (FPGAs) lie in the middle of the programmability-efficiency spectrum, with Graphic Processing Units (GPUs) being more programmable and Application Specific Integrated Circuits (ASICs) being more efficient.

FPGAs provide massive parallelism and are reconfigurable, which makes them very well suited for the fast-changing needs of DL applications. However,

the generic building blocks available on traditional FPGAs limit the acceleration that can be achieved. Hence, FPGAs trail ASICs by an order of magnitude in terms of performance. So, how can the gap between ASICs and FPGAs be minimized, while retaining the strength of FPGAs - the reconfigurability?

This dissertation describes research that aims to find the answer to this question by proposing new domain-optimized FPGAs for Deep Learning. The key idea is to integrate new hardware blocks to the FPGA that provide domain-specialized functionality, while still keeping them largely general and allowing them to be used with traditional FPGA flows. Specifically, new DL-optimized FPGAs containing blocks called Tensor Slices and CoMeFa RAMs are presented. The architecture of these blocks, along with the tradeoffs in exploring their architectures, is explained. Results show that significant performance improvement and energy reduction can be obtained for DL applications by using DL-specialized FPGAs containing these blocks. New benchmarks, called Koios, developed to explore FPGA architectures for DL are also explained. These benchmarks are open-sourced and work with VTR (an academic open source FPGA architecture exploration tool).

New DL-optimized FPGAs, containing Tensor Slices and CoMeFa RAMs, are significantly more efficient at accelerating DL workloads, while still being reconfigurable at a fine-grain. With the abundance of DL applications, making DL-optimized FPGAs is an attractive proposition.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Deep Learning and FPGAs

Deep Learning (DL) has become ubiquitous in today's world. DL is employed to perform tasks like image classification, natural language processing, sentiment analysis, speech-to-text, etc. Some common applications of DL include self-driving cars, virtual assistants, drug discovery, recommendation systems, etc. Owing to the enormous computation requirements of DL workloads, traditional computers (CPUs) are not very efficient at executing them. Therefore, hardware acceleration of DL workloads is commonplace. There are 3 main alternatives for accelerating DL workloads in hardware - Application Specific Integrated Circuits (ASICs), Field Programmable Gate Arrays (FPGAs) and Graphic Processing Units (GPUs). Table 1.1 compares the various alternatives for DL acceleration.

ASICs are chips designed to perform a specific type of processing very efficiently. Architecting, designing and fabricating an ASIC takes multiple years and is very costly. Programming an ASIC efficiently requires a new software stack to be built for every ASIC.

GPUs, on the other hand, are chips that were originally designed for the

	GPUs	ASICs	FPGAs
Generality	Turing-complete	Specific domain	Any custom HW
Architecture	Many cores/threads	Suits target domain	Spatial Architecture
HW Specialization	Fixed datapath & memory subsystem	Full flexibility	Reconfigurable
Power Consumption	High power	Most efficient	Moderate
Latency	Limited IOs + Fixed memory hierarchy	Custom IOs & HW	IOs + Custom HW
NRE Cost	Off-the-shelf	Very high	Off-the-shelf
Ease of Programming	Software (compilers & libraries)	Build your own stack	RTL / HLS / Overlays

Table 1.1: Comparing hardware acceleration alternatives for DL (Source: Vaughn Betz's slides at the tutorial on FPGA Architectures for Deep Learning at International Symposium on Microarchitecture 2022)

highly-parallel graphic workloads. However, in the last two decades, they have been deployed for multiple applications, including DL. This is because GPUs (sometimes also called GPGPUs - General Purpose GPUs) offer a powerful programming model that can be used to express parallel computation easily. However, GPUs are throughput-oriented machines and have high power consumption.

FPGAs lie somewhere in the middle of this efficiency-programmability spectrum. FPGAs are reconfigurable substrates that can be easily configured at the circuit level to perform any computation. FPGAs provide massive spatial parallelism, while being power-efficient compared to GPUs. They have significantly lower NRE (non-recurring engineering) costs compared to ASICs and, hence, a fast time-to-market. They exhibit low latency owing to (1) the custom memory hierarchies that can be easily built on them, and (2)

the presence of configurable I/O (input/output) ports. These ports can be used to efficiently connect an FPGA to the external world, e.g. sensors in an application. The challenge with FPGAs is that they are programmed using hardware description languages (at the register transfer level (RTL)), although other methods such as HLS (high level synthesis) and overlays (described later in this section) have become common in recent years as well. Nevertheless, the dominant ways of programming FPGAs still remains RTL and hence, hardware level expertise is needed to efficiently use them. FPGAs are extensively used for Deep Learning because they are very well suited for the rapidly changing world of DL.

There are 3 main paradigms used for DL acceleration using FPGAs:

1. By implementing a layer-specific accelerator on an FPGA. For example, an accelerator for convolution layers or an accelerator for GEMM (general matrix multiplication). The full DNN (Deep Neural Network) is executed on a CPU host, and specific layers are offloaded to the accelerator on the FPGA. This paradigm has a limitation that the cycles spent in transferring data back-and-forth between the CPU host and the FPGA can degrade performance.

2. By implementing a custom DNN-specific dataflow architecture on an FPGA. This provides high efficiency as the whole DNN is implemented in hardware. Different layers are unrolled differently to fit the DNN onto the FPGA. Such designs are typically generated using HLS (high level

synthesis) or using script-based generators and/or compilers. HPIPE [47] is an example of this style. This paradigm provides the best performance among the 3 paradigms because the hardware is specific to an application.

3. By implementing a flexible software-programmable processor with a custom instruction set on an FPGA. These processors are called overlays. They can execute different DNNs without the need to reconfigure the FPGA with a new bitstream. Microsoft Brainwave [37] architecture is an example of this style. This paradigm provides a software based approach to program the accelerator on the FPGA. However, the performance is reduced because the overlay is generic to support multiple DNNs.

1.2 Motivation and Problem Statement

Many works have compared the performance of FPGAs with ASICs and GPUs. Zhang et al. show that FPGAs exhibit 5.7× and 2× higher energy efficiency for VGG16 (a popular Deep Neural Network for image classification) compared to GPUs, at batch 1 and 16 respectively [138]. Hall et al. achieve a 2× lower latency at 1.4× higher throughput for Resnet50 (another popular Deep Neural Network for image classification) inference, using FPGAs compared to GPUs [46]. So, it can be seen that FPGAs provide higher performance and efficiency compared to GPUs for DL inference. Compared to GPUs, the challenge with FPGAs in that they are hard to program, because users have to write hardware descriptions using languages like Verilog, instead

of coding in a C-like programming language. The bodies of research working to develop High-Level Synthesis (HLS) and overlays (software programmable programs with custom instruction sets and hardware, configured onto FPGAs) are working towards solving this problem.

On the other hand, FPGAs lag behind ASICs in performance. Kuon and Rose [72] posit that FPGA designs have on average 17× more area, consume 7.1× more power, and are 3× slower compared to ASICs. This study uses 90 nm technology process, and uses benchmarks that are relatively small and do not make heavy use of hard blocks (DSP Slices and Block RAMs) that are common in modern designs. Nevertheless, these results imply an order of magnitude of difference in compute throughput between FPGAs and ASICs. Boutros et al. find that accelerators for CNNs (Convolutional Neural Networks) using FPGAs are ∼8× larger and ∼3-6× slower than ASIC accelerators. Nurvitadhi et al. compare the performance of LSTMs (Long Short Term Memory; a common DNN used for time series data such as speech and text) on FPGAs with ASICs, and find a ∼7× difference in performance per unit power. These observations bring up a question: Can the gap between the performance of FPGAs and ASICs for DL workloads be reduced? Finding the answer to this question is the motivation of the research work presented in this dissertation.

At their heart, FPGAs comprise fine-grained programmable logic blocks (LBs), embedded memory structures (RAM blocks), and fixed-function math units (DSP slices). A more detailed description of these blocks is provided

in Chapter 2. These generic building blocks make FPGAs flexible, but also limit the performance that can be achieved with FPGAs for domain-specific applications like DL. To reduce the gap between ASICs and FPGAs, domain-optimized FPGAs need to be designed. With the abundance of DL applications, making FPGAs that are optimized for the DL domain is an attractive proposition.

The problem statement this dissertation identifies is: How to architect DL-optimized FPGAs? What aspects of the FPGA architecture need to be changed to make them better DL accelerators?

1.3 DL Optimized FPGAs

As mentioned in the previous section, this dissertation presents DL-optimized FPGAs to reduce the efficiency gap between ASICs and FPGAs. To architect DL-optimized FPGAs, both the compute and the memory aspects of an FPGA are optimized.

Tensor or matrix operations are at the core of DL applications. These include matrix-matrix multiplication, matrix-vector multiplication, element-wise matrix additions and multiplications. Designing a matrix multiplier using the traditional compute blocks on an FPGA (i.e. logic blocks and DSP slices) does not result in an efficient implementation. Experiments conducted for this dissertation show that a 4x4 matrix multiplier on an FPGA is $\sim$4$\times$ slower and $\sim$10$\times$ larger than an ASIC in the 45 nm technology process. This is because multiple logic blocks and DSP slices have to be connected with each other

using the inefficient routing/interconnect. To perform matrix multiplication efficiently on an FPGA, a new hard block called Tensor Slice [7] [13] is added to FPGAs in this dissertation. These blocks are integrated into the FPGA fabric (a term commonly used to refer to the fine-grained programmable routing/interconnect) and connected to the rest of the FPGA like other blocks. These new blocks support the common matrix operations mentioned above. By doing this, an average speedup of 2.3× is observed across multiple DNNs, as shown later in this dissertation. Adding Tensor Slices to an FPGA results in a minor degradation of performance for non-DL applications (2.3% in Frequency and 7.7% in routed wirelength when 30% of the area of the FPGA is converted to Tensor Slices). However, DL-optimized FPGA families can be designed focused on DL applications, and non-DL applications can continue using non-DL optimized FPGAs.

FPGAs have hundreds of Megabits of on-chip memory in the form of Block RAMs. These in-fabric RAM blocks can be used to store DNN weights or activations. They provide high bandwidth to the compute units (logic blocks and DSP slices) via the programmable routing, thereby consuming power. Additionally, the bandwidth available is limited by the number of signals at the routing interface of the BRAM. To make DL-optimized FPGAs, compute capabilities are added to the BRAMs on FPGAs in this dissertation. This is done by adding bit-serial processing elements to the sense amplifiers inside the RAM block, and by exploiting the dual-ported nature of FPGA Block RAMs. These new blocks are called CoMeFa RAMs [9]. This reduces data movement through

the general-purpose routing, reduces the dependence on routing, increases the available bandwidth, and increases the compute throughput of the FPGA. For a low area overhead (3.8% for an FPGA with delay-optimized CoMeFa RAMs and 1.2% for an FPGA with area-optimized CoMeFa RAMs), a speedup of up to 2.6× is observed in multiple DNNs, as shown later in this dissertation. A negligible overhead (upto 2.3% in area, less than 0.5% in frequency, less than 0.5% in routed wirelength) is seen on non-DL applications.

1.3.1 Optimizing FPGAs for specific application domains

In this dissertation, DL-optimized FPGAs are proposed. Optimizing an FPGA for one application could seem antithetical to the idea of an FPGA at the first glance. However, creating FPGA families targeted towards large swaths of the application space is not uncommon. For example, Intel Stratix 10 AX family of FPGAs integrate high-performance data converters focused on RF applications. Lattice Semiconductor's CrossLink family of FPGAs is geared towards vision and video processing applications. It hardens MIPI interfaces and has higher DSP and BRAM to Logic Block ratio. Xilinx's recently announced Alveo SmartNIC products contain hardened blocks for network function acceleration.

While optimizing an FPGA for a specific application may not make sense, but optimizing an FPGA for large application domains does. DL is a very large application space, and is rapidly widening. In the future, almost every application will likely have DL components. Hence, developing families

of FPGAs optimized for DL is an attractive proposition.

A question arises: When optimizing FPGAs for an application domain, how much should the extent of specialization be? Introducing too much specialization for an application domain can be harmful because it reduces the generality of the FPGA significantly. At that point, the FPGA becomes more like an ASIC and loses its core strength - reconfigurability. The sweet spot in the generality-specialization spectrum is extremely difficult to determine. However, an FPGA architect does not have to define one FPGA architecture for an application domain. Instead, a family of FPGAs for that application domain is typically developed. Currently, all FPGA vendors offer FPGAs with different ratios of LBs : DSPs : BRAMs, and different numbers and sizes of other components such as hard memory controllers, configurable I/Os interfaces, etc. Therefore, for DL, this dissertation does not propose one FPGA architecture. It leaves out prescribing specific numbers/ratios of each new block to be added to the FPGA. Different FPGA chips with different proportions of various resources (LBs : DSPs : BRAMs : Tensor Slices : CoMeFa RAMs) can be created. A user can choose the specific FPGA they need from the vendor's catalog based on their application's needs, similar to how it is done currently. For example, if a healthcare related product is being developed and the majority of the application involves signal processing, but some part of the application involves Deep Learning, an FPGA with a higher ratio of DSPs and lower ratio of Tensor Slices would be the right choice.

Having said that, although the focus of this dissertation is DL, the op-

timizations proposed in this dissertation are not entirely DL specific. Other application domains can take advantage of these optimizations. Tensor Slices accelerate matrix operations like matrix-matrix multiplication and matrix-vector multiplication, which are common in HPC (high-performance computing) applications as well. Additionally, the Tensor Slice can be used for just multiplications or multiply-accumulate (using the individual PE mode; explained in Section 4.1.4). CoMeFa RAMs are not DL-specific blocks. They provide precision-agnostic bit-serial computation capabilities. They can be used for any application that suits a SIMD paradigm. For example, they can be used in signal processing and multimedia applications. In Section 5.3, results of using CoMeFa RAMs for some non-DL benchmarks, like FIR filter, are shown.

1.3.2 Distinction from CGRAs

A CGRA (Coarse Grain Reconfigurable Architecture) is a reconfigurable architecture that operates on coarser granularity than FPGAs. It contains an array of tiles, where each tile is either a processing element or a memory tile. CGRAs for Deep Learning have been developed [81] [112] [96]. When a CGRA for DL is designed, the software stack has to be built from scratch.

The work described in this dissertation - DL-optimized FPGAs - is significantly different from CGRAs for DL. In a DL-optimized FPGA, the majority of the chip still consists of traditional FPGA blocks that are programmable

at fine-grain. The key idea of designing a DL-optimized FPGA is to consume only a small portion of the area of the FPGA for new hardware blocks that provide DL-specialized functionality. Very importantly, the routing architecture remains the same as that of a traditional FPGA. There are no changes done to the configurable I/O blocks available on the FPGA as well. This allows a DL-optimized FPGA to be used with traditional FPGA design flows. A DL-optimized FPGA is programmed just like a traditional FPGA. Some updates are needed to tools to support the new blocks, but there is no need to rebuild the entire ecosystem - software stack, learning a new programming style, etc.

1.3.3 Distinction from DL-specific ASICs

Many DL-specific ASICs have been developed as well [65] [44] [45]. Developing such an ASIC requires significant investment in developing both the hardware as well as the software stack. However, once such an ASIC has been developed, it can be a serious contender when a user wants to accelerate a DL application, especially compared to using a DL-optimized FPGA.

DL-optimized FPGAs provide significant advantages over using DL-specific ASICs. DL-optimized FPGAs provide configurable I/O blocks that enable the chip to be connected in a variety of systems, which is not the case with DL-specific ASICs. In edge applications, the availability of configurable I/O blocks allows the DL-optimized FPGA to be easily connected to a variety of sensors.

A custom memory hierarchy matching the requirements of a future DL application can be easily created on a DL-optimized FPGA. Similarly, DL-optimized FPGAs allow for creating custom dataflows by connecting compute blocks (the new DL-specialized compute blocks as well as traditional compute blocks) in application-specific ways. Most of the DL-optimized FPGA still consists of Logic Blocks, which allows for customizing the control flow to a specific DL application. Such customization is not possible with existing DL-specific ASICs.

Most applications have only a part that involves DL. Common DL applications such as computer vision, natural language processing, healthcare, etc. are not performed in isolation, especially on the edge. There are other parts of the whole system-level application like gathering real-world data from sensors, data pre-processing, result post-processing, filtering, dynamic control, network functions, etc. These parts require hardware acceleration as well. If a DL-specific ASIC is used, only the DL portion of the application can be accelerated, and the rest of the application has to be executed on a host CPU, which is inefficient. However, using a DL-optimized FPGA perfectly fits such usecases. All parts of the application can be accelerated by the FPGA - the DL portions can take advantage of the DL-optimized hardware blocks and the non-DL portions can take advantage of hardware acceleration by using the traditional FPGA components.

1.4 Architecture Exploration for DL Optimized FPGAs

The process of FPGA architecture exploration is explained in Chapter 3. In this process, representative benchmarks play a vital role. It is essential that the benchmarks used capture the markets and application domains targeted by the candidate FPGA architecture. Using an unrepresentative set of benchmarks means optimizing for the wrong targets. However, existing open-source FPGA benchmark suites do not focus on (or even capture any) benchmarks from the DL domain. Existing suites either have small designs that are not representative of the modern applications, or do not have DL specific designs. So, a new benchmark suite called Koios [12] is also developed as a part of this dissertation. The Koios benchmark suite consists of 40 DL-specific designs, including original designs, designs re-created from prior works and proxy/synthetic benchmarks.

1.5 Thesis Statement

Integrating DL-specialized blocks such as matrix multipliers and compute capable RAMs in Field Programmable Gate Arrays (FPGAs) can provide increased acceleration for Deep Learning (DL) workloads.

1.6 Contributions of this Dissertation

In this section, the contributions of this dissertation are listed.

1. The first contribution of this dissertation is adding new blocks specialized

for matrix/tensor computations to FPGAs. The architecture, design and implementation of these new blocks, called Tensor Slices, is described. The impact of adding Tensor Slices on an FPGA for a variety of DL benchmarks is quantified, by comparing their performance to a contemporary FPGA. The degradation of various metrics such as frequency and routing wirelength for non-DL benchmarks when deployed on an FPGA with Tensor Slices is also studied. Additionally, the sensitivity of various metrics to the percentage of the FPGA area consumed by Tensor Slices is studied. It is observed that replacing DSP Slices on a modern FPGA with Tensor Slices results in a speedup of 2.3 $\times$ for common DNNs. Adding Tensor Slices to an FPGA results in a minor degradation of performance for non-DL applications (2.3% in Frequency and 7.7% in routed wirelength when 30% of the area of the FPGA is converted to Tensor Slices).

2. The second contribution of this dissertation is incorporating compute capabilities in Block RAMs on FPGAs. The architecture and operation of new compute-enabled BRAM blocks, called CoMeFa RAMs, is described. The versatility of CoMeFa RAMs is shown by mapping several applications with different workload characteristics, with a special focus on DL applications. Novel processing-in-memory hardware concepts are introduced while designing CoMeFa RAMs : a configurable processing element and exploiting dual ported'ness of RAMs to perform computation. A new way to program these RAMs using a stored program concept

is also illustrated. Novel processing-in-memory algorithmic concepts are introduced as well: one-operand-outside-ram (OOOR) operations. The performance and energy benefits of using CoMeFa RAMs are quantified for multiple microbenchmarks. It is observed that replacing Block RAMs on an FPGA with CoMeFa RAMs results in a speedup of up to 2.6× for common DNNs, for a low area overhead (3.8% for an FPGA with delay-optimized CoMeFa RAMs and 1.2% for an FPGA with area-optimized CoMeFa RAMs). A negligible overhead (upto 2.3% in area, less than 0.5% in frequency, less than 0.5% in routed wirelength) is seen in non-DL applications.

3. The third contribution of this dissertation is creating an open-source benchmark suite for FPGA DL architecture research. This new benchmark suite, called Koios, consists of 40 designs, and they work with the Verilog-To-Routing (VTR) framework, which is the most popular framework for FPGA architecture research. A framework for generating synthetic/proxy benchmarks with specific circuit characteristics is also introduced. Results of running Koios benchmarks through VTR using an FPGA architecture model developed to capture complex hard blocks typical of recent FPGAs are presented. Properties of these benchmarks are compared to the VTR and Titan benchmarks to highlight the added value of Koios. Example case studies that use these benchmarks to explore architecture and CAD optimizations for DL are shown.

1.7 Dissertation Organization

The remainder of this dissertation is organized as follows:

- Chapter 2 provides background information and describes related work.
- Chapter 3 provides an overview of the methodology used for evaluating the proposals.
- Chapter 4 presents Tensor Slices, new DL-specific compute blocks for DL-optimized FPGAs.
- Chapter 5 presents CoMeFa RAMs, new compute-enabled Block RAMs for DL-optimized FPGAs.
- Chapter 6 describes the Koios benchmark suite, a new suite to explore FPGA architectures for DL.
- Chapter 7 presents the summary and future work.

Chapter 2

Background and Related Work

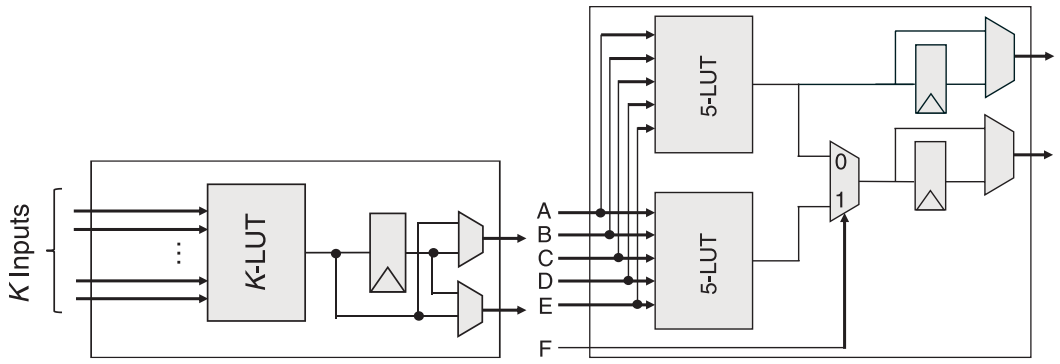
2.1 FPGA Architecture

FPGAs consist of a large number of functional blocks connected using a configurable interconnect/routing network. The main functional blocks in a modern FPGA are: Logic Blocks, DSP Slices and Block RAMs.

2.1.1 Logic Blocks

A Logic Block (LB) consists of a cluster of Basic Logic Elements (BLEs) along with local interconnect. A basic logic element (BLE), shown in Figure 2.1a, consists of a K-input Look-Up Table (LUT) coupled with a flip-flop and a bypass multiplexer. Common values of K are 4,5,6. A BLE can implement a logical function of K inputs using the K-input LUT with registered or unregistered output.

In modern FPGAs, each LUT in a BLE can be *fractured* into smaller LUTs to improve utilization. A fracturable K-input LUT can be configured as a single LUT of size K, or can be fractured into 2 LUTs of size up to $K-1$. Figure 2.1b shows an example. Also, instead of 1 flip-flop per BLE, modern FPGAs have two flip-flops per BLE to accommodate the increased demand



(a) Basic Logic Element (BLE) **(b)** A 6-input fracturable BLE

Figure 2.1: Block diagrams of logic elements (adapted from [19])

for flip-flops from highly pipelined designs.

The local interconnect in a Logic Block consists of multiplexers that allow feeding either the external inputs of the LB or the BLE outputs into BLE inputs. N is usually used to denote the number of BLEs in a LB, and usually ranges from 8-10. A Logic Block is shown in Figure 2.2.

Modern FPGA architectures include hardened arithmetic circuitry in their logic blocks as well. E.g., providing a 1-bit adder at the output of each BLE pair, and then connecting the carry-out of one adder to the carry-in of the next adder to form a carry chain along the logic block.

2.1.2 Programmable Interconnect

Most commercial FPGAs use an island-style architecture where the function blocks are arranged in a 2D array with programmable interconnect/routing between them. Island-style routing consists of 3 components: routing wire segments, connection blocks, and switch blocks. Figure 2.3 depicts

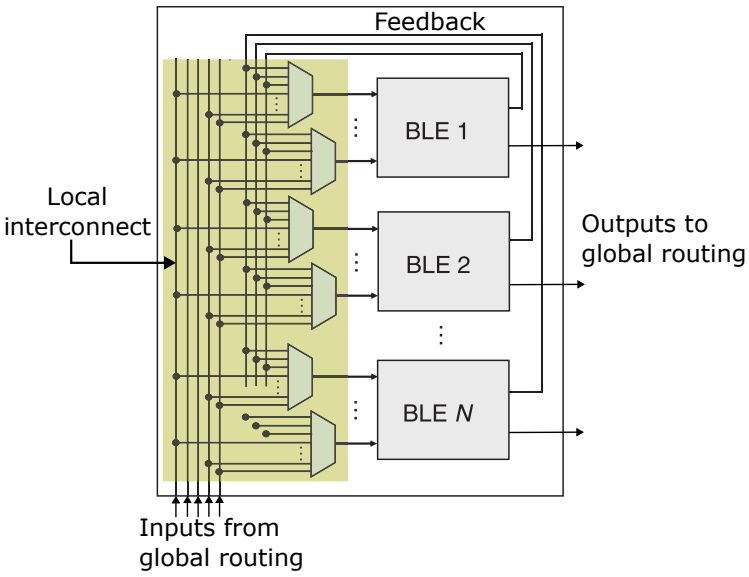


Figure 2.2: Logic Block architecture (adapted from [19])

this architecture. A connection block is a set of multiplexers that connect function block inputs to the routing wires. A switch block contains switches that connect routing wire segments together to realize longer routes. Outputs from function blocks are also fed into switch blocks. Wire segments of various lengths can be used. Parameters of the routing architecture of an FPGA include: number of routing wires each function block input or output can connect to (typically referred to as Fc), number of other routing wires each wire can connect to (typically referred to as Fs), the lengths of the routing wire segments, the routing switch pattern, and the number of routing wires per channel (typically referred to as W or channel width).

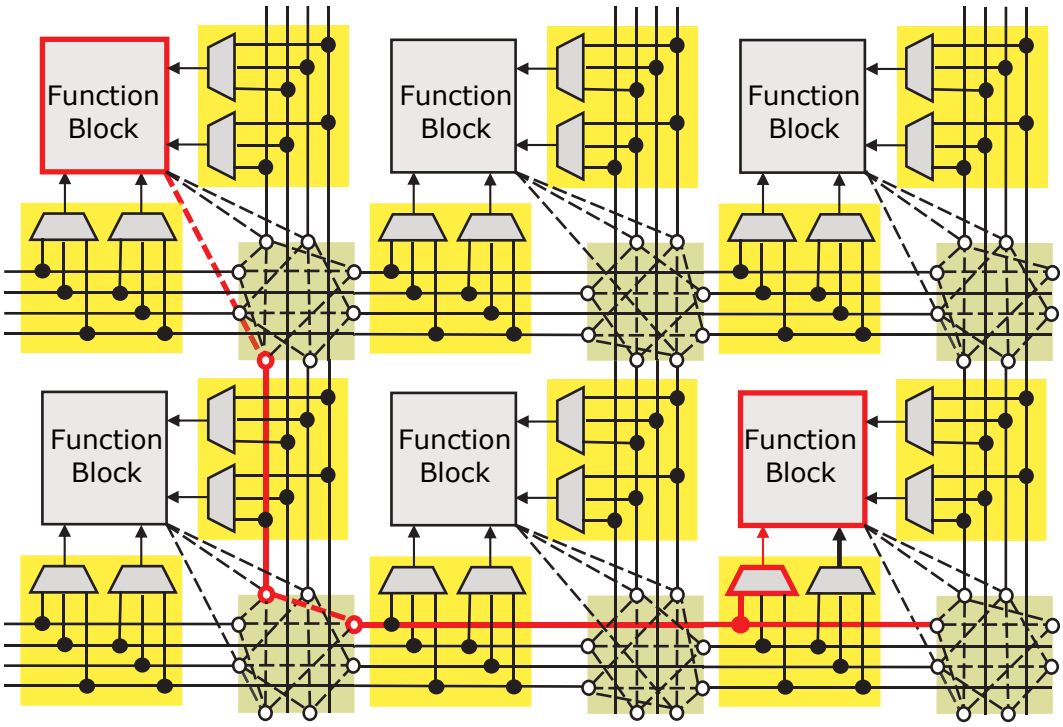


Figure 2.3: Island-style routing architecture (adapted from [19]). Thick solid lines are routing wires, while dashed lines are programmable switches. Connection and switch blocks are shaded in yellow and yellow-green, respectively.

2.1.3 Block RAMs

Block RAMs (BRAMs) are static RAMs that provide denser on-chip storage in FPGAs (denser compared to storing data in flip-flops in LBs). Just like an SRAM, a BRAM contains a memory cell array, along with peripheral circuitry that includes row decoders, column decoders, write drivers, sense amplifiers and bitline drivers. Typical sizes of BRAMs found in modern FPGAs are 18 Kilobits, 20 Kilobits and 36 Kilobits. They can be used to implement buffers, FIFOs, ROMs, register files, etc. FPGA BRAMs contain additional circuitry that allows them to be configurable to be used for diverse usecases,

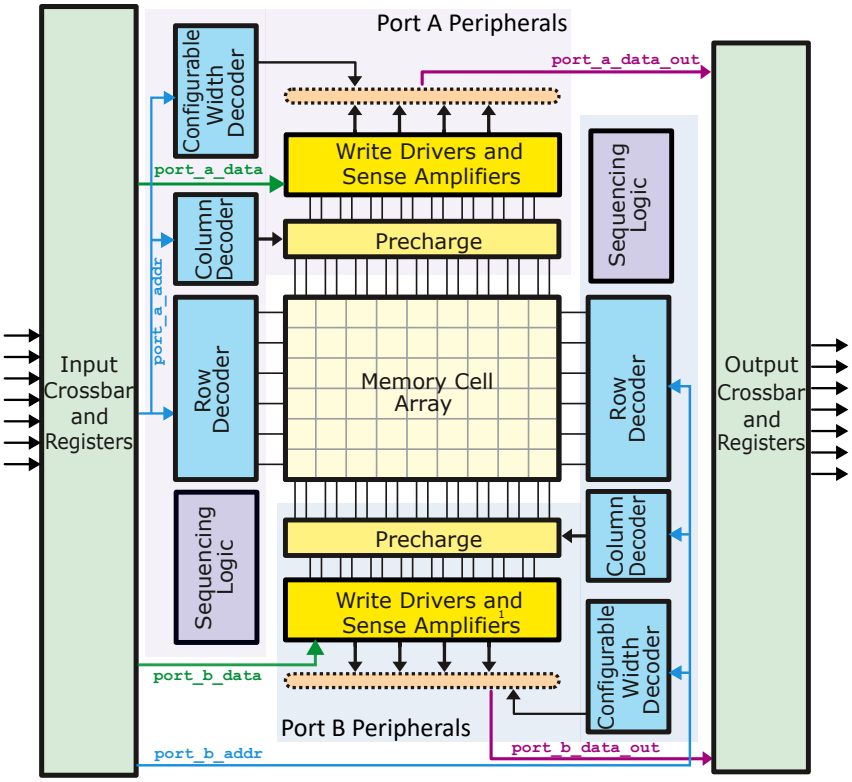


Figure 2.4: Block RAM architecture

and also to connect them to the programmable routing. See Figure 2.4 for details. The crossbars added to the inputs and outputs improve the routability of the block by making the pins of the block swappable when connecting to the programmable routing. BRAMs support single port (1 read or write port), simple dual port (1 read and 1 write port) and true dual port (2 read or write ports) modes. They can also be configured in different heights and widths. For example, a 20 Kilobit memory can be configured in 512x40, 1024x20 and 2048x10 geometries. This is achieved by adding a new block called the width configurability decoder into the RAM. BRAMs also interface with the routing

through connection blocks and switch blocks. More recently, a few large RAM blocks (hundreds of Kilobits in size) have been integrated into FPGA chips as well.

2.1.4 DSP Slices

DSP slices in modern FPGAs are an evolution of hard multipliers that were added because implementing multipliers in LBs was inefficient and multiplications are commonly used in signal processing, which is a major application domain for FPGAs. Today's DSP slices are much more than just multipliers. They also have input and output registers to enable high frequency operation. Instead of one multiplier, two or more multipliers may be present, followed by reduction logic (to add the results of the multipliers) or accumulation logic. Multipliers of varying precisions are typically supported. E.g. 18x18, 24x24, 25x18 and 27x27. These precisions are commonly used in signal processing applications. Floating-point multiplications and additions (32-bit IEEE Floating Point) are also supported in DSP slices in modern FPGAs. These are targeted towards supporting high-performance computing (HPC) applications. Larger multipliers can be fractured into multiple smaller multipliers. For example, the DSP slice could be configured to be used as one 27x27 multiplier or two 18x18 multipliers. Some vendors support a full ALU following the multiplier instead of just reduction or accumulation logic. Pre-adders (adders before the multipliers) are also available in some commercial DSP slices. These are useful in DSP filtering applications. Additionally, dedicated cascade inter-

connect between neighboring DSP slices is provided to create chains that are typically used to efficiently implement Finite Impulse Response (FIR) filters. Multiplexers are provided to bypass some stages, for example, bypassing the accumulation stage. A high level diagram of the architecture of a DSP Slice is shown in Figure 2.5.

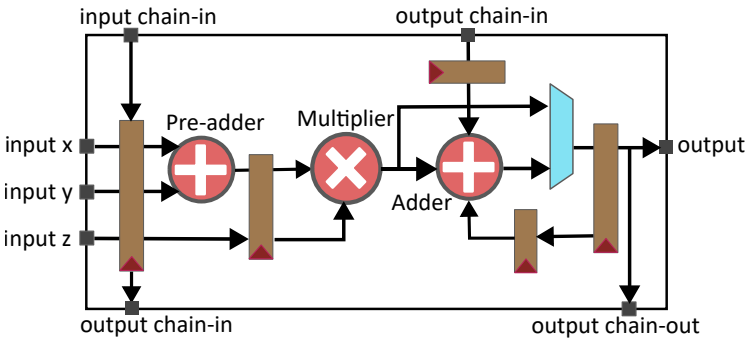


Figure 2.5: DSP Slice architecture

2.1.5 IO Blocks

FPGAs also include configurable input-output (IO) blocks that allow them to communicate with a wide variety of other devices. One set of physical IOs support many different IO interfaces and standards. FPGA IOs use buffers that can operate across a range of voltages. IOs are implemented with multiple pull-up/pull-down circuits. In addition to electrical and timing configurability, FPGA IO blocks contain additional hardened digital circuitry e.g. a capture register, double to single rate conversion registers (used with DDR memories) and serial-to-parallel converters. For implementing high speed serial protocols, FPGAs include separate differential IO blocks that can be used

as serial transceivers.

2.2 DL Acceleration and FPGAs

In the last decade, many architectures have been proposed and deployed to meet the ever-increasing compute and memory demands of DL workloads. The Google TPU v1 [65] is an ASIC accelerator for DL applications, implementing a large 256x256 systolic array based matrix multiplication engine supporting int8 precision. In later generations of the TPU, Google has added support for more precisions and reduced the size of the matrix multiplier and increased the number of matrix multipliers on a chip. There are many other ASIC chips for DL from companies like Sambanova, Habana, Cerebras, Graphcore, and Groq. Designing these dedicated ASIC chips for DL takes a long time. Not only the chips, the entire software stack has to be rebuilt as well. Therefore, the time-to-market and the cost is very high. Also, these chips can not be easily interfaced with sensors and deployed in an edge scenario. These chips can only execute DL. However, most applications include a portion that is DL, but there are other parts that may not be DL specific. So, these portions of the application have to be executed on a host CPU. Additionally, being dedicated chips, they do not allow a user to customize the dataflow required for their specific (futuristic) application at the hardware level, limiting their benefit.

NVIDIA Volta GV100 GPU [92] adds specialized cores called Tensor Cores to the Streaming Multiprocessors (SMs) on the GPU. Each Tensor Core

provides a 2D processing array that performs the matrix multiplication operation on 4x4 matrices in fp16 precision. Different sizes of matrix units and more precisions have been added by NVIDIA in later versions of these architectures. The main advantage of using GPUs is that they provide a software approach of programming them. For example, NVIDIA GPUs provide the CUDA programming language and model. However, GPUs are very power hungry and usually used for training scenarios. GPUs are not a good fit for edge inference scenarios.

FPGAs offer many advantages in accelerating DL applications primarily because of their reconfigurability - the ability to customize the computation architecture, variable precisions, low NRE (non-recurring engineering) costs, low time to market, and easy integration into systems using configurable I/O. FPGAs provide for spatial parallelism, with tightly integrated on-chip memory blocks. An instruction stream is not required to be decoded to perform computation. This leads to low energy and low latency, especially for edge inference applications. Many FPGA based solutions exist for DL acceleration. Microsoft's Brainwave [37] is a soft NPU (Neural Processing Unit) with dense matrix-vector multiplication units at its heart, implemented on Intel's Stratix 10 FPGAs. Intel's DLA [1] uses a 1-D systolic processing element array to perform dot product operations commonly required in neural networks. Xilinx's xDNN [131] is an overlay processor, containing a systolic array based matrix multiplier, that can be implemented on Xilinx FPGAs.

There are many accelerators that are designed using OpenCL, instead

of an HDL [110] [139] [119]. Accelerators leveraging the flexibility of FPGAs to accelerate sparse workloads have also been proposed [107] [83]. In addition to accelerators themselves, there are many frameworks to generate efficient accelerator designs automatically. DNNWeaver [104] and DNNBuilder [140] are open-source frameworks that generate the Verilog code for an accelerator specialized for a specified network. Verigood-ML [36] presents an automated methodology for generating Verilog with no human in the loop, starting from a high-level description of a DNN in a standard format such as ONNX. The Verilog RTL can then be implemented on an FPGA.

Highly quantized DNNs, such as Binary Neural Networks or Ternary Neural Networks, have been proposed to reduce the computation requirements and memory footprint of DNNs. FPGAs provide bit-level programmability making it possible to implement such quantized DNNs efficiently. LUTNet [120] and LogicNets [115] use LUTs as processing elements for performing inference. FINN [116] and HLS4ML [89] are frameworks for building FPGA accelerators for quantized DNNs.

While most of the research on FPGAs is targeted towards using FPGA based accelerators for DNN inference, some recent works have performed training on FPGAs as well [141] [82] [39] [70]. Training requires large amounts of data to be accessed from DRAM, and typically FPGAs have less DRAM bandwidth compared to GPUs. The other advantage of FPGAs is the low latency and low energy you can get compared to GPUs. But those are typically not the goals when training a DNN. Training is better done on something that has

Table 2.1: Existing DL-specific FPGA Architecture Optimizations

Method	Proposal	Reference
Changes to Logic Blocks	Extra adder chain	[20][33]
	Adding more adders in the existing carry chain	[20][33]
	Shadow multipliers in a logic block	[20][33]
	Additional 6-input XOR gate	[99]
Changes to DSP Slices	Low precision support	[18] [98]
	Semi-2D chaining capability	[98]
	Adding a register file for data reuse	[98]
Changes to Block RAMs	Compute capable BRAMs	[124]
New in-fabric blocks	Intel AI Tensor Block	[75]
	Achronix Machine Learning Processor	[2]
New out-of-fabric blocks	Xilinx AI Engines	[132]
New out-of-die blocks	Intel Tensor Tile	[91]
	Intel Tensor RAM	[90]

a more software-ish interface (like a GPU) so one can run iterations quickly.

2.3 DL-specific FPGA Architecture Optimizations

The FPGA based solutions mentioned above use traditional FPGAs to accelerate DL. They do not modify the architecture of the FPGA itself. The FPGA industry has deployed many DL-specific modifications to the FPGA architecture in recent years. Several academic research ideas to enhance FPGA architecture for DL have also been proposed. Table 2.1 lists these proposals. In this section, an overview of these proposals is provided.

2.3.1 Changes to Logic Blocks

Eldafrawy et al. [33] propose changes to the LB architecture to increase the compute throughput for mac (multiply accumulate) operations. They ex-

plore 3 improvements to the LB architecture: (1) adding an extra adder chain into a logic block to enable more efficient adder compressor trees, (2) fracturing 4-input LUTs to 3-input LUTs to increase utilization and adding adders in the existing carry chain, and (3) incorporating a shadow multiplier in LBs (shown in Figure 2.6). Improvement #1 achieves 13-16% area reduction and 10-13% frequency improvement across multiplication, mac and matrix multiplication benchmarks. Improvement #2 leads to 1.5× area reduction in matrix multiplication benchmarks, and up to 10% speedup for several arithmetic-heavy benchmarks, for only a 3% die area increase. Improvement #3 with 9-bit shadow multiplier results in a die area overhead of 12%, but leads to a 2.4× increase in density and 17% speedup on matrix multiplication benchmarks.

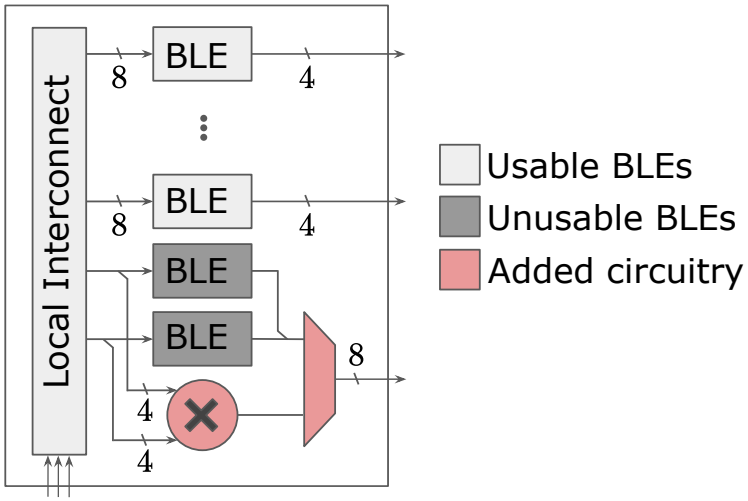


Figure 2.6: Logic Block with a shadow multiplier proposed by Eldafrawy et al. (adapted from [33])

Rasoulinezhad et al. [99] propose augmenting the BLEs in Logic Blocks with a 6-input XOR tree. This can enable efficient mapping of compressor trees

onto FPGA logic blocks. Compressor trees are commonly used in adder implementations. Across many benchmarks they consider, 35% of the compressors are C6:111, which maps efficiently to their enhanced Logic Block. The silicon area overhead of their enhancement is less than 0.5%, and the frequency degradation is 1-6%. They observe an average reduction of 13-19% in logic utilization on micro-benchmarks from a variety of domains. Additionally, the majority of the operations in a Binary Neural Network (BNN) involve the Xnor-Popcount operation. Compressor trees can be used for this operation. Using their enhanced Logic Block, an average reduction of 37-47% in area is seen for BNNs.

2.3.2 Changes to DSP Slices

Use of reduced precisions in DL inference tasks has been shown to improve performance and reduce the memory footprint of the network. FPGAs can be used to create custom bit-width datapaths using LBs. But the DSP slices in most FPGAs do not support precisions below 18-bit. Recent FPGAs like Intel Agilex [53] and Xilinx Versal [40] have introduced native support for low precisions like int8, fp16 and bfloat16 DSP slices. Low precisions are commonly used in DNN inference.

Boutros et al. suggest strengthening DSP blocks by efficiently supporting low-precision multiplications [18]. This is shown in Figure 2.7. Their enhanced DSP slice efficiently packs 2× as many 9-bit and 4× as many 4-bit multiplications compared to a baseline Intel Arria-10 like DSP block. The

cost of these enhancements is only 12% area overhead at the block level, and only 0.6% at the overall die area. The enhanced DSP slice improved the performance of 8-bit and 4-bit CNN accelerator designs by 1.32× and 1.6× and reduced the utilized chip area by 15% and 30% respectively.

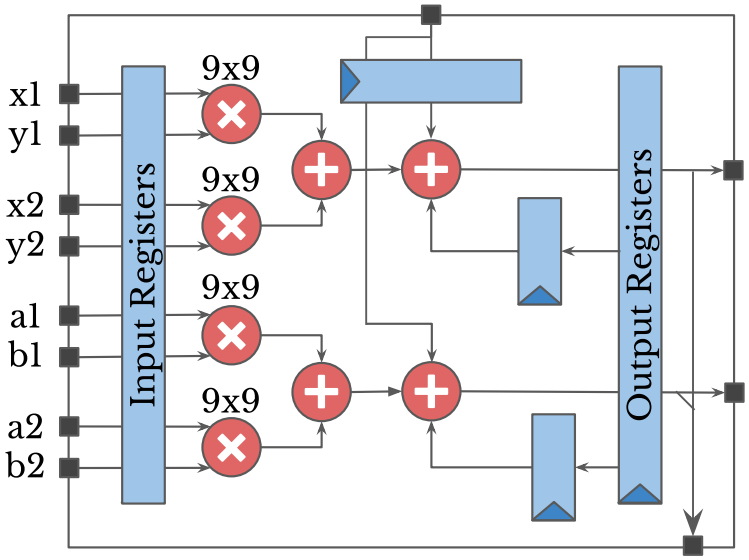


Figure 2.7: DSP Slice supporting low precision proposed by Boutros et al. (adapted from [18])

Rasoulinezhad et al. [98] propose 3 changes to the DSP blocks to improve support for multi-precision DNNs: (1) a flexible precision, run-time decomposable multiplier architecture, (2) a semi-2D chaining capability that supports the low-precision multiplier, and (3) adding a register file for data reuse. Their DSP block is called Precision, Interconnect, and Reuse optimized DSP (PIR-DSP). Compared with a baseline non-DL optimized DSP, the PIR-DSP offers a 6× improvement in mac operations per DSP in the 9×9-bit case, 12× for 4×4 bits, and 24× for 2×2 bits. Additionally, the PIR-DSP

decreases the run time energy to 31/19/13% of the baseline value, in a 9/4/2-bit MobileNet-v2 DNN implementation.

2.3.3 Changes to Block RAMs

Wang et al. propose Compute Capable BRAMs (CCB) [124]. They propose adding compute capabilities into block RAMs (BRAMs) to increase the computational throughput of FPGAs and reduce interconnect usage. Details are provided in Section 2.4.

2.3.4 New in-fabric blocks

Intel's Stratix 10 NX FPGAs have in-fabric AI tensor blocks [75], instead of DSP slices. These blocks have 30 MACs and 15× more int8 (8-bit fixed point) compute than a Stratix 10 DSP slice (7.5× compared to an Intel Agilex DSP slice). The base precisions are int8 and int4, along with shared exponent support for block floating point fp16 and fp12. All additions/accumulations can be done in int32 or IEEE single precision floating point (fp32), and multiple blocks can be cascaded together to support larger matrices. Stratix 10 NX chip achieves 143 int8/fp16 TOPs/FLOPs, or 286 int4/fp12 TOPs/FLOPs at 600MHz. The architecture of the Intel Tensor Block in the int8 precision mode is shown in Figure 2.8.

Achronix Speedster7t FPGAs [2] have embedded machine learning processor (MLP) blocks that have an array of multipliers, an adder tree and accumulator, integrated with a memory. Each MLP supports 4x int16, 16x int8

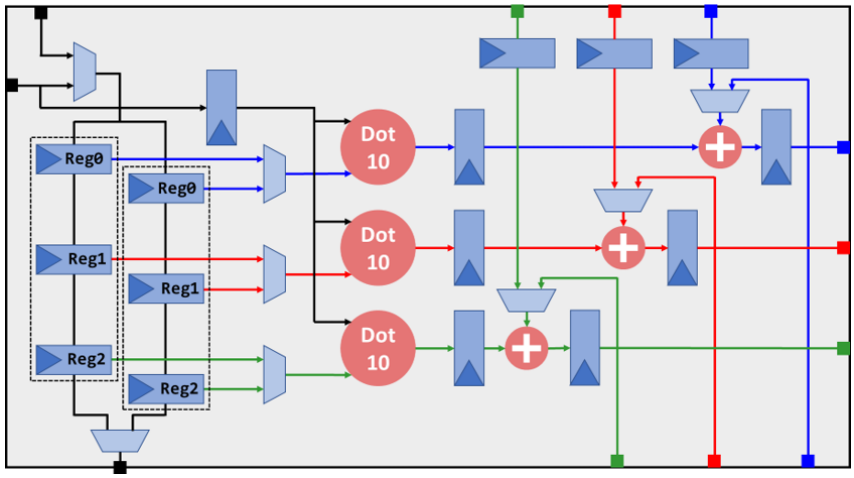


Figure 2.8: Intel's AI Tensor Block shown in the int8 mode. There are 3 dot product units with 10 multipliers each. (from [21])

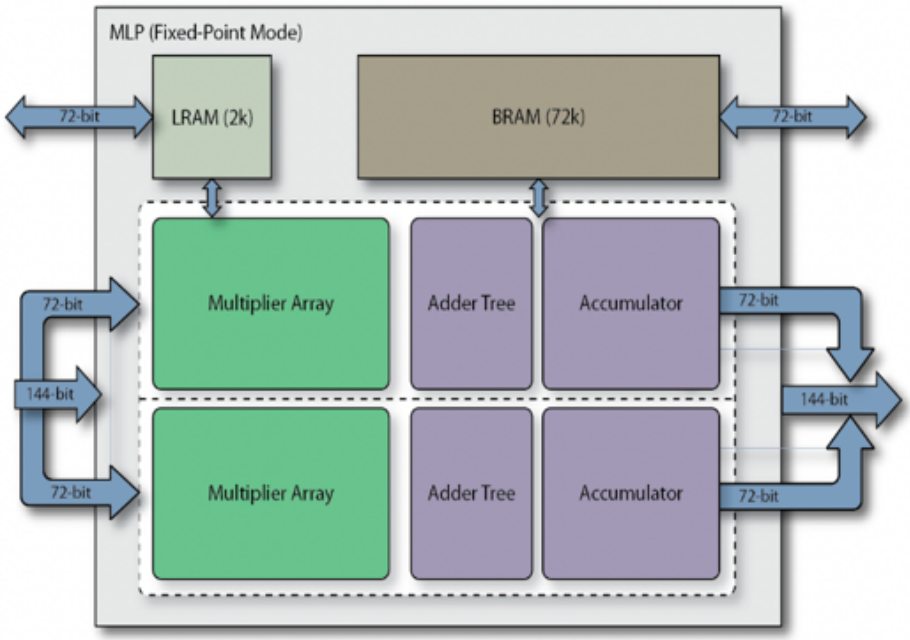


Figure 2.9: Achronix's Machine Learning Processor (MLP) (from [2])

or 32x int4 multiplications. Multiple floating point precisions including fp16, fp24, bfloat16 and block floating-point (BFP) are also supported. Each MLP also incorporates two memories that are closely coupled to the multipliers and accumulator. One memory is a large, dual-port, 72-Kilobit RAM, and the other is a 2-Kilobit cyclic buffer (for reuse). Hard paths between memory and other MLP blocks enable high-performance while freeing up general-purpose routing. A high level architecture of the Achronix MLP is shown in Figure 2.9.

In this dissertation, new in-fabric blocks called Tensor Slices are described, which add hardened support for matrix operations like matrix-matrix multiplication, matrix-vector multiplication and elementwise operations like addition and multiplication. No existing DL-optimized FPGAs propose adding in-fabric blocks that can perform matrix operations directly in hardware. For example, Intel's AI Tensor Blocks support the dot product operation, and Achronix's MLP leaves a compute operation like matrix multiplication to be mapped to it through the user's design. Additionally, Tensor Slices support common DL-specific floating point precisions like fp16 and bf16, whereas Intel's AI Tensor Blocks only support block floating point formats. Tensor Slices and Intel AI Tensor Blocks are compared quantitatively in later in this dissertation in Chapter 4. In the same chapter, Tensor Slices are also compared qualitatively with Xilinx AI Engines (mentioned in the next section).

2.3.5 New out-of-fabric blocks

Xilinx's Versal Adaptive Compute Acceleration Platform (ACAP) family [40] adds AI engines [132] on the same die as the programmable logic (also commonly referred to as the FPGA fabric). The AI engines are laid out as 2D array of tiles, each containing a VLIW (Very Large Instruction Word) vector processor, a scalar processor, and tightly integrated memory. There is a NoC (Network on Chip) based interface to connect the AI engines to the other resources on the chip, e.g. the FPGA fabric. The AI engines are software programmable (C/C++), instead of being programmed using HDLs (Hardware Description Languages) like Verilog. Figure 2.10 shows a high level architecture of the AI engines.

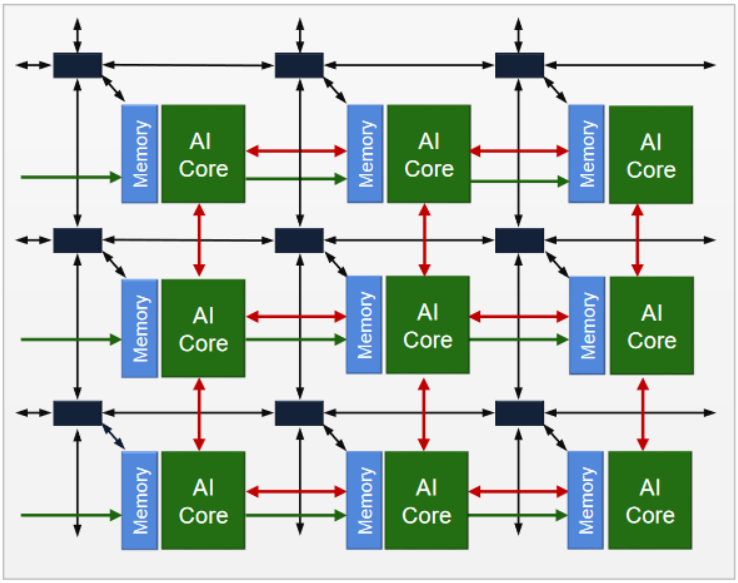


Figure 2.10: A high level architecture of Xilinx AI Engines (from [132]). Black rectangles = Interconnect switches, Red arrows = Memory interface, Black arrows = Stream interface, Green arrows = Cascade interface

2.3.6 New out-of-die in-package blocks

Chiplet technology involves connecting multiple dies using high-bandwidth interfaces and packaging them as one chip. The various dies can be ASIC chiplets, transceiver chiplets, memory chiplets or FPGA chiplets. A high level diagram is shown in Figure 2.11. For example, Intel's FPGAs integrate heterogeneous ASIC tiles using EMIB (Embedded Multi-Die Interconnect Bridge) interface in a System-In-Package (SIP). An example of this is integrating domain-specific accelerator tiles called TensorTiles with an FPGA die in the same package [91]. The TensorTile supports key matrix/vector multiply/accumulate operations with low precision support, including int4. This approach does not change the FPGA fabric (allowing reuse of existing ecosystems), and allows freedom in ASIC design (area/freq/etc. unconstrained by the FPGA fabric). A 4× improvement in performance of AlexNet is observed with 2 TensorTiles connected to an Intel Stratix 10 FPGA. Another example is integrating ASIC chiplets called TensorRAM [90] with an FPGA in an SIP to enhance on-chip memory bandwidth. The TensorRAM chiplet contains low-latency register files (RF) and SRAMs as well as a number of near-memory compute units matching the memory bandwidth. The compute units are 64-wide int8 dot product engines (DPEs) that can be configured as 128-wide int4 or 256-wide ternary or 512-wide binary DPEs. The results show that a small Stratix 10 FPGA with a TensorRAM (int8) offers 15.9× better latency than a GPU (using fp32) and 34× higher energy efficiency for several RNN, GRU and LSTM workloads.

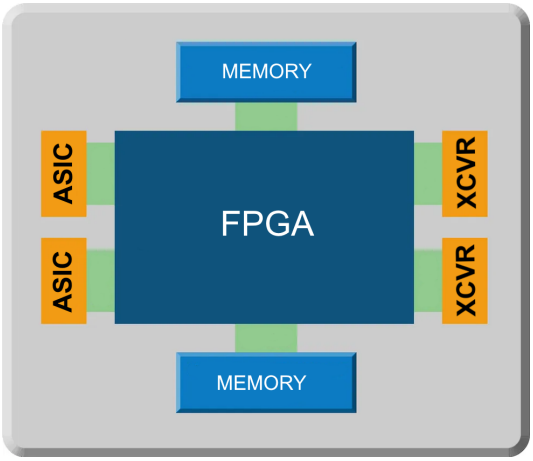
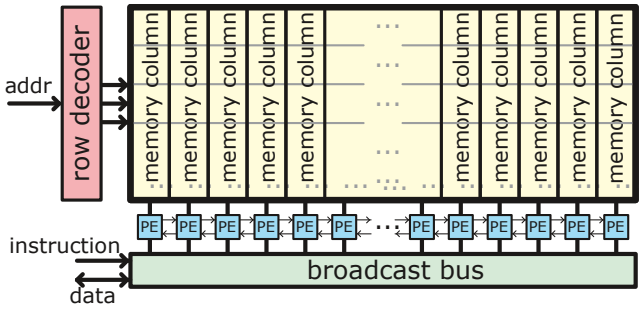


Figure 2.11: FPGA dies can be integrated with ASIC, XCVR (transceiver) and memory tiles to create a SIP (System In Package)

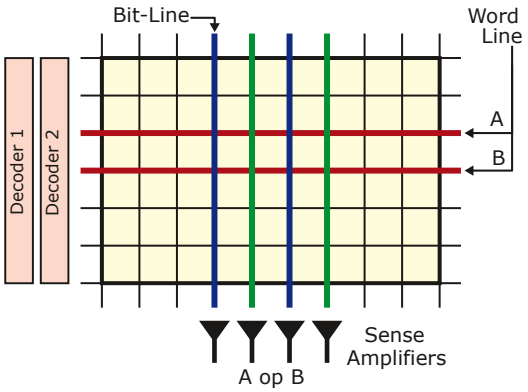
2.4 Compute-In-Memory

Compute-In-Memory or Processing-In-Memory (PIM) [43] is the paradigm of bringing computation closer to the data, instead of moving data to distant compute units. Many accelerators using PIM have been proposed and deployed: ReRAM based [103] [27] [50], DRAM based [80] [102] [41], and SRAM based [68] [4] [67] [121].

Computational RAM (or C-RAM) [35] is an architecture where a row of processing elements (PEs) is added to a memory (DRAM or SRAM) to convert it into a SIMD processor, as shown in Figure 2.12a. Each PE is pitch-matched with a memory column (bitline). An instruction is received by the memory from the host, operand rows (wordlines) are read and stored in the PEs, the operation is then performed, and the results are stored back into a row. All PEs in a memory execute the same instruction in a cycle. This is



(a) Computational RAM



(b) Logic-in-memory

Figure 2.12: Two approaches for compute-in-memory

shown to achieve significant speedup for applications like image processing, databases, computer-aided design, etc.

Jeloka et al. [64] created a logic-in-memory SRAM prototype shown in Figure 2.12b. where multiple word lines are activated simultaneously and the shared bitlines can be sensed, effectively performing logical AND and NOR operations on the data stored in the activated wordlines. This technology is deployed on CPU caches to transform them into parallel processing engines

[3], leading to speedups in many applications involving operations like word count, string match, etc. In Neural Cache, Eckert et al. apply this technology to DL applications [32], adding processing elements to the sense amplifiers and deploy bit-serial compute to perform DL operations.

Wang et al. proposed integrating the technology from Neural Cache into FPGA BRAMs to create Compute Capable BRAMs (CCB) [124]. Speedup is shown for recurrent neural networks (vanilla RNN, LSTM, and GRU), for int8 and 8-bit block floating-point precisions. To avoid data corruption because of multi-row access, the wordline voltage (and hence the frequency of operation) has to be lowered significantly. An additional row decoder is required for multi-row access. Additionally, in this architecture, one sense amplifier for each pair of bitlines (BL/BLB) is replaced with two sense amplifiers (one with BL/Vref and another with BLB/Vref). The complexity associated with these changes to the memory array makes this architecture not very practical to implement on a large scale.

In this dissertation, CoMeFa RAMs are described. These are compute-capable SRAMs, specifically targeted for FPGAs. CoMeFa RAMs exploit the dual-ported'ness of FPGA BRAMs instead of activating multiple wordlines to access two operands and perform computation between them. No existing work proposes exploiting the dual-ported'ness of FPGA BRAMs. For example, CCB (based on the same technology as Neural Cache) proposes activating multiple wordlines at the same time, leading to robustness issues. Additionally, CoMeFa RAMs have more feature-rich processing elements, deploy novel al-

gorithms (e.g. One Operand Outside RAM (OOOR) operations), and can be programmed easily (e.g. using a stored program instead of hardcoded FSM). Two architectures of CoMeFa RAMs with different area delay tradeoffs are presented. More details of these and quantitative comparison between CCB and CoMeFa RAMs is presented later in this dissertation in Chapter 5.

2.5 FPGA Benchmark Suites

Several benchmark suites have been curated and used by FPGA architecture and CAD researchers over the past three decades. Table 2.2 provides an overview of the features of the different suites. The classic MCNC20 (the twenty largest MCNC) benchmarks [135] are extremely small (less than 10K LUTs) and simple designs that do not use any FPGA hard blocks. While these designs were used in many early CAD and architecture studies such as [17], they are no longer very representative of modern FPGA use-cases. The UMass RCG HDL Benchmark Collection [5] has somewhat larger designs of up to $14,000$ look-up tables (LUTs) mostly representing digital signal processing (DSP) applications. However, this suite does not target an open-source FPGA framework, which limits its use in architecture and CAD studies as they generally need modifiable and retargetable CAD tools. The Groundhog benchmarks [62] are intended to be architecture independent; they work with academic tool flows and are targeted towards evaluation of power consumption of FPGAs for mobile computing applications. However, only two of the benchmarks have HDL realizations (and hence can be run through an

Table 2.2: Comparing FPGA benchmark suites

Benchmark Suite	Max. primitives per design	Use of Hard Blocks	Open Source CAD	Captures DL Domain
MCNC20 [135]	10K	×	✓	×
UMass RCG [5]	14K	✓	×	×
Groundhog [62]	1K	✓	✓	×
ERCBench [25]	65K	✓	×	×
VTR [85]	165K	✓	✓	×
Titan [86]	1.8M	✓	×	×

implementation CAD flow) and both are very small (under 1,000 primitives). ERCBench [25] consists of hybrid hardware/software applications. The designs in this suite are from the multimedia, wireless communications and cryptography domains, and it contains some medium size designs (up to 65,000 LUTs). They do not contain DL benchmarks, and do not readily work with academic (open source) FPGA tools.

VTR [85] has a suite of Verilog benchmarks as well. These VTR benchmarks vary from small (321 netlist primitives) to medium-sized designs (165,809 primitives) and they include applications from several domains including image processing, soft processors and arithmetic. The Titan benchmark suite [86] contains modern heterogeneous large designs (90K to 1.8M netlist primitives); these are HDL benchmarks (some of which were generated from high-level synthesis) that are provided as both as the source HDL and BLIF [93] format netlists that can be input to VPR [17, 85]. However, they target a hybrid CAD flow that is architecture-specific, as logic synthesis is performed using the Intel Quartus tool only for the Stratix-IV architecture.

In this dissertation, the Koios benchmark suite is presented, which is the only one that provides large, heterogeneous, architecture-agnostic benchmarks that work with a completely open-source flow, and focuses on the increasingly important DL domain. No other existing FPGA benchmark suite mentioned above satisfies these requirements. They either have small non-heterogenous designs that do not represent today's workloads, or do not work with an open-source CAD flow. None of them have DL-specific circuits. Hence, existing FPGA benchmark suites are not suitable for DL-specific FPGA architecture and CAD research. Quantitative comparison of Koios benchmarks with VTR and Titan benchmark suites is presented later in this dissertation in Chapter 6.

Chapter 3

Methodology

3.1 FPGA Architecture Exploration

Figure 3.1 shows the FPGA architecture exploration flow. There are 3 main components of this flow: an FPGA architecture model, a suite of benchmarks and a CAD tool. The FPGA architecture model captures the information about organization and types of different blocks on the FPGA and its routing architecture. For the various blocks, there is information such as the area and delay obtained from circuit-level implementation, and also how they are arranged on the chip. Additionally, there is information about the type of modes each block supports (e.g. how many heights and widths are supported by a Block RAM, what precisions are supported by the DSP Slice, etc.). For the routing architecture, there is information such as the length of wire segments, routing channel width, architecture of switch blocks, the type of switches, etc.

The benchmarks are designs written in an HDL (Hardware Description Language) like Verilog, or synthesized from an HLL (High Level Language) like C, using HLS (High-Level Synthesis). These benchmarks should be representative of the key markets and application domains that the FPGA is targeted

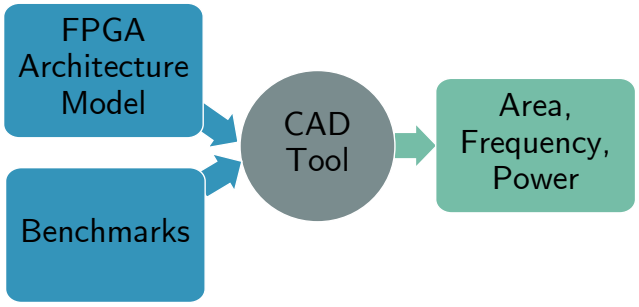


Figure 3.1: FPGA architecture exploration

towards. Typically, FPGA vendors have proprietary benchmarks obtained from their customers. But FPGA researchers in the academia use open-source benchmarks or handcrafted benchmarks for each project.

The CAD tool maps the given benchmarks to the give FPGA architecture model. It performs 4 main steps: synthesizing the benchmark into a netlist, mapping and packing the netlist onto the different blocks specified in the FPGA architecture, placing the mapped blocks at specific locations on the FPGA grid, and routing the connections between them using the routing architecture specified in the FPGA architecture model. The tool generates area, timing and power reports.

To explore FPGA architecture, the process or flow shown in Figure 3.1 is repeated with multiple FPGA architecture candidates for the same set of benchmarks, and the reports from each candidate architecture are compared to find the best architecture.

3.2 Tools

Experiments employed for this dissertation use the following tools:

- VTR (Verilog-To-Routing) [85] for FPGA architecture exploration
- COFFE [136] for FPGA architecture modeling
- Xilinx Vivado for implementation for Xilinx FPGAs (synthesis, placement and routing)
- Intel Quartus for implementation for Intel FPGAs
- Xilinx Vivado HLS for high level synthesis (from C to RTL (Register Transfer Level))
- Synopsys VCS for RTL simulations
- Synopsys Design Compiler for ASIC RTL synthesis
- Cadence Encounter for ASIC placement and routing
- Synopsys Primetime for ASIC static timing analysis
- Synopsys HSpice for SPICE simulations
- Python for scripting
- Mako template engine for Verilog code generation

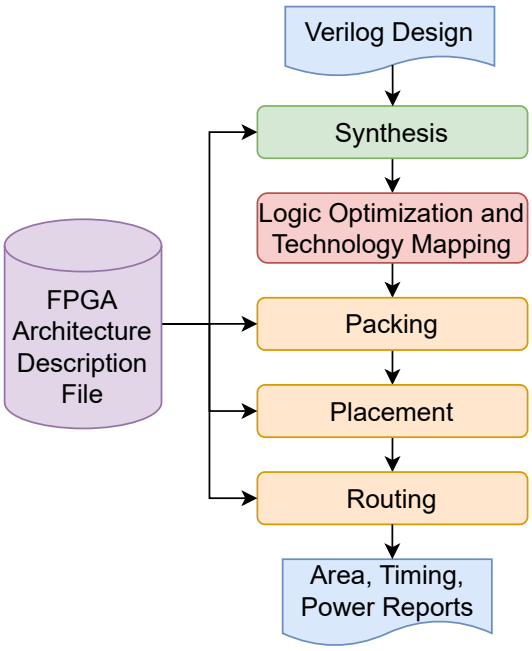


Figure 3.2: VTR Flow

In this section, these tools are briefly explained. Specific versions are mentioned when methodology for each dissertation contribution is explained in the subsequent chapters.

VTR (Verilog To Routing) [85] is the most popular CAD tool for performing FPGA architecture exploration. VTR takes two inputs. The first input is an architecture description file containing information about an FPGA's building blocks and interconnect resources. VTR provides a very expressive XML-based language to describe an FPGA architecture model. The second input is a benchmark, usually in the form of a Verilog design (or circuit). VTR synthesizes and implements the design on the given FPGA architecture and generates resource usage, area and timing reports. The VTR flow is shown in

Figure 3.2. VTR uses ODIN [63] as its synthesis tool (with Yosys [129] being recently integrated). ABC [23] is used for logic optimization and technology mapping. VPR [17] is used for packing, placement and routing. VTR provides many knobs that can be used to control the behavior of each tool - ODIN, ABC and VPR. It also provides scripts to run the flow for many benchmarks and many architecture files easily, and to parse results from generated reports.

To obtain the area and delay values for the various components of an FPGA (to enter them in the FPGA architecture description file for VTR), another tool called COFFE [136] is used. The COFFE flow is shown in Fiure 3.3. COFFE is a transistor sizing and FPGA modeling tool. The inputs to COFFE are the properties of the FPGA architecture (e.g. Number of BLEs in a Logic Block (N), Number of inputs on each LUT (K), Routing Channel Width (W), etc.), and the SPICE technology/process model. COFFE performs SPICE simulations to iteratively optimize the transistor sizing and generates areas and delays for the various subcircuits in the FPGA tile. This is shown in the left part of the figure. These subcircuits include routing subcircuits (switch block multiplexers, connection block multiplexers, local interconnect blocks, etc.), logic block subcircuits (LUT, LUT drivers, etc.), and RAM block subcircuits (row decoder, sense amplifier, configurable width decoder, etc.). The tool Synopsys HSPICE is used for SPICE simulations.

COFFE also supports a flow in which the core of hard blocks like DSP Slices or Tensor Slices is implemented using a standard cell flow and the interface to the routing/interconnect (local crossbar, switch block, connection

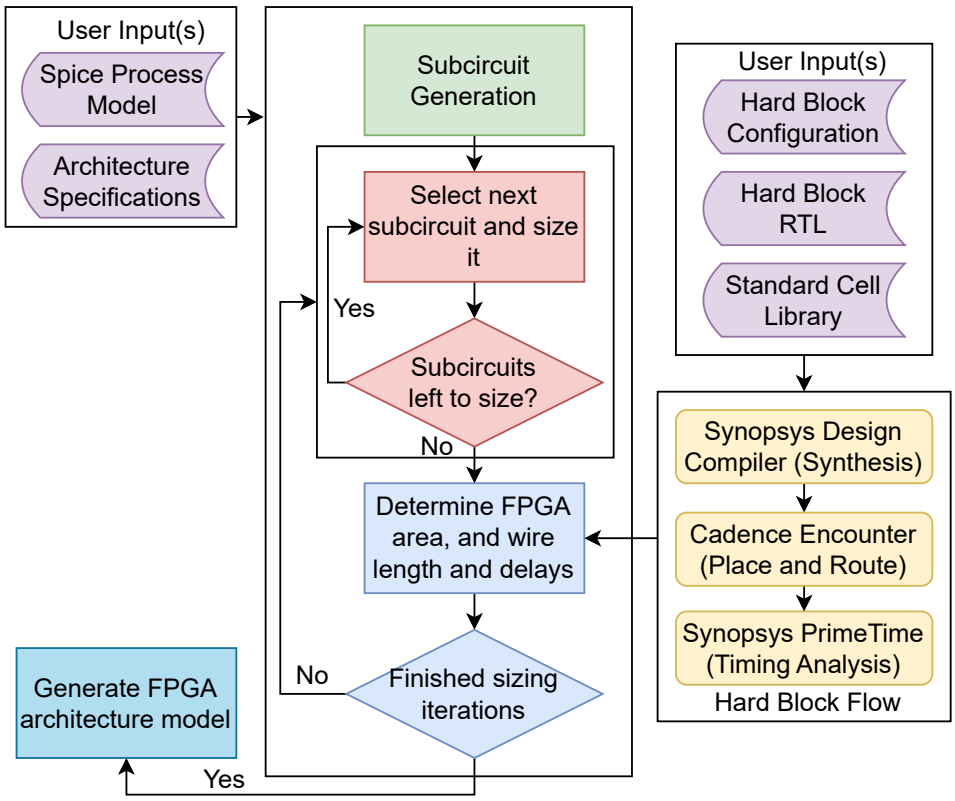


Figure 3.3: COFFE Flow

block, etc) is implemented in full custom using SPICE. This flow is shown in the right part of the figure. This hard block flow takes in the configuration of the hard block (e.g. number of inputs and outputs, height, target frequency, etc.), the hard block's RTL and a standard cell library as inputs. It uses Synopsys Design Compiler for synthesis, Cadence Encounter for placement & routing, and Synopsys Primetime for timing analysis. Results from these tools are fed into the transistor sizing iterations to appropriately size the routing interface of the hard block. When running COFFE, a cost factor of $area * delay^2$ is used as it reflects the greater emphasis on delay compared to area, which is typical of high-performance FPGAs.

For Verilog simulations, to verify designs generated during the research described in this dissertation, commercial tools such as Synopsys VCS and DVE, and Vivado's integrated simulator are used. Python scripts and Mako templating library are used for auto generation of Verilog, when needed. Python scripts are also used for automation of flows, such as running multiple architecture explorations in parallel and parsing results. For performance and energy modelling, analytical models are used, which are coded using Python. For implementing designs for commercial FPGAs, Xilinx Vivado and Intel Quartus are used. Xilinx Vivado HLS is used for high-level synthesis of designs, which involves generating Verilog RTL from high-level languages like C.

3.3 Libraries and PDKs

SPICE simulations performed for experiments described in this dissertation use the 22 nm transistor level libraries from Predictive Technology Model [117]. The 45 nm GPDK library from Cadence and the FreePDK45 [88] library are used when a standard cell library (also referred to as a PDK (Process Design Kit)) is needed, for example, while synthesizing the Tensor Slice. Scaling factors from Stillmaker et al. [108] are used to scale down from 45 nm to 22 nm.

Chapter 4

Adding DL-Specialized Compute Blocks in FPGAs

In this chapter, the first contribution of this dissertation is described: adding new blocks called Tensor Slices to the FPGA. These blocks are specialized for performing tensor/matrix operations like matrix-matrix and matrix-vector multiplication, which are common in DL workloads. Adding such blocks to an FPGA helps pack far more compute in the same area footprint and improves the performance of FPGAs for DL applications.

A Tensor Slice is architected and implemented, and the performance of an FPGA architecture with Tensor Slices (in addition to Logic Blocks, DSP Slices and RAM Blocks) is compared with an FPGA architecture similar to state-of-the-art Intel's Agilex FPGAs. A portion of the FPGA's area is converted into Tensor Slices and a significant performance boost is observed for DL benchmarks. Different percentages of the FPGA area spent on Tensor Slices are explored.

FPGAs provide flexibility to meet the requirements of a broad range of applications. Adding Tensor Slices to an FPGA is focused on accelerating DL applications. It can be a concern that adding such slices may impact the

generality of an FPGA, and hence may degrade the performance of non-DL applications by causing a higher routing wire length and longer critical paths. To that end, the impact of this contribution on non-DL applications is also evaluated. The Tensor Slice is enhanced so that it can be fractured into smaller math units like multipliers and MACs (multiply-and-accumulate) and used for non-DL workloads.

This part of the dissertation resulted in a paper publication in the ACM International Symposium on Field Programmable Gate Arrays (ISFPGA) [7] and an article in the ACM Transactions on Reconfigurable Technology and Systems (TRETS) [13]. The following contributions from the co-authors of these papers/articles are acknowledged:

- Samidh Mehta: Verilog implementation of the DSP Slice, and drawing several diagrams
- Moinak Ghosh: Adding int16 and bf16 support to Tensor Slice, creating separate implementations for each mode from the full feature Tensor Slice, and drawing several diagrams

4.1 Tensor Slices

4.1.1 Overview

A Tensor Slice is to Deep Learning, just like a DSP slice is to Digital Signal Processing. DSP Slices support the most common DSP operations like the MAC operation, along with additions and multiplications. Similarly, Tensor Slices support the most common machine learning operations like matrix-matrix multiplication and matrix-vector multiplication, along with element-wise matrix addition, subtraction and multiplication. The matrix-matrix and matrix-vector multiplication operations are pervasive in DL layers like fully-connected, convolution and recurrent layers. Element-wise (also referred to as Eltwise) matrix addition and subtraction is commonly found in layers like normalization, residual add and weight update. Eltwise matrix multiplication is used in layers like dropout. The Tensor slice also has support for bias-preloading and tiling.

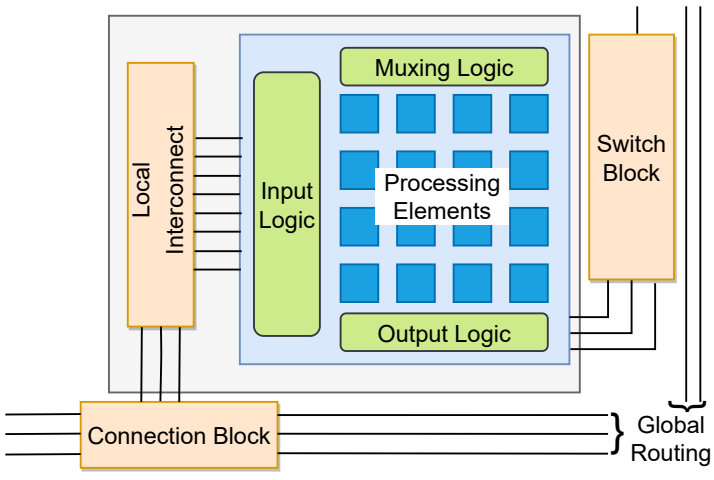


Figure 4.1: High-level block diagram of the Tensor Slice

Figure 4.1 shows a logical block diagram of Tensor Slice. The total number of inputs pins (including clock and reset) and output pins on the Tensor Slice are 310 and 298 respectively. The slice interfaces with the FPGA interconnect through connection block (for inputs) and switch block (for outputs), similar to other blocks on modern FPGAs. Outputs being directly connected to a switch box instead of using a separate output connection box is referred to as direct drive [78]. The Fc_{in} and Fc_{out} values used for the Tensor Slice are 0.15 and 0.10 respectively.

The slice has a 50% sparsely populated local input crossbar, that makes the input pins of the slice swappable and, hence, increases the routability of the FPGA. There are no feedback paths from the outputs of the slice feeding into the inputs via the crossbar. As seen later in Section 4.2.3, because of the large number of inputs on the Tensor Slice, this crossbar is large in area and adds a significant amount of delay, reducing the achieved frequency. There are multiple ways to reduce this area and delay. One is to reduce the population of the crossbar to below 50%. This is similar to experiments done in prior work [84] [77]. Another method is to have multiple smaller crossbars instead of one large crossbar as explored by Yazdanshenas et al. [136]. As shown later, the Tensor Slice spans 8 rows on the FPGA grid. So, the crossbar could be split into 8 smaller crossbars. Both these methods reduce the area and delay of the crossbar, but also reduce the routability of the Tensor Slice. This tradeoff analysis between crossbar area (and delay) and routability is not done in this dissertation and is left as future work.

The core of the Tensor Slice is a 2D array of 16 processing elements (PEs) along with control logic. Each PE consists of a multiplier and an adder which can act as an accumulator when a MAC operation is desired. The control logic consists of input logic, output logic and muxing logic. The input logic sets the input data correctly (e.g. appropriately delay it for systolic computation) to be processed by the PEs. The output logic selects the output data appropriately from the PEs and shifts it out. The muxing logic selects between various modes of operation of the slice.

The PEs are arranged as a 2D systolic array. Systolic arrays allow reusing a piece of data multiple times and never having to read it again, making them very efficient for compute-intensive tasks like matrix multiplication. Only the PEs along the left column and the top row of the 2D PE array have to interface with the programmable routing. Other PEs receive data from neighboring PEs. This allows for a reduced I/O footprint, which is very important to connect a block in an FPGA. Low pin density avoids increasing channel width and reduces routing congestion.

Tensor Slices are laid out along columns in the FPGA, similar to how other blocks (Logic Blocks, DSP Slices and Block RAMs) are laid out in a modern island-style FPGA. Some other layouts of large blocks like the Tensor Slice are explored in prior work [137] [8], but it is decided to keep the column layout. This allows for easy adoption of the proposal, because this makes FPGA design flow similar to current existing flow and also leads to fewer updates to FPGA CAD tools to support a new block.

The Tensor Slice supports four precisions natively: 8-bit fixed-point (int8), 16-bit fixed-point (int16), IEEE half-precision (fp16) and Brain Floating Point (bf16) [52]. These are the most commonly used precisions in DL inference and training. In int8 mode, all multiplications happen in int8, but accumulations are done in 32-bit fixed-point (int32). In int16 mode, all multiplications happen in int16, but accumulations are done in 48-bit fixed-point (int48). In the fp16 and bf16 modes, all multiplications happen in fp16 and bf16 respectively, but accumulations are done in IEEE single precision (fp32).

There are two primary modes of operation of the Tensor Slice: Tensor mode and Individual PE mode. In the Tensor mode, the slice operates on matrix inputs, whereas in Individual PE mode, it operates on scalar inputs. There are five sub-modes of the Tensor mode: Matrix-Matrix Multiplication, Matrix-Vector Multiplication, Eltwise Addition, Eltwise Subtraction and Eltwise Multiplication. There are two sub-modes of the Individual PE mode: Multiplier and MAC. All the modes and sub-modes supported by the Tensor Slice are shown in Figure 4.2. The mode of operation of the slice is dynamically selectable. That is, the mode bits can be changed during run-time without requiring reconfiguration of the FPGA.

As discussed later in this section, the Tensor Slice supports 4x4 matrices in the Tensor mode for the 16-bit precisions, and 8x8 matrices in the Tensor mode for the 8-bit precision. An exploration of the Tensor Slice size to use is performed. Sizes from 2x2, 4x4, 8x8, 16x16 are explored.

Designing large matrix multipliers using smaller Tensor Slices means

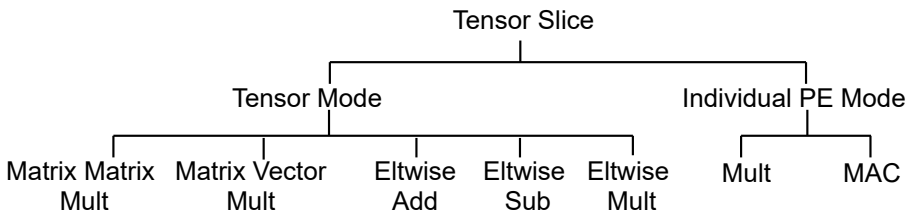


Figure 4.2: Modes supported by the Tensor Slice

using the FPGA programmable routing for any communication between the slices. Larger slices (e.g. 16x16x16) lead to less area, reduced routing wire-length and reduced power consumption for a given design, but they also lead to more routing area per block, increased channel width and increased routing wire segments per net.

The problem of under-utilization or fragmentation happens when a big Tensor Slice block (e.g. 16x16x16) is available, but a smaller matrix multiplication problem (e.g. 12x12x12) is to be calculated. This also happens when a larger problem size (e.g. 14x14x14) is to be calculated, but smaller Tensor Slices are available and do not evenly divide the edges of the problem size (e.g. 8x8x8). Providing smaller sized Tensor Slices on an FPGA means having less under-utilization and fragmentation problems, compared to providing larger sized Tensor Slices.

Results of this exploration are shown in Section 4.3.1. Sizes 4x4 and 8x8 have the best tradeoffs, and having one slice with 4x4 for 16-bit precisions and 8x8 for int8 precision allows the most reuse of the I/Os as well.

4.1.2 Processing Element

For this section, each processing element (PE) in the Tensor Slice is referred to as a physical PE, and the logic/circuitry required to process 1 matrix element is referred to as a logical or functional PE. There are 16 physical PEs in the slice. In 16-bit precision modes, the slice needs to process 16 matrix elements. So, there is a one-to-one correspondence between a physical PE in the slice and a logical PE required in 16-bit precision modes. For example, logical PE00 is physical PE00 of the slice, logical PE01 is physical PE01 of the slice, and so on up to PE33. However, in 8-bit precision mode, the slice processes 64 matrix elements, so it needs 64 logical PEs. Because of hardware sharing, each physical PE in the slice acts as 4 logical 8-bit PEs. So, physical PE00 in the slice maps to logical 8-bit PE00, PE01, PE02, PE03. Physical PE01 in the slice maps to logical 8-bit PE04, PE05, PE06, PE07. And so on. Figure 4.3 shows the diagram of one physical processing element (PE) configured for 8-bit precision operation (int8) as 4 logical PEs and for 16-bit precision operation (int16/fp16/bf16) as 1 logical PE.

Each PE consists of registers for shifting input data and a MAC. The MAC is shown in Figure 4.4. The figure also shows the multiplexing in the MAC required for the individual PE mode. Logically, the MAC consists of a multiplier and an adder. But to enable hardware sharing between the integer and floating point modes, the MAC contains multiple small-sized adders and multipliers which are combined to form larger adders and multipliers, along with multiplexing logic, floating-point logic (aligning, normalizing, executing,

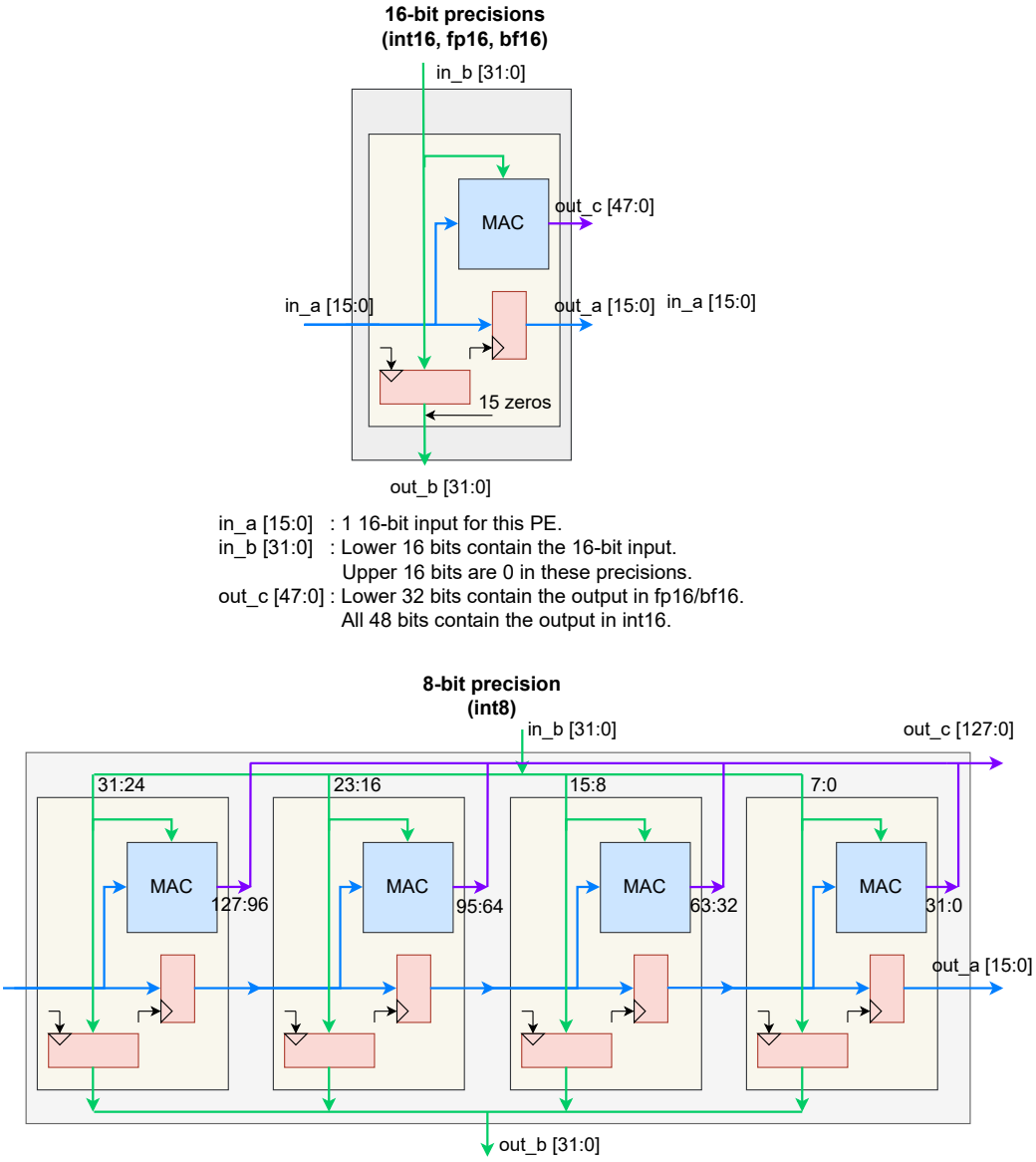


Figure 4.3: A processing element (PE) in 16-bit and 8-bit precisions. There are 16 such PEs in the Tensor Slice.

rounding, etc.) and pipelining registers. There are 4 8-bit multipliers and 16 8-bit adders in the MAC block.

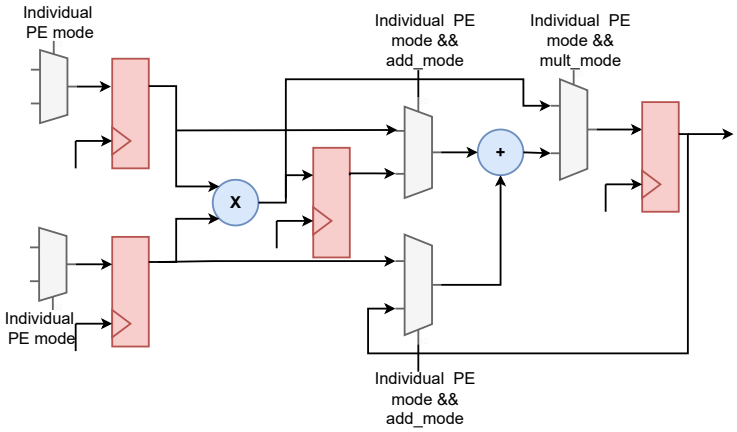


Figure 4.4: Functional diagram of the MAC block which forms the core of a PE

When operating in the int8 mode (Figure 4.5a), 4 int8 multiplications and 4 int32 additions are required. The 4 8-bit multipliers are directly used, and 4 int32 additions are performed by combining the 8-bit adders. When operating in the int16 mode (Figure 4.5b), 1 int16 multiplication and 1 int48 addition is required. The multiplication uses 4 8-bit multipliers along with 10 8-bit adders to add the partial sums. The int48 addition is performed by combining 6 8-bit adders.

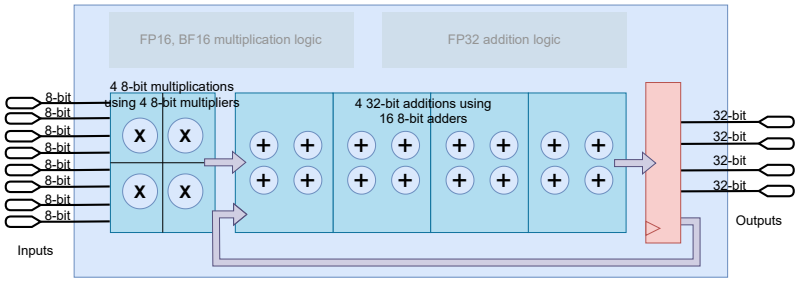
In floating point modes, the floating point logic reuses the 8-bit multipliers and 8-bit adders as required. In fp16 mode (Figure 4.5c), 1 fp16 multiplication and 1 fp32 addition are required. The fp16 multiplication logic needs to do an 11-bit multiplication (for mantissas), for which it uses the 4 8-bit multipliers and 8 8-bit adders (to add partial sums). It also needs a 5-bit

addition (for exponents), for which it uses 1 8-bit adder. In bf16 mode (Figure 4.5d), 1 bf16 multiplication and 1 fp32 addition are required. The bf16 multiplication logic needs to do an 8-bit multiplication (for mantissas), for which it uses 1 8-bit multiplier. It also needs an 8-bit addition (for exponents), for which it uses 1 8-bit adder. For the fp32 addition (required by both fp16 and bf16 modes) uses the same hardware. In the implementation of the fp32 adder, it needed 1 24-bit addition and 3 8-bit additions during its various stages. For this, it uses 6 8-bit adders. Some 8-bit adders stay unused in floating point modes.

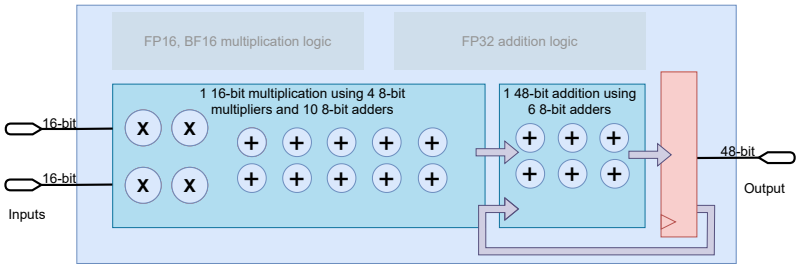
4.1.3 Tensor Mode

The I/O (Input/Output) pins on the Tensor Slice in Tensor mode are shown in Table 4.1. The Tensor Mode is configured by setting the `mode` input to 0. When configured to use int8 precision (`dtype` = 00), the Tensor Slice acts on 8x8 matrix operands and generates a 8x8 matrix result. In int16 (`dtype` = 01), fp16 (`dtype` = 10) and bf16 (`dtype` = 11) precisions, the Tensor Slice acts on 4x4 matrix operands and generates a 4x4 matrix result. By doing this, the I/O pins of the Tensor Slice can be fully utilized in each mode. Also, there are ample opportunities to share hardware between 4x4 fp16/bf16/int16 and 8x8 int8 array of processing elements.

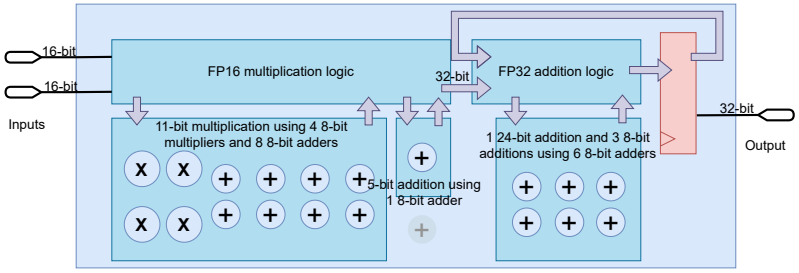
The Tensor Slice performs a tensor operation over multiple clock cycles. The input `start` is asserted to start the operation. The input matrices/vector would typically be stored in RAM blocks and some control logic implemented



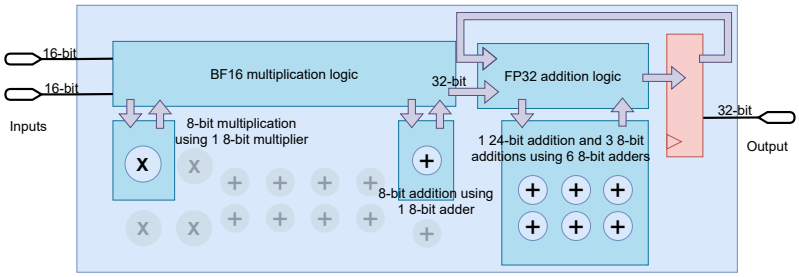
(a) Precision = int8



(b) Precision = int16



(c) Precision = fp16



(d) Precision = bf16

Figure 4.5: Sharing of arithmetic units (adders and multipliers) between different precisions of the MAC block. Multiplexers and pipeline registers not shown for clarity. Floating point logic refers to circuitry for alignment, normalization, rounding, etc.

Table 4.1: Inputs (I) and outputs (O) of the Tensor Slice in Tensor mode

I/O	Signal	Bits	I/O	Signal	Bits
I	clk	1	I	a_data_in	64
I	reset	1	I	b_data_in	64
I	mode	1	I	valid_mask_a_rows	8
I	accumulate	1	I	valid_mask_b_cols	8
I	preload	1	I	valid_mask_a_cols_b_rows	8
I	dtype	2	I	final_op_size	8
I	op	3	I	out_ctrl	1
I	start	1	O	b_data_out	64
I	x_loc	5	O	a_data_out	64
I	y_loc	5	O	c_data	160
I	a_data	64	O	c_data_available	1
I	b_data	64	O	flags	8
I	no_rounding	1	O	done	1

in soft logic would read the RAM blocks to feed the inputs to the slice. Alternatively, inputs may be generated from some upstream logic (e.g. hardware for the previous layer of a neural network) and fed directly into the slice without being stored in a RAM block. As the input matrices/vector are fed into the slice, control logic inside the slice orchestrates the data and applies the right data elements at the right time to specific PEs. When the output data is available in the PEs, it is sent out on `c_data` and `flags`. If out_ctrl is 0, the output data is automatically shifted out cycle-by-cycle when it is ready, but the user can control when to shift it out by setting `out_ctrl` to 1. The output `c_data_available` is asserted when output data is valid on `c_data` and `flags`. `flags` contain the logical OR of the exception flags from the PEs in a column and are only valid for floating-point precisions. The output data can be stored in a RAM block, or directly fed to downstream logic (e.g. hardware

for the next layer of a neural network) as it is generated by the slice. When the entire operation is done, the slice asserts the `done` signal.

Although the size of the matrix operations performed by the Tensor Slice are 4x4 and 8x8, the Tensor Slices can be chained to perform larger matrix operations. Section 4.1.3.4 provides details about this. Similarly, the Tensor Slice can support non-square inputs as well. For this purpose, there are validity masks for the inputs. This is done using `valid_mask_a_rows`, `valid_mask_a_cols_b_rows` and `valid_mask_b_cols` pins on the slice. For example, when multiplying a 6x4 matrix with a 4x7 matrix in int8 mode, the values of these inputs can be 8'b0011_1111, 8'b0000_1111 and 8'b0111_1111 respectively. Note that the number of columns in input matrix A is required to be the same as the number of rows in input matrix B.

In Tensor mode, bias and tiling support can be enabled. For bias (controlled using `preload`), the Tensor Slice supports *pre-loading* the PEs with an input matrix, which is effectively added to the result of the subsequent matrix operation. For tiling (controlled using `accumulate`), the Tensor Slice supports the choice of *not-resetting* the results in the PEs before starting another operation. This can be used in performing tiled or blocked matrix multiplications, where the partial sums need to be accumulated across tiles or blocks.

4.1.3.1 Matrix-Matrix Multiplication (matmul) Mode

The matrix-matrix multiplication mode is enabled when `op` = 000. The matrix-matrix multiplication operation in the Tensor Slice is done systolically. Only the PEs along the left column and the top row of the 2D PE array receive external data. Other PEs receive data from neighboring PEs. The elements of the first input matrix (matrix A) move from left to right, and the elements of the second input matrix (matrix B) move from top to bottom. The result is calculated *during* the shifting process, and it *stays* in the respective PE until its computation is done. After that, the resulting matrix (matrix C) is shifted out left to right column-wise in a pipelined fashion. When the results are being shifted out, another tensor operation can be started on the Tensor Slice.

Elements of one operand matrix are applied column-wise to the input `a_data`. Elements of the second operand matrix are applied row-wise to the input `b_data`. In one cycle, 8 int8 elements or 4 int16/fp16/bf16 elements are applied to `a_data` and the same number of elements are applied to `b_data`. The output data is available on `c_data` and `flags`. In one cycle, results from one column of PEs are shifted out (See Section 4.1.3.5 for more details). Only 128 bits of c_data and 4 bits of flags are used in this mode.

Figure 4.6 (a) shows the systolic setup of data from matrix A (left-to-right). See the path from **a_data** to the PEs through the flip-flops and A-mux. The muxing required for chaining (A-mux) that selects between **a_data** and **a_data_in** is discussed later in Section 4.1.3.4. Figure 4.6 (b) shows the

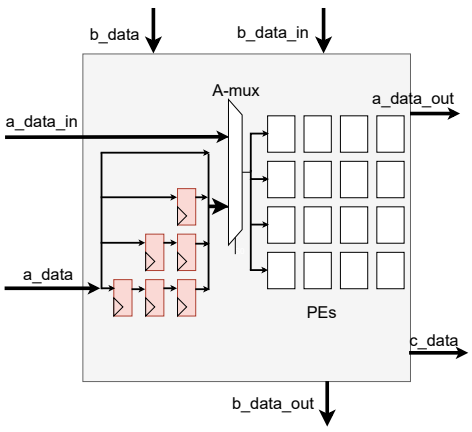
same logic, but for data from matrix B (top-to-bottom). Figure 4.6 (c) shows the movement of matrix A elements (in red) and matrix B elements (in yellow) through the PEs. Figure 4.6 (d) shows the shift out of the results (i.e. data for matrix C).

Matrix-matrix multiplication operation in the tensor mode is the most compute intensive operation done by the Tensor Slice. When using 16-bit precisions (int16, fp16, bf16), the slice performs 16 MAC operations in 1 cycle. So the math throughput of the slice is 16 MACs/clock. When using 8-bit precision (int8), the slice's math throughput is 64 MACs/clock. To keep the slice fed with data, it reads 8 16-bit elements every clock cycle in 16-bit precision modes and 16 int8 elements every clock cycle in int8 precision mode. So, the on-chip memory bandwidth requirement of the Tensor Slice is 16 bytes/clock.

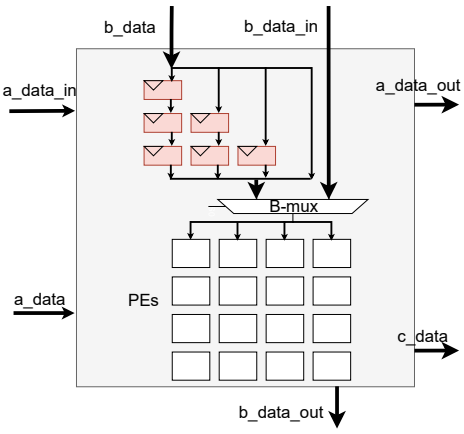
4.1.3.2 Matrix-Vector Multiplication (matvec) Mode

The matrix-vector multiplication mode is enabled when `op` = 100. The matrix-vector multiplication operation in the Tensor Slice is also done systolically. The elements of the matrix move from left to right and the elements of the vector move from top to bottom. The result is calculated *during* the shifting process, and it *stays* in the respective PE until its computation is done. After that, the resulting vector is shifted out column-wise in 1 cycle.

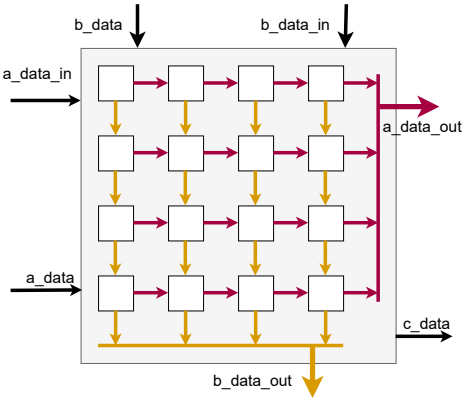
Elements of the input matrix are applied column-wise on the input `a_data`. Elements of the input vector are applied to `b_data`. Note that only 8 bits of `b_data` are used for int8 precision, and 16 bits of `b_data` are used



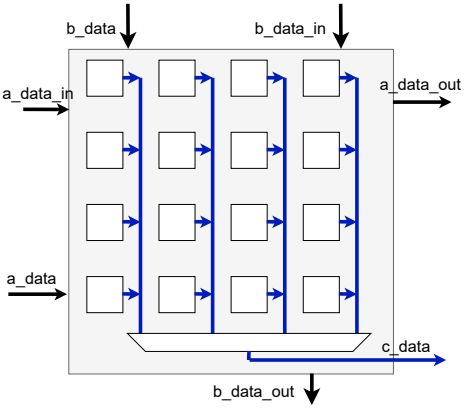
(a) Systolic data setup for Matrix A



(b) Systolic data setup for Matrix B



(c) Flow of input matrices within the Tensor Slice



(d) Flow of output matrix within the Tensor Slice

Figure 4.6: Various aspects of operation of the Tensor Slice

for int16/fp16/bf16 precisions. Since there is only 1 column in a vector, this implies only the PEs in one column of the 2D PE array are utilized. An opportunity to improve the utilization of PEs is identified by observing that there are many I/O pins on the Tensor Slice that are required only for the matrix-matrix mode, but are not required in matrix-vector mode (and eltwise modes as well). Multiplexers are added in front of the PEs in the third column to expose them to already-existing unused I/O pins so that these PEs can also be loaded directly from the outside (instead of getting data from PEs to their left). Through this set of wires (called `second_a_data`), another matrix can now be fed in the matrix-vector mode. This is shown in Figure 4.7. This is a slight deviation from a pure systolic design, in which only the PEs on the periphery read/write data from outside. However, the overhead of adding this feature is low, and the utilization of the slice doubles in the matrix-vector mode. More multiplexers can be added to PEs in other columns and rows to further increase the utilization of the Tensor Slice in matrix-vector mode. However, this will require new I/O pins to be added to the Tensor Slice. I/O pins on a block in the FPGA fabric are costly in terms of area (larger local crossbar) and routing (more congestion). Adding multiplexers also increases the combinatorial delays of timing paths going through them. So, the cost-benefit tradeoff needs to be carefully studied before adding multiplexers to more columns and rows.

The second vector can be fed from the bits of b_data that are unused in this mode. With this enhancement, two independent matrix-vector products

can be calculated at the same time in the slice. Some other unused I/Os can be used for validity masks and for reading out the output results. Here is the mapping of I/O pins used for the reading a second matrix and a second vector, and for outputting a second result in the Matrix-Vector Multiplication mode:

- `second_a_data` is mapped to a_data_in
- `second_validity_mask_a_rows` is mapped to validity_mask_b_cols
- `second_validity_mask_a_cols_b_rows` is mapped to b_data[23:16]
- `num_rows_matrix` is mapped to final_op_size
- `num_cols_matrix` is mapped to b_data[31:24]
- `second_b_data` is mapped to b_data[47:32]
- `second_c_data` is mapped to {c_data[159:128], b_data_out[63:48], b_data_out[31:16], a_data_out[63:0]}
- `second_flags` is mapped to flags[7:4]

The `num_rows_matrix` is used to specify the number of rows of the matrix, whereas the number of columns in the matrix (and hence the number of elements in the vector) is specified using `num_cols_matrix`. These are used inside the Tensor Slice to calculate the number of cycles elapsed to start shifting out the results and to assert the `done` signal.

In matrix-vector multiplication mode, when using 16-bit precisions, the slice performs 8 MAC operations in 1 cycle (4 in column 1 and 4 in column 3). So the math throughput of the slice is 8 MACs/clock. When using 8-bit precision, the slice's math throughput is 16 MACs/clock. To keep the slice fed with data, it reads 10 16-bit elements every clock cycle in 16-bit precision modes, and hence the on-chip memory bandwidth requirement is 20 bytes/clock. Similarly, it reads 18 int8 elements every clock cycle in int8 precision mode, and hence, the on-chip memory bandwidth requirement is 18 bytes/clock.

4.1.3.3 Eltwise Modes

Element-wise matrix operations are supported by the slice as well, and can be performed by selecting the right settings of the `op` pins (`op` = 001 => eltwise multiplication; `op` = 010 => eltwise addition; `op` = 011 => eltwise subtraction). For the eltwise operations, the elements of the first matrix move left to right and the elements of the second matrix move from top to bottom. The result calculation happens *after* all inputs have reached their respective locations in the PE array. This method of moving data through the PEs in eltwise mode increases the number of cycles required for an eltwise operation.

The enhancement used for matrix-vector mode to increase the utilization of PEs can be extended to reduce the cycles required in eltwise mode by 2×. Instead of only feeding data into the 2D PE array from the left-column and top-row, additional PEs internal to the array can be fed, without adding

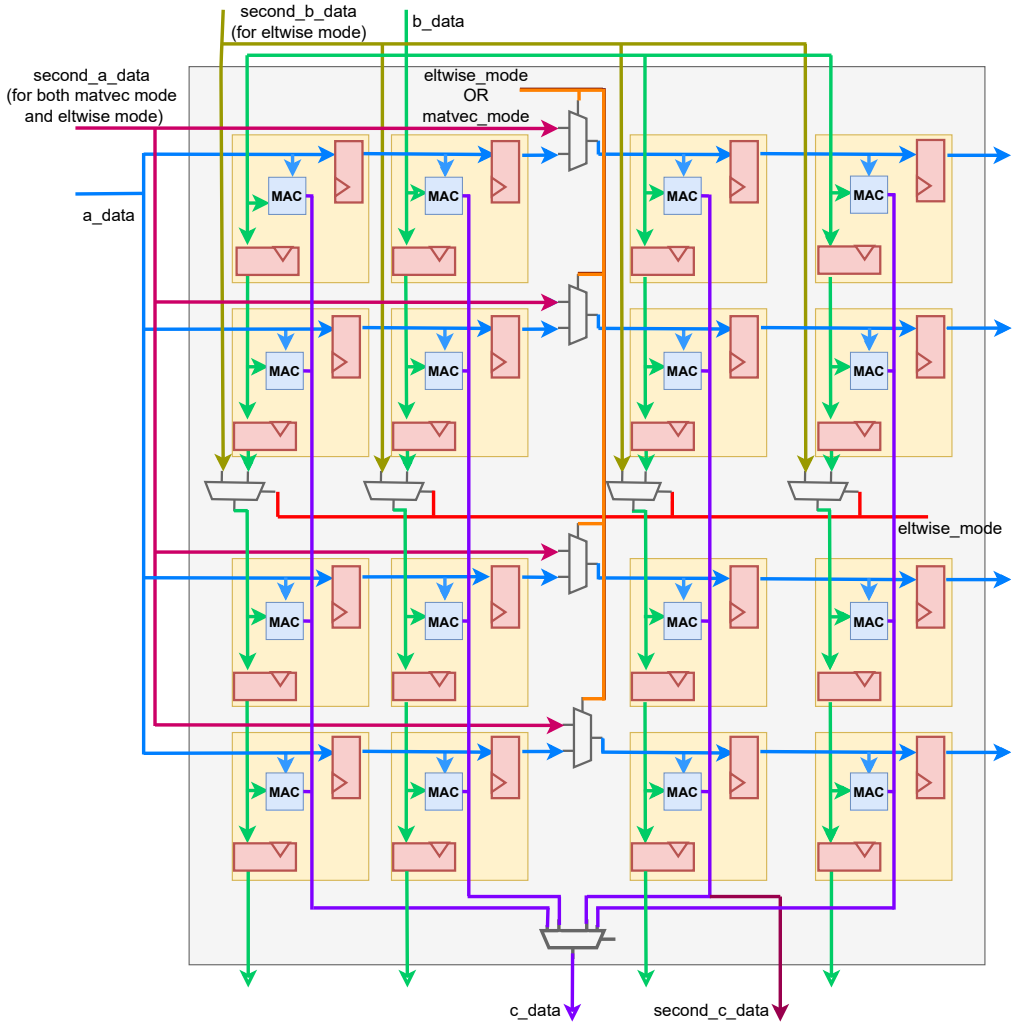


Figure 4.7: Exposing internal PEs to increase the utilization in matrix-vector multiplication mode and reduce the number of cycles in eltwise modes.

any extra I/O cost. For matrix-vector multiplication mode, the third column is exposed on existing I/Os. In addition to exposing the third column, the third row is also exposed in eltwise mode. This enables loading of two columns of matrix A and two rows of matrix B at the same time, doubling the loading

speed without adding any I/Os. The cost is a few multiplexers. I/Os unused in matrix-matrix multiplication mode are used for reading out the output results. This is also shown in Figure 4.7. The following list shows the mapping of I/O pins used for the reading a second matrix and a second vector, and for outputting a second result in Eltwise Modes:

- `second_a_data` is mapped to a_data_in
- `second_b_data` is mapped to b_data_in
- `second_c_data` is mapped to a_data_out[63:0]
- `second_flags` is mapped to flags[7:4]

4.1.3.4 Chaining

Multiple Tensor Slices can be chained to perform operations on larger matrices. This is useful in matrix-matrix and matrix-vector multiplication operations. Figure 4.8 shows a logical view of 4 Tensor Slices chained in x and y directions to perform a larger matrix-matrix multiplication operation (e.g. a 8x8 matrix multiplied with a 8x8 matrix using 4 slices in fp16 mode). Signals `a_data_in` and `a_data_out` are used to chain the inputs from matrix A along the x direction. The signals `b_data_in` and `b_data_out` are used to chain the inputs from matrix B along the y direction. Only the Tensor Slices at the periphery are fed inputs. Inputs flow through the slices through the chains. The `c_data` signal contains the output of the Tensor Slice. It can be chained

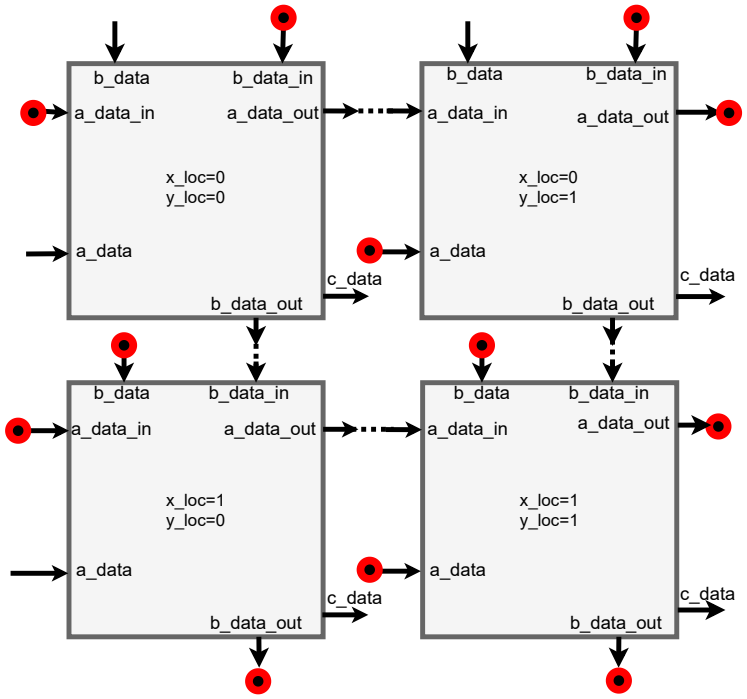


Figure 4.8: Multiple Tensor Slices can be chained together to perform larger matrix-matrix multiplications. Here, 4 slices are shown to be chained together in x & y direction (logically).

with the output from neighboring Tensor Slices using soft logic or directly consumed from each Tensor Slice block, depending on the requirements of the user's design.

Note that the figure shows a logical connectivity of the slices in the x (horizontal) and y (vertical) directions. Physically, these slices can be anywhere on the FPGA. For example, 4 Tensor Slices in one grid column of the FPGA could be connected to perform a larger matrix operation. The inputs `x_loc` and `y_loc` are used to specify the logical location to the slices. Note that `x_loc` and `y_loc` do not determine or are related to the physical location

of a slice in the FPGA grid. These signals are decoded internally to select the correct input port(s) whose data should feed the PEs. For example, the top-left slice in the logical grid of slices has `{x_loc,y_loc}=00`, implying that this slice should use the input data received at `a_data` and `b_data` ports. The bottom-right slice in the logical grid of slices has `{x_loc, y_loc}=11`, implying that this slice should use the inputs received at `a_data_in` and `b_data_in` from its logical neighbors to the left and top respectively. Not only do different slices in a logical grid receive data from different ports, they get data at different times as well. For example, the inputs going into the slice with `{x_loc, y_loc}=11` are delayed with respect to the inputs going into the slice with `{x_loc, y_loc}=00`. `x_loc` and `y_loc` are also used in the control logic in the slice to sample the incoming data at the appropriate time.

The input `final_op_size` is used to specify the overall size of the matrix operation being performed. In the case of the example shown in Figure 4.8, assuming int16 operation, the `final_op_size` will be set to 8, because 4 slices are connected together, and each slice performs a 4x4 matrix operation. This signal is used in the control logic in the slice to determine when the computation is finished and when to start shifting out the result.

Consider a matrix-matrix multiplication, where a MxK matrix is multiplied with a KxN matrix. For large values of M, the Tensor Slices are chained in the y (logically vertical) direction. For large values of N, the Tensor Slices are chained in the x (logically horizontal) direction. For large values of K, instead of chaining, typically, a longer number of cycles is used to accumulate

the results. In other words, M and N dimensions are handled by using more hardware (more "space"), whereas K dimension is handled by using more cycles (more "time"). An advantage of mapping the K dimension on "time" is that the extended precision intermediate results do not need to move. The same concept applies to a matrix-vector multiplication, except that there is no requirement of chaining in the x (logically horizontal) direction. It only makes sense to chain Tensor Slices in the y (logically vertical) direction.

4.1.3.5 Rounding

As mentioned above, accumulations in the Tensor Slice are done at a higher precision, compared to the multiplication. In other words, the results have higher precision compared to the operands. When `no_rounding` is set to 1, the outputs are shifted out in the higher precision without being rounded to the input precision. But when `no_rounding` is set to 0 by the user, the outputs are rounded to input precision before being shifted out. Convergent rounding or "round half to even" rounding [128] is used to round the results. When rounded results are shifted out, it can take less number of cycles depending on the precision. For example, in the matrix-matrix multiplication int8 mode, if rounding is disabled, the output from 8 PEs (8 PEs = 1 column of PEs for int8 precision) is 8*32=256 bits. The `c_data` signal is 128 bits. So, it takes 16 cycles to shift out the data of all 8 columns. However, if rounding is enabled, the output from 8 PEs is 8*8=64 bits. So, it takes 8 cycles to shift out data of all 8 columns. In matrix-matrix multiplication fp16 mode, if rounding is

disabled, the output from 4 PEs (4 PEs = 1 column of PEs for fp16 precision) is 4*32=128 bits. So, it takes 4 cycles to shift out the data of all 4 columns. If rounding is enabled, the output from 4 PEs is 4*16=64 bits. So, in this case also, it takes 4 cycles to shift out data of all 4 columns.

4.1.4 Individual PE Mode

When the `mode` input pin is set to 1, the Tensor Slice changes to Individual PE mode. The main goal of providing this mode is to reduce the impact of adding Tensor Slices to an FPGA on non-DL applications and to improve utilization. In this mode, the slice is fractured such that inputs and outputs of individual PEs are exposed to the pins of the slice, enabling the PEs to be used like mini-DSP slices. Each PE can be separately and dynamically configured in two sub-modes: Multiplier or MAC. Furthermore, all the 4 precisions (int8, int16, fp16 and bf16) are available and can be dynamically selected. In int8 mode, each PE can be configured to be used as 2 8-bit multipliers or 1 8-bit MAC with 32-bit accumulation. In int16 mode, each PE can be configured to be used as 1 16-bit multiplier. In fp16 and bf16 modes, each PE can be configured to be used as 1 16-bit multiplier or 1 16-bit adder or 1 16-bit MAC with fp32 accumulation. Note that because of the large delay to access the PEs in the Tensor Slice (because of the local input crossbar), using the Individual PE mode will not be performant compared to, for example, a DSP slice based multiplication or MAC.

There is a limitation of this mode. The number of inputs and outputs

on the slice (also called the I/O footprint of the slice) is governed by the Tensor mode (310 inputs, including clock and reset, and 298 outputs). Based on that, 8 PEs out of the 16 PEs can be exposed. Additional inputs and outputs could be added to the slice to accommodate for exposing all 16 PEs in individual PE mode, but that would mean worsening the I/O footprint of an already large slice. Increasing the number of I/Os can lead to more routing congestion and higher channel width requirement, and also a larger area of the Tensor Slice.

The inputs and outputs of an exposed PE are:

- `direct_in_a[15:0]`
- `direct_in_b[15:0]`
- `direct_mode` (Multiply or MAC)
- `direct_dtype[1:0]` (int8, int16, fp16 or bf16)
- `direct_out[31:0]`
- `direct_flags[3:0]` (exception flags for floating point mode)

Each exposed PE's inputs and outputs are mapped onto the top-level inputs and outputs of the slice (shown in Table 4.1). The mapping of all inputs and outputs to various PEs is not significant for the work in this dissertation, but here's an example of the pin mapping for exposed PE #1:

- `direct_in_a[15:0]` is mapped to {valid_mask_b_cols, final_op_size}

- `direct_in_b[15:0]` is mapped to a_data[31:16]
- `direct_mode[1:0]` is mapped to x_loc[3:2]
- `direct_dtype` is mapped to accumulate
- `direct_out[31:0]` is mapped to c_data[31:0]
- `direct_flags[3:0]` is mapped to b_data_out[7:4]

4.2 Evaluation Methodology

The goal of evaluating the Tensor Slice is to compare the performance of an FPGA with Tensor Slices, LBs, DSPs and RAMs on it, with an FPGA with only the traditional building blocks (LBs, RAMs and DSPs). Intel Agilex-like FPGA architecture is used as the baseline. There are differences between this baseline architecture and Intel Agilex (e.g. HyperFlex is not modeled), but for the purposes of this evaluation, as long as the baseline and the proposed FPGA architectures only differ in presence/absence of Tensor Slices, the results will hold. Also, 22 nm technology node is used in the evaluation here, but Intel Agilex devices are 10 nm.

4.2.1 Tools Used

To evaluate and compare FPGA architectures, the Verilog-to-Routing (VTR) tool flow is used [85]. COFFE [136] is used to obtain the area and delay values for the various components of an FPGA (to enter them in the FPGA architecture description file for VTR).

4.2.2 DSP Slice Implementation

A DSP slice that closely matches the DSP slice from Intel Agilex DSP user guide [54] is designed. It supports all major modes and all precisions - 9x9, 18x19, 27x27, fp16, bf16, fp32. The slice also supports input chaining (scanin-scanout) and output chaining (chainin-chainout). This Agilex-like DSP slice does not include some features that are present in the Agilex DSP slice. For

e.g. there is limited support for internal coefficients and constants, no double accumulation support, limited support for bypassing pipeline registers, and no enable signals for registers. To sanity check this implementation, a fixed-point-only slice (similar to the fixed-point part of Intel Arria 10) is first implemented and its area and delay numbers are compared to those from Boutros et al. [18]. Very similar numbers are obtained after scaling for technology nodes. For floating-point units, the architecture described in [24] (hard multiplier and adder based, not the soft logic based, not the iterative design) is used, with a slightly different pipelining scheme. The round to nearest tie-breaks-to-even (RNE) rounding scheme is used. Support for exceptions is present as well.

Table 4.2: Breakdown of the DSP Slice area (post P&R)

Component	**Area (um^2)**
Standard-cell core	7701
Local input crossbar	1480
Dedicated output routing	26
Switch box (4)	2736
Connection box	652
Total	**12597**

For this Agilex-like DSP slice, the critical path delay comes out to be 2.93ns (341 MHz) in floating point mode and 2.33ns (429 MHz) in fixed point mode. The delay of the 50% sparsely populated local crossbar is 0.33ns. This DSP slice has 130 non-dedicated inputs and 74 non-dedicated outputs. It spans 4 rows in the FPGA grid (1 LB spans 1 row) and the DSP slice column is 1.6× wide (compared to that of a LB). Table 4.2 shows the breakdown of the DSP slice area obtained from COFFE. About 40% of the area of the slice

Table 4.3: Overhead of supporting more precisions in the DSP Slice (post-synthesis area ratios)

DSP Slice variant	Ratio
Fixed point 18x19 and 27x27 (similar to Intel Arria)	1
Add fp32 support (similar to Stratix 10)	1.26
Add support for fp16, bf16 and fixed point 9 bit (similar to Agilex)	1.51

is routing, whereas 60% is the core. Table 4.3 shows the post-synthesis area of the DSP Slice as more precisions are added to it.

4.2.3 Tensor Slice Implementation

A Tensor Slice using the architecture described in Section 4.1 is designed. The Tensor Slice uses the same core fixed-point and floating-point adders and multipliers as the ones used in the DSP slice. The number of inputs in the Tensor Slice is much larger than that of the DSP Slice, so the local input crossbar delay is higher (0.765ns). The critical path delay of the Tensor Slice is 3.31ns (302 MHz) in floating point mode and 2.56ns (391 MHz) in fixed point mode. The Tensor Slice has 308 non-dedicated inputs and 298 non-dedicated outputs. It spans 8 rows in the FPGA grid, and the Tensor slice column is 3.5× wide (compared to that of a LB).

Tables 4.4 and 4.5 show the breakdown of the Tensor Slice area obtained from COFFE. About 18% of the area of the slice is routing, whereas 82% is the

core. Within the core, ∼90% of the area is consumed by the PE array. Inside each PE, the adder takes ∼44% area and the multiplier takes ∼30% area. The adder takes more area than the multiplier, primarily because of the presence of fp32 adder. Table 4.6 shows the post-synthesis area of the Tensor Slice as more modes are added to it. Adding the individual PE mode adds about 18.5% of the area on top of the Tensor Slice supporting matrix multiplication (with the fp16 and int8 precisions). This overhead is high but makes the Tensor Slice usable in non-DL applications, if required. Adding elementwise and matrix-vector modes adds 13% area in the end. The elementwise mode is found to be the least useful when mapping benchmarks to Tensor Slices (presented in Section 4.2.5.1), so it is a contender for removal to recover some area for future enhancements.

Table 4.4: Breakdown of the Tensor Slice area (post P&R)

Component	**Area (um^2)**
Standard-cell core	45404
Local input crossbar	2771
Dedicated output routing	0
Switch box (8)	5473
Connection box	1570
Total	**55219**

Table 4.5: Area distribution of the various components of the Tensor Slice core (left) and the Processing Element (right)

Component	**Area (%)**
Input logic	3.14
Output logic	6.25
2D PE array	90.61
Total	**100**

Component	**Area (%)**
Adder	44.1
Multiplier	29.7
Rest	26.1
Total	**100**

Table 4.6: Overhead of adding more functionality/modes/precisions in the Tensor Slice (post-synthesis area ratios)

Tensor Slice variant	Ratio
8x8 int8 matrix mult only	1
4x4 fp16 matrix mult only	0.89
4x4 fp16 and 8x8 int8 matrix mult	1.34
Add individual PE modes	1.59
Add bf16 and int16 modes	1.86
Add element-wise and matrix-vector multiplication modes	2.10

4.2.4 Baseline vs. Proposed FPGAs

The routing and tile parameters of the FPGA architecture used for the baseline and proposed FPGAs are shown in Table 4.7. These are based on modern Intel FPGAs. Blocks on the FPGAs (LBs, DSPs, etc) are arranged in columns. There are no sectors or super-logic-regions. I/O pads are arranged along the perimeter of the FPGA. Unidirectional routing with wire segments of length 4 (260 out of 300 wires) and length 16 (40 out of 300 wires) are used. The switch blocks use a custom switching pattern based on the Stratix-IV-like architecture used in the Titan flow [86]. Each logic block (LB) contains 10 basic logic elements (BLEs). Each BLE has a 6-input LUT which can be fractured into two 5-input LUTs. The BLE also has 2 flip-flops and 2 bits of arithmetic, with dedicated carry chains between LBs. RAM blocks have a capacity of 20 Kilobits and have registered inputs and outputs. True and simple dual port modes with varied heights and widths (512x40, 1024x20, 2048x10) are supported. DSP Slice supports multiple precisions -

9x9, 18x19, 27x27, fp16, bf16, fp32 and has almost all the modes and features present in Intel Agilex DSP Slice. The FPGA architecture file provided with the Koios benchmarks [12] is used because it matches these features. It is modified to update the properties of the DSP Slice from Section 4.2.2, to create the baseline FPGA architecture. Then, the Tensor Slice block from Section 4.2.3 is added, to create the proposed FPGA architecture. The switch setting for all hard blocks, include Tensor Slices, is left to the default value: $external\_full\_internal\_straight$, which means that there are *full* switchblocks outside a block (i.e. both straight-through connections and turns are allowed) and *straight* switchblocks inside (i.e. only straight-through connections are allowed). This implies that routing wires can cross the Tensor Slice and there are no switches/transistors on the global routing wires crossing the Tensor Slice. Having switches on global routing wires inside the Tensor Slice can increase routability, but increases the area of the Tensor Slice. Similarly, not allowing global routing wires to pass through the Tensor Slice can adversely impact routability of the FPGA. Exploring such different switchblock configurations for Tensor Slices is left for future work.

From Intel Agilex's product table, the product with the most compute intensive resource mix is identified: AGF 027 (91280 LBs, 8528 DSP slices and 13272 RAM blocks). Based on the areas of each block on the FPGA obtained from COFFE, the total area of the FPGA consumed by each type of block is calculated. For the baseline architecture, the exact same resource mix as Intel Agilex AGF 027 is used in terms of percentage of the area and percentage

Table 4.7: Routing and tile architecture parameters of the baseline and proposed FPGA architectures

Parameter	Value	Definition
N	10	Number of BLEs per cluster
W	300	Channel width
L	4,16	Wire segment lengths
I	60	Number of cluster inputs
O	40	Number of cluster outputs
K	6	LUT size
Fs	3	Switch block flexibility
Fc_{in}	0.15	Cluster input flexibility
Fc_{out}	0.1	Cluster output flexibility
Fclocal	0.5	Local input crossbar population

of the count of various blocks on the FPGA. The absolute size of the FPGA is smaller than Agilex to ensure speedy simulations and also to ensure high utilization of the FPGA for the benchmarks used for realistic results. Table 4.8 shows the total count of each block type, the percentage of total area spent on each block type and the percentage count of each block type for Intel Agilex AGF 027 and the baseline FPGA. Six variations of the proposed FPGA architecture are created by spending 5%, 10%, 15%, 20%, 25%, 30% area of the FPGA on Tensor Slices respectively. These architectures are referred to as "Prop_Xpct", with X taking a value from {5,10,15,20,25,30}. Table 4.9 shows the resource mix of these FPGA architectures. The main goal of creating multiple variations of the proposed FPGA is to study the sensitivity of various metrics for non-DL benchmarks to increasing the area consumed by Tensor Slices on an FPGA.

Architectures that spend more than 30% of the FPGA area on Tensor Slices are not considered for the evaluation in this dissertation. That is

because, as shown later in Section 4.3, the degradation in metrics like area, frequency and routing wirelength is monitored for non-DL benchmarks with increasing area spent on Tensor Slices. A compound metric for performance is obtained by multiplying the change in average area, frequency and routing wirelength across the non-DL benchmarks. This metric exceeds 10% when the area spent on Tensor Slices is 30%. In other words, it is observed that non-DL applications do not degrade significantly (less than 10% degradation) on FPGA architectures where up to 25% of the area is spent on Tensor Slices. Additionally, the largest DL microbenchmark used for evaluation needs 160 Tensor Slices and the Prop_25pct variation does not have enough Tensor Slices for this microbenchmark to be successfully mapped. So, an architecture with 30% area spent on Tensor Slices is required for evaluation[1]. Qualitatively, spending more than 25-30% area on Tensor Slices makes the FPGA too DL-specific. It reduces the number of other blocks - LBs, DSPs, BRAMs - on the FPGA. So, non-DL applications may not fit the FPGA, if they can not use Tensor Slices[2]. This dissertation does not the prescribe a specific value of FPGA area to be spent on Tensor Slices. FPGA vendors typically design multiple FPGA families. So, different FPGA chips with different area occupied by Tensor Slice can be designed.

The grid dimensions of all the architectures used in experiments are

[1] A case where a large DL benchmark requires more Tensor Slices than present on the FPGA is evaluated in Section 4.3.11

[2] A case where a large non-DL benchmark requires more DSP Slices than present on the FPGA is evaluated in Section 4.3.10

Table 4.8: Resource mix in Agilex AGF047 and the baseline FPGA architecture

		AG047			Baseline		
Block	Relative area	# blocks	% area	% count	# blocks	% area	% count
Logic Block	1	91280	48.52	80.72	8480	45.19	80.92
DSP Slice	6.5	8528	29.49	7.54	800	27.37	7.63
RAM Block	3.12	13272	21.99	11.74	1200	27.44	11.45
Tensor Slice	28.5	0	0	0	0	0	0

Table 4.9: Resource mix in various variations of the proposed FPGA architecture. # denotes the number of blocks of a type. % denotes the percentage of area consumed by that block type.

	Prop_5pct		Prop_10pct		Prop_15pct		Prop_20pct		Prop_25pct		Prop_30pct	
Block	#	%	#	%	#	%	#	%	#	%	#	%
Logic Block	8000	45.59	7920	45.11	7760	44.37	7680	43.89	7600	43.4	7120	40.69
DSP Slice	780	28.92	720	26.68	660	24.55	600	22.3	540	20.06	500	18.59
RAM Block	1160	20.61	1040	18.47	920	16.4	800	14.25	680	12.11	640	11.4
Tensor Slice	30	4.88	60	9.74	90	14.67	120	19.55	150	24.43	180	29.33

around 170x80. The number of columns containing Tensor Slices are 3, 6, 9, 12, 15, and 18 in Prop_5pct, Prop_10pct, Prop_15pct, Prop_20pct, Prop_25pct, and Prop_30pct architectures respectively. The Tensor Slice columns are placed in the middle of the FPGA to keep the layout symmetric and to ensure shorter paths between neighboring Tensor Slices. This is an important aspect of the proposed architecture and is governed by the reduction in frequency observed when Tensor Slice columns are placed far apart in the FPGA. Figure 4.9 shows a part of the Prop_5pct FPGA showing the 3 Tensor Slice columns in this architecture.

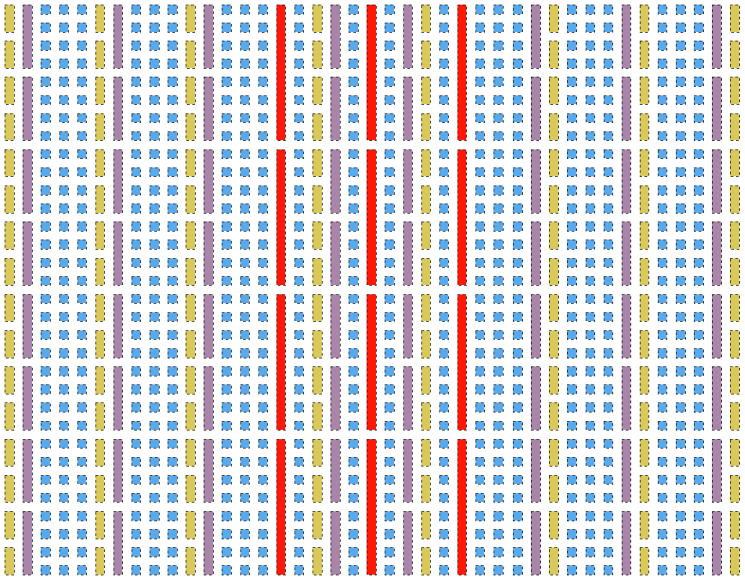


Figure 4.9: A zoomed-in version of the Prop_5pct FPGA architecture obtained from VTR. Blue: Logic Block, Yellow: RAM Block, Purple: DSP Slice, Red: Tensor Slice. This is not a physical layout; different column types have different widths in the actual layout.

4.2.5 Benchmarks

For benchmarking the proposed FPGA architecture, a set of DL and non-DL designs is used. Table 4.10 contains a list of all the benchmarks used, along with a brief description of the nature of each workload.

4.2.5.1 DL benchmarks

For DL benchmarks, several designs from the Koios benchmark suite [12] are used. These designs cover various sub-applications within DL like Multi-Level Perceptrons, Convolutional Neural Networks, Recurrent Neural Networks, etc. Design variations are added to cover multiple precisions -

int8, int16, bf16 and fp16. When evaluating DL benchmarks on the proposed FPGA(s), Tensor Slices are manually instantiated in Verilog, because the synthesis tool cannot automatically infer them. The process of conversion of existing benchmarks that use DSP slices to new benchmarks that use Tensor Slices involved:

- Identifying occurrences of matrix-matrix multiplication, matrix-vector multiplication and elementwise operations in the Verilog code of the benchmarks
- Replacing that code with Tensor Slice instances i.e. instantiating the module called `tensor_slice` with the same port list as defined in the VTR architecture file
- Designing appropriate control logic modules (Finite State Machines) that orchestrate the data movement in and out of the slice (read data stored in RAMs, feed it to the slice at the right time, write outputs to RAMs, etc.)
- Making connections between the new control logic, the Tensor Slice instances and the rest of the existing code
- Verifying the operation of the new design with simulation and comparing the results with the original unmodified benchmark

In 2 benchmarks (`lstm` and `attention`), the process of mapping to Tensor Slices involved some changes to the original architecture of the benchmark.

These designs are modified keeping them as close to the original architecture as possible - in terms of clock cycles spent, data elements consumed per cycle, etc.

In addition, the performance improvement obtained by using an FPGA with Tensor Slices is evaluated for real-world Deep Neural Networks (DNNs) from 3 common types: Fully Connected Networks (Multi-Level Perceptron (MLP)), Recurrent Neural Networks (Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU)), Convolutional Neural Networks (Tiny Darknet) and Residual Neural Networks (Resnet)). To evaluate these neural networks, a Microsoft Brainwave-like accelerator [37] is created based on Boutros et al.'s work [21]. This accelerator consists of five pipeline stages: the matrix unit (MU) for matrix-vector multiplication operations, the selector unit for skipping the MU when necessary, two multi-function units (MFUs) for vector elementwise operations (e.g. activation, addition, multiplication), and the loader (LD) which interfaces with the DRAM to load and unload data. Register files (MRF and VRF) store the data locally. Figure 4.10 shows the architecture of the accelerator. Two versions of this accelerator are created: one for the baseline FPGA and another for the proposed FPGA. For the baseline FPGA, the MU consists of dot product engines (DPEs) that contain DSP slice cascade chains. Each DPE generates 1 result. The MFUs use LB-based activation blocks, LB-based addition blocks and DSP-based multiplication blocks. The proposed FPGA in this case does not contain DSP Slices. The total area spent on DSP Slices in the baseline FPGA is instead spent on Tensor Slices

in the proposed FPGA. On the proposed FPGA, Tensor Slices are used in the MU as well as the MFU. Tensor Slices in the MU are configured in the Tensor mode, but those in the MFU are configured in the Individual PE mode.

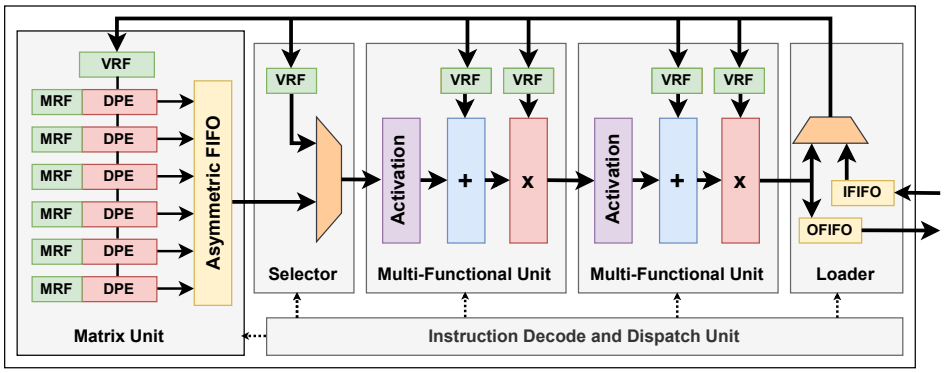


Figure 4.10: Microsoft Brainwave-like accelerator used for evaluating DNNs

An analytical model is used to estimate the cycles consumed for evaluating each network on the baseline FPGA and the proposed FPGA. A DDR4 memory with 64-bit channel at 4800 MTPS (Mega Transfers Per Second) is assumed to be connected to the FPGA. A batch size of 8 and a precision of int8 is used. The Brainwave-like accelerator does not directly support convolutions. So, for CNNs, convolution is expressed as matrix multiplication using the `im2col` operation. It is assumed that the `im2col` operation is performed in hardware. Although this can be optimized by designing an accelerator specifically for convolution, the goal here is to showcase the gains from using Tensor Slices rather than designing the most efficient accelerator.

Five DNN benchmarks are considered for this part of the evaluation. The `mlp` benchmark is a 5-layer MLP with each hidden layer having 512 neu-

rons, with 1.2M parameters. The `gru` benchmark is a GRU has a hidden size = 512, embedding size = 512, and time steps = 50. It has 1.5M parameters. The `tdarknet` benchmark is Tiny Darknet, a small image classification network for edge devices. It has 650K parameters. The `lstm_net` benchmark is an LSTM with hidden size = 1024, embedding size = 1024, and time steps = 50. It has 8.4M parameters. The `resnet` benchmark is the ResNet-50 variation of ResNet. It has 24M parameters. Out of these, the `mlp`, `gru`, `tdarknet` fit on the proposed FPGA, but `lstm_net` and `resnet` do not. For the latter, the time taken to load/unload weights is included to evaluate the speedup.

4.2.5.2 Non-DL benchmarks

For non-DL benchmarks, the VTR benchmark suite [85] is used. These designs cover several domains like computer vision, medical physics, math, finance, etc. These include designs with/without floating point operations, heavy/low DSP usage, and heavy/low/no RAM usage. The 10 largest designs in the benchmark suite (based on number of netlist primitives) are used, so that the utilization of the FPGAs (baseline and proposed) is fairly high to ensure realistic results.

Table 4.10: Non-DL and DL benchmarks used for evaluation shown in increasing order of number of netlist primitives. The netlist primitives are from when the benchmark is implemented on the baseline FPGA with DSP slices.

Benchmark	Netlist primitives	Nature of the workload
Non-DL benchmarks		
or1200	4305	Soft processor. Some DSP and RAM usage
blob_merge	11527	Image processing. No DSP or RAM usage. Only soft logic.
arm_core	18156	Soft processor. No multiplications in this design, but some RAM usage
stereovision1	29075	Computer vision. Some fixed point multiplications and no RAM usage
stereovision0	31090	Computer vision. No multiplications in this design and no RAM usage
LU8PEEng	39042	Math. Has floating point operations and some RAM usage
bgm	42293	Finance. Has floating point operations and no RAM usage
stereovision2	68683	Computer vision. Lot of fixed point multiplications and no RAM usage
LU32PEEng	128132	Math. Has floating point operations and extensive RAM usage
mcml	178845	Medical physics. Large design with multiple fixed point multiplications
DL benchmarks		
eltadd	10778	A design that adds two 6x14 int8 matrices elementwise. Inputs read from RAMs and output stored into RAMs
fcl.int	19426	Fully connected layer using int8 precision. Has num_features = 15, batch_size = 16 and num_outputs = 14. APB based control and configuration logic. RAMs store inputs and outputs.
conv.fp	22603	Convolution using 8x8 fp16 input image with 3 channels, padding=1, stride=1, filter size = 3x3 and batch size=2.
eltmul	23437	A design that multiplies two 24x8 fp16 matrices stored in RAMs, elementwise. Inputs and outputs stored into RAMs.
tpuld.16	29722	A design similar to Google's TPU v1 [65] with a 16x16 systolic array and normalize, pool and activation units. RAMs store activation and weights. Precision = int8
conv.int	32299	Same convolution design as above, but with int16 precision
attention	51858	Self-attention module in Transformers [118]. Involves matrix vector multiplication, elementwise multiplication and softmax (precison = int16)
fcl.bf	64758	Fully connected layer using bf16 precision with AXI programming interface. No RAM usage. 20x20 activation matrix, 20x20 weight matrix and 20x20 output matrix
tpuld.32	96497	Same TPU design as above, but with a 32x32 systolic array
lstm	340460	An LSTM layer [48] involving several matrix vector multiplications, elementwise operations and activations (precision = int16)

4.3 Results

4.3.1 Size

In this section, the results for experiments to identify the best Tensor Slice size are presented. Four sizes are considered - 2x2, 4x4, 8x8, 16x16, focused on the matrix multiplication operation in the Tensor Mode. The notation used is: a MxNxK matrix multiplier multiplies a MxK matrix (matrix A) with a KxN matrix (matrix B) to produce a MxN matrix (matrix C). For this experiment, a Tensor Slice with a given size only supports a specific precision, and multiple Tensor Slices are used to perform a given multiplication operation.

Two cases are considered. For the first case, a design size without fragmentation issues is considered. Figure 4.11 plots various metrics for a 32x32x32 matrix multiplier design implemented on an FPGA with different sizes of the Tensor Slice. The metrics are plotted relative to the baseline - a 32x32x32 matrix multiplier implemented on an FPGA with DSP Slices. Both int8 and fp16 precisions are considered. A lower value of all metrics is desirable. The logic area (plotted on the right axis because of a different range) and routed wirelength decrease as the Tensor Slice size increases. This is because more of the design is being hardened on moving right along the x-axis. Critical path remains relatively constant. The average wire segments per net increases as larger Tensor Slices are used. This is because the area of the Tensor Slice increases, and it occupies more locations in the FPGA grid, making wires connecting to the block longer. Additionally, routing channel

width requirement increases slightly on moving right along the x-axis.

The second case is a design with fragmentation issues. A 35x35x35 matrix multiplier is designed using 4 Tensor Slice sizes. int8 precision is used. Table 4.11 shows the results from these experiments. Because of fragmentation effects, more time is consumed when larger Tensor Slices are used. The utilization of the Tensor Slice is much higher with smaller building blocks. Utilization refers to the fraction of the number of processing elements of all the Tensor Slices performing useful work during the operation.

Tensor Slice	Freq (MHz)	Cycles	Time (us)	Utilization
2x2	540	150	0.28	0.94
4x4	500	150	0.30	0.94
8x8	470	166	0.35	0.76
16x16	465	198	0.42	0.53

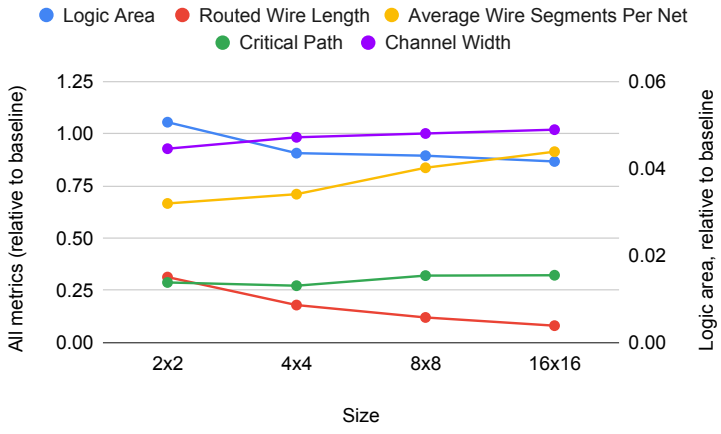
Table 4.11: A matrix multiplier with high fragmentation problems (35x35x35) designed using different Tensor Slice sizes

Considering both fragmentation effects and the metrics shown in Figure 4.11, it is concluded that sizes 4x4 and 8x8 exhibit the best tradeoffs.

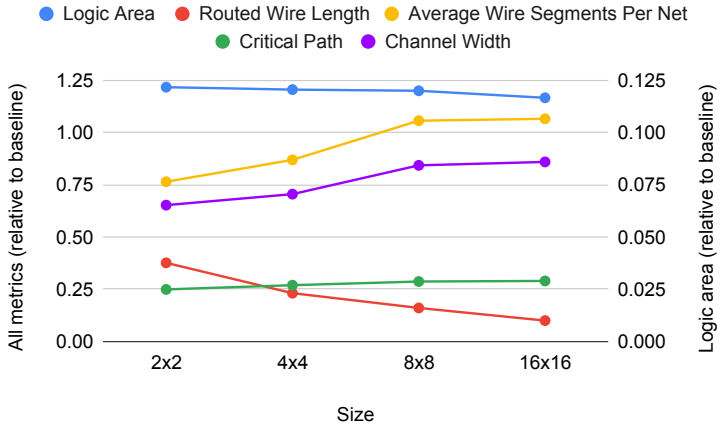
4.3.2 Peak Throughput

One of the most important benefit of adding Tensor Slices to FPGAs is to increase the compute density. That is, an FPGA with Tensor Slice has more compute throughput per unit area than a baseline FPGA. In this section, the peak throughput of the baseline and the proposed FPGAs is evaluated.

To evaluate the peak throughput, the MAC (multiply-accumulate) operation, which is the most common operation in DL applications, is considered.



(a) Precision=int8



(b) Precision=fp16

Figure 4.11: Metrics for a matrix multiplier with no fragmentation problems (32x32x32) designed using different Tensor Slice sizes

(a) INT8

(b) INT16

(c) FP16

(d) BF16

Figure 4.12: Compute throughput of the full FPGA for each precision increases as the percentage of area consumed by Tensor Slices on the FPGA is increased. 5pct, 10pct, etc denote variants of proposed FPGA architecture. The letters "Prop_" are omitted for brevity.

For LBs, one MAC is implemented on the FPGA to determine the operating frequency and number of LBs consumed. Using this and the total number of LBs on the FPGA, the total number of MACs that can be implemented on the FPGA using LBs is calculated. For DSPs, the number of MACs that can be implemented on 1 DSP Slice (for a given precision) is multiplied with the number of DSPs on the FPGA to find the total number of MACs. For Tensor Slices, the method to evaluate the total throughput is the same as the method for DSP Slices. Then the throughput from all compute resources (LBs, DSPs and Tensor Slices) is added to find the total peak MAC throughput of the FPGA. Note that while doing this calculation, it is assumed that the frequency of operation when the FPGA is filled with MACs using a compute unit is the same as the operating frequency of 1 MAC for that compute unit. This ignores the frequency degradation as the FPGA is filled, but serves the purpose of evaluating peak throughput.

Figure 4.12 shows the peak throughput for each precision supported by the Tensor Slice, obtained from each computing resource in GigaMACs per second. A significant increase is seen in the throughput by spending some area of the FPGA on Tensor Slices. For example, for the Prop_10pct case, 10% of the area of the FPGA is spent on Tensor Slices, and it can be seen that the peak compute throughput increases by $\sim$1.86$\times$ for int8 and $\sim$1.42$\times$ for int16, fp16 and bf16 precisions. For Prop_30pct variation of the proposed architecture, the throughput increases by 3.5$\times$ for int8, 2.2$\times$ for int16, 2.18$\times$ for fp16 and 2.17$\times$ for bf16.

Table 4.12: Resource usage of various benchmarks. Resource usage is the same for all variants of the proposed FPGA architecture, hence there is only one moniker used here: "Proposed"

	Logic Blocks		**DSP Slices**		**RAM Blocks**		**Tensor Slices**	
Benchmark	Baseline	Proposed	Baseline	Proposed	Baseline	Proposed	Baseline	Proposed
`arm_core`	836	836	0	0	40	40	0	0
`bgm`	2132	2132	11	11	0	0	0	0
`blob_merge`	540	540	0	0	0	0	0	0
`LU32PEEng`	5890	5890	32	32	274	274	0	0
`LU8PEEng`	1728	1728	8	8	73	73	0	0
`mcml`	6792	6792	106	106	294	294	0	0
`or1200`	202	202	4	4	4	4	0	0
`stereovision0`	582	582	0	0	0	0	0	0
`stereovision1`	490	490	40	40	0	0	0	0
`stereovision2`	1958	1958	483	483	0	0	0	0
`attention`	1706	757 (0.44×)	73	9	188	188	0	16
`conv.fp`	630	243 (0.38×)	75	0	56	56	0	7
`conv.int`	913	243 (0.26×)	42	0	56	56	0	7
`eltadd`	287	71 (0.25×)	0	0	24	24	0	2
`eltmul`	630	206 (0.32×)	96	0	48	48	0	6
`fcl.bf`	2017	788 (0.39×)	200	0	0	0	0	25
`fcl.int`	464	97 (0.20×)	112	0	24	24	0	4
`lstm`	6982	1883 (0.27×)	642	2	532	532	0	160
`tpuld.16`	744	134 (0.18×)	148	0	14	14	0	4
`tpuld.32`	2440	258 (0.10×)	552	0	26	26	0	16

4.3.3 Resource Usage

Table 4.12 shows the resource usage obtained from VTR for the various benchmarks when implemented on the baseline and proposed FPGAs. For DL benchmarks, the usage of LBs and DSPs reduces greatly with the usage of Tensor Slices. The highest reduction in LB usage is in `tpuld.32` with 0.10× usage (i.e. 90% reduction from baseline architecture). The same number of blocks are used by the benchmarks across the variants of the proposed FPGA; so only one column for Proposed is shown. For non-DL benchmarks, there is

no difference in the resource usage between the baseline and proposed FPGAs. This is because the designs do not have any matrix operations in them and use the same blocks in both types of FPGAs. A larger percentage of Tensor Slice area can lead to insufficient resources for a large non-DL design, and hence the design may not fit. However, for large non-DL designs, it is better to choose an FPGA from a device family that is oriented towards non-DL applications, instead of using a DL-optimized FPGA that has Tensor Slices.

4.3.4 Area

The total area consumed by a circuit on an FPGA is the sum of the logic area and the routing area. Logic area is available in the VTR output report, but routing area is not. The routing area is estimated approximately by adding the routing area of all tiles that have at least one operation mapped to.

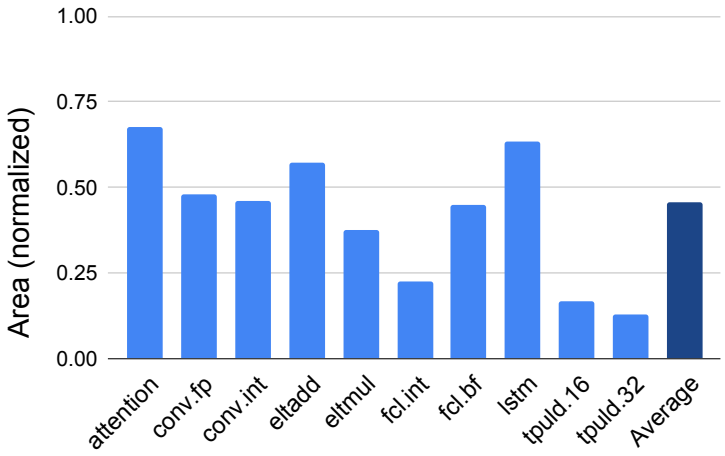


Figure 4.13: Comparison of used area for various benchmarks. Non-DL benchmarks not shown because of no change in their area.

For DL benchmarks, the total used area reduces significantly (Figure 4.13). This follows directly from the usage of Tensor Slices instead of LBs and DSP slices. On average, area reduces to 0.45×. The best case is 0.22× for `fcl.int`. Tensor Tiles harden the circuitry that would otherwise be implemented in soft logic and DSP slices. These results (along with Routed Wirelength results in Section 4.3.6) provide a first order approximation of the potential power reduction that can be achieved by using Tensor Slices. The area reduction does not differ significantly for different variations of the proposed FPGA architecture, similar to resource usage.

Non-DL benchmarks do not show any change in total area[3]. Logic area is not expected to change because the resource usage does not change. The routing area may change slightly because of the presence of Tensor Tiles. However, the routing area model used is approximate as it only considers the routing area of the tiles that have at least one operation mapped to, which also stays constant for non-DL benchmarks. Therefore, the total area improvement for only DL benchmarks is shown in Figure 4.13.

4.3.5 Frequency

Figure 4.14 (a) shows the improvement in the frequency of operation of DL benchmarks on a proposed FPGA compared to the baseline FPGA. An increase in frequency implies a reduction in execution time at the application

[3]Note that because the proposed FPGA has fewer LBs, DSPs and BRAMs than the baseline FPGA, some large benchmarks may not fit the proposed FPGA. An analysis of such a case is done in Section 4.3.10

level. A maximum frequency improvement of 2.43× for the `conv.fp` benchmark and an average improvement of 1.63× across benchmarks is seen. This boost happens because on the baseline FPGA, the critical paths include long paths across LBs and DSP slices, which are inside the hard Tensor Slice on the proposed FPGA. The achieved frequency does not change significantly across different variations of the proposed FPGA architecture (i.e. with different area percentage consumed by Tensor Slices on the FPGA), so results for one variation are shown in the figure.

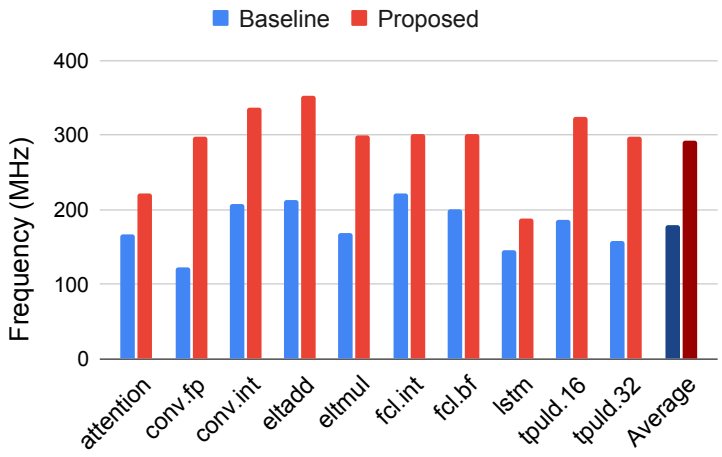
In non-DL benchmarks, the frequency degrades when Tensor Slices are added, although not significantly. Some degradation is expected because the presence of Tensor Slices can increase the routing wire length required to route a circuit, causing an increase in the critical path delay. Figure 4.14 (b) shows the sensitivity of frequency of operation for non-DL benchmarks as the area spent on Tensor Slices increases in the different variations of the proposed FPGA. An average degradation of 2.3% is observed for Prop_30pct variation of the proposed architecture. The maximum degradation observed is 7.9% in the `blob_merge` benchmark. Reduction in frequency implies increase in execution time. If a large non-DL design can not utilize Tensor Slices (e.g. in Individual PE mode), then it may not fit on an FPGA where a large portion of the area is consumed by Tensor Slices. To fit the design on the FPGA, some parallelization may have to be reduced (if possible, based on the nature of the design), leading to the application consuming more time. However, for large non-DL designs, it is better to choose an FPGA from a device family that is

oriented towards non-DL applications, instead of using a DL-optimized FPGA that has Tensor Slices.

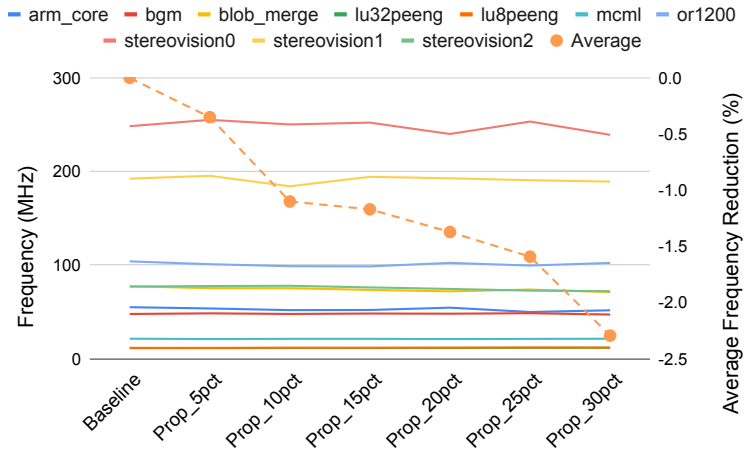
4.3.6 Routed Wirelength

Figure 4.15 shows the impact of using Tensor Slices on the total routed wirelength for various benchmarks. For DL benchmarks (Figure 4.15 (a)), a significant reduction in routed wirelength is seen. This is because a lot of wiring required to connect soft logic and DSP slices on the baseline FPGA is effectively absorbed inside the hard Tensor Slice. On average, the routed wirelength reduces to 0.45× (55% reduction). In some circuits where the portion of the design that is mapped to Tensor Slices is large, the routed wirelength reduction is very high. E.g. for `tpuld.32` design, the routed wirelength is reduced by $\sim$ 90%. In other designs which have a significant portion of the design that is not mapped to Tensor Slices, the routed wirelength reduction is low, e.g. wirelength reduction of $\sim$ 35% and $\sim$ 31 % is seen in `lstm` and `attention` designs respectively. These results (along with area results in Section 4.3.4) provide a first order approximation of the potential power reduction that can be achieved by using Tensor Slices. Different variations of the proposed FPGA show similar reduction in routed wirelength and hence, they are not shown in the figure.

For non-DL benchmarks (Figure 4.15 (b)), the routed wirelength increases slightly. Adding a large block on the FPGA can increase wire length required to route a design that does not use the large block. An average in-

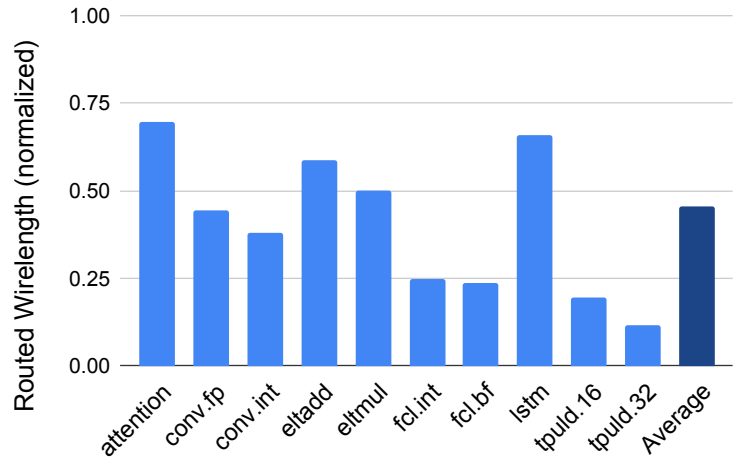


(a) DL benchmarks

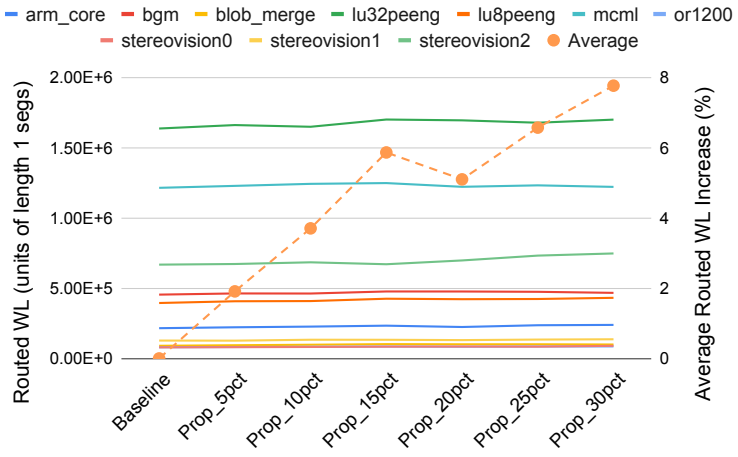


(b) Non-DL benchmarks

Figure 4.14: Comparison of achieved frequency of operation for various benchmarks



(a) DL benchmarks



(b) Non-DL benchmarks

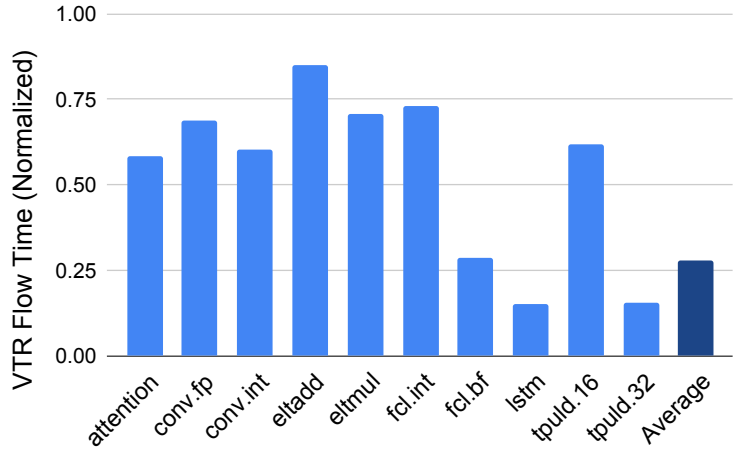
Figure 4.15: Comparison of routed wirelength used for various benchmarks

crease of 1.9% is seen across non-DL benchmarks for the Prop_5pct FPGA architecture. As the percentage of area consumed by the Tensor Slices in the variations of the proposed FPGA architecture is increased, the routed wirelength increases. That's because longer net lengths are required to make connections between blocks with an increasing number of Tensor Slices in the fabric. An average increase of 7.7% in routed wirelength is seen in the Prop_30pct FPGA architecture. The maximum increase in routing wirelength of 14.1% is seen in `stereovision0`) in this architecture.

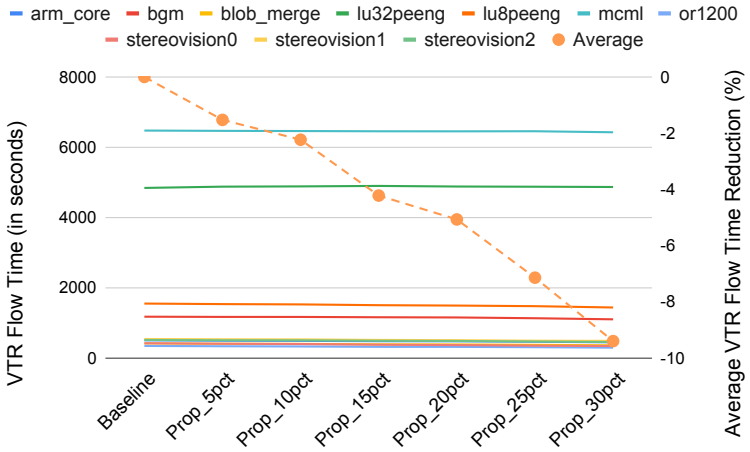
4.3.7 VTR Flow Run Time

Tensor Slices are large hard blocks that make the FPGA more coarse-grained. A design using Tensor Slices can go through the FPGA tool chain (synthesis, packing, placement, routing) faster. Having Tensor Slices in the FPGA fabric reduces the total number of blocks on the FPGA (for the same area), and hence reduces exploration space available to the packing, placement and routing algorithms, making the runtime shorter. Also, Tensor Slices are instantiated as hard macros in the RTL. There is no behavioral code that is analyzed to infer Tensor Slices. This reduces the runtime of the synthesis engine.

Figure 4.16 shows the VTR CAD flow time for various benchmarks. The time taken to run the VTR CAD flow reduces by an average of 73% for DL benchmarks when using Tensor Slices. Since the time taken does not change significantly across different variants of the proposed FPGA architecture, the



(a) DL benchmarks



(b) Non-DL benchmarks

Figure 4.16: Comparison of VTR flow run time for various benchmarks

results for all variants are not shown in the figure. For non-DL benchmarks, there is a trend in the time taken by the CAD flow. As the percentage of area consumed by Tensor Slices on the FPGA increases, the time taken by the flow reduces. A reduction of up to 9.4% is seen for the Prop_30pct architecture.

4.3.8 Routing Channel Width

In all the results discussed so far, a fixed routing channel width of 300 is used for all the FPGA architectures. In this section, the routing channel width is varied to study the impact of adding Tensor Slices on channel width. VTR provides a mode where it finds the minimum channel width required to route a circuit on a given FPGA architecture. Minimum channel width requirement is influenced by the pin density of the building blocks in an FPGA architecture. Blocks with small area and large number of I/Os can cause routing congestion and hence lead to higher minimum channel width requirement. A large block with a large perimeter (or area), like the Tensor Slice, does not negatively impact the routability, even with many I/Os. If a block disrupts the routing fabric by not allowing wiring to cross it, then that can cause routability issues as well. The Tensor Slice tiles are defined to have full switch boxes outside and straight switch boxes inside the block [85], alleviating the routability impact. Note that this requires careful physical layout of the Tensor Slice block. Adding a local crossbar inside the Tensor Slice blocks helps with improving the routability of the proposed FPGA as well.

Figure 4.17 shows the minimum routing channel width required for var-

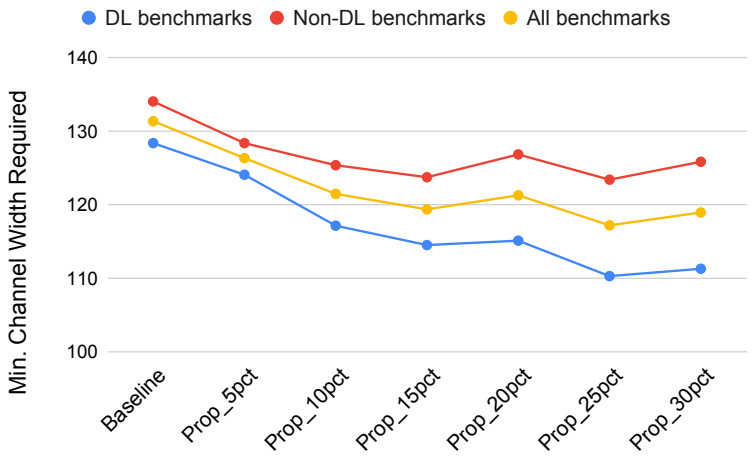


Figure 4.17: Minimum routing channel width required for the baseline FPGA architecture and the different variations of the proposed FPGA architecture, averaged across different categories of benchmarks

ious FPGA architectures, averaged across the benchmarks. DL benchmarks have a lower channel width requirement compared to non-DL benchmarks, even for the baseline FPGA. As the area consumed by Tensor Slices increases (in variations of the proposed FPGA architecture), the channel width requirement reduces. This is because of the more spread out placement of the blocks used for the circuits because of the presence of more and more Tensor Slices (which leads to increase in routed wirelength, as seen in Section 4.3.6).

4.3.9 FPGA Grid Area

In all the results discussed so far, fixed FPGA grid areas (number of columns × number of rows) are used. The grid dimensions are calculated such that the FPGA is capable to fit the total number of blocks of each type

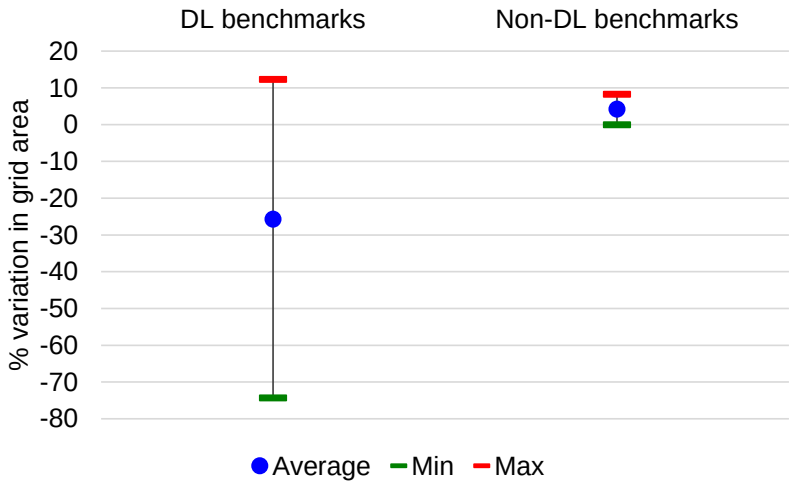


Figure 4.18: Variation in the FPGA grid area observed when comparing Proposed(Auto) FPGA architecture relative to the Baseline(Auto) FPGA architecture, across DL and non-DL benchmarks

mentioned in Tables 4.8 and 4.9. In this section, the grid dimensions of the FPGA are varied to observe how the minimum FPGA size required for a benchmark changes with the addition of Tensor Slice columns in the FPGA architecture. VTR provides a mode called `auto_layout`, where it expands the grid dimensions (by repeating a specified set of columns and rows) to find the minimum FPGA size required to successfully implement a circuit on the given FPGA architecture. With the introduction of Tensor Slices (i.e. replacing some columns with columns containing Tensor Slices in the FPGA), the total number of columns required to implement a circuit that does not use Tensor Slices can increase. In other words, to fit the same design, a larger FPGA may be required. This can show the impact of adding Tensor Slices, especially on non-DL designs.

For this study, a set of columns is specified in the FPGA architecture, and this set is repeated over and over by VTR to satisfy the resource requirement of a design. For the baseline architecture, this set includes 11 Logic Block columns, 4 DSP Slice columns and 3 RAM Block columns. For the proposed architecture, this set includes 11 Logic Block columns, 4 DSP Slice columns, 3 RAM Block columns and 1 Tensor Slice column. These architectures are referred to as Baseline(Auto) and Proposed(Auto) respectively.

Figure 4.18 shows the results from this study. For DL benchmarks, the average grid area required reduces by 26% when the Proposed(Auto) architecture is used, compared to when the Baseline(Auto) architecture is used. This is because DL benchmarks use Tensor Slices and a smaller number of columns are now required to fit the same design. The minimum reduction in FPGA grid area required is seen to be 74.2%. However, an interesting behavior is seen. For DL benchmarks, it is observed that the FPGA grid area required increases in some cases, the maximum increase being 12.3%. This happens because of the arrangement of Tensor Slice columns in the Proposed(Auto) architecture. As the FPGA grid size is increased by VTR, to get an additional Tensor Slice column, the grid size has to be increased by 19 columns, even if 18 out of 19 columns may not be utilized because they contain other resources. The Tensor Slice columns are equally spread out in the architecture. This illustrates the importance of having Tensor Slice columns close to each other like in the Prop_Xpct architectures, instead of being spread out on the entire FPGA. For non-DL benchmarks, however, the average grid area required increases by

4.3% when the Proposed(Auto) is used. A larger FPGA is now required to fit all the designs.

4.3.10 Non-DL benchmarks when sufficient DSP Slices are not available

Adding Tensor Slices on an FPGA takes away area from other resources like Logic Blocks, DSP Slices and RAM Blocks. This implies that for large non-DL designs requiring many DSP Slices, the number of DSP Slices in a proposed FPGA may be less than the number of DSP Slices required by the design. In such cases, Tensor Slice's Individual PE mode can be used. Note that the Individual PE mode of the Tensor Slice is not as performant as a DSP Slice, because of the lower frequency of operation of the Tensor Slice compared to the DSP Slice, and it supports smaller precisions only (int8, int16, fp16 and bf16) compared to larger precisions supported by DSP Slice (e.g. 27x27 and fp32). Note that the frequency of operation of the Tensor Slice in this mode can be improved by using multiple local input crossbars as mentioned in Section 4.2.3. In this section, the impact of using Tensor Slice's Individual PE mode is evaluated.

Table 4.13: Tensor Slices used by a non-DL design in Individual PE mode when sufficient DSP Slices are not available

Block	Baseline	Prop_30pct
Logic Blocks	3107	2713
RAM Blocks	0	0
DSP Slices	599	298
Tensor Slices	0	108

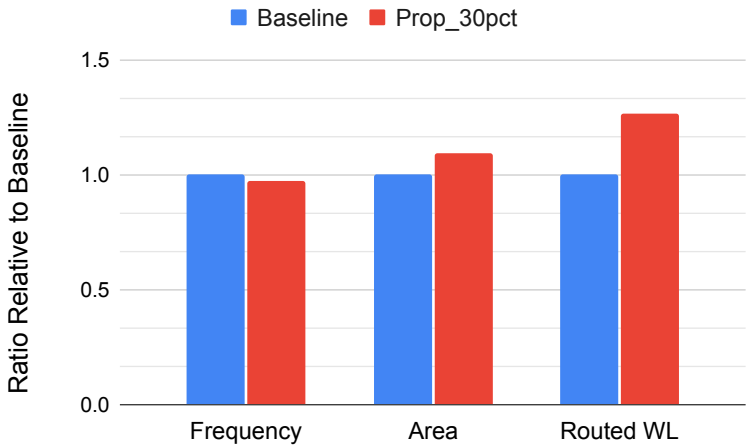


Figure 4.19: Comparing various metrics for a non-DL design when implemented on a baseline FPGA using DSP Slices vs. proposed FPGA using Individual PE mode of Tensor Slices for some compute.

Since none of the benchmarks in the VTR benchmark suite is very DSP-intensive, a synthetic benchmark is created by instantiating multiple stereovision designs (1 stereovision2 instance and 3 stereovision1 instances). This design requires 599 DSPs on the baseline FPGA (Table 4.13). The worst case is considered by using the Prop_30pct FPGA for this experiment. In the Prop_30pct FPGA, there are only 500 DSP Slices. So, when this design is implemented on the Prop_30pct architecture, some operations get mapped to the Tensor Slice (in Individual PE mode). Table 4.13 shows that 298 DSP Slices get used, and 108 Tensor Slices get used.

The results of this study are shown in Figure 4.19. A frequency degradation of 3% is seen, whereas the area increases by 9% and the routed wirelength increases by 25%. This shows the usefulness of Individual PE mode

Table 4.14: Resource usage of the `lstm` benchmark for the baseline architecture and different variations of the proposed architecture

Block	Baseline	Prop_5pct	Prop_10pct	Prop_15pct	Prop_20pct	Prop_25pct	Prop_30pct
Logic Blocks	6982	6285	5388	4479	3724	2816	1883
RAM Blocks	532	532	532	532	532	532	532
DSP Slices	642	562	450	349	242	130	2
Tensor Slices	0	20	48	76	100	128	160

of the Tensor Slice. This design would not have otherwise fit on this FPGA. But because of Individual PE mode, this design can be implemented on the proposed FPGA (even the one that has 30% area spent on Tensor Slices) with a minor performance degradation. The increase in area and routing wirelength actually indicates an improved utilization of the FPGA in this case, because these resources would have been lying idle otherwise.

4.3.11 DL benchmarks when sufficient Tensor Slices are not available

In the experiments shown so far, different variations of the proposed FPGA are evaluated - the area of the FPGA spent on Tensor Slices increases from 5% to 30% in increments of 5%. A question arises: What happens when the number of Tensor Slices on the FPGA is less than the Tensor Slices required by a design? The answer is that computation has to be mapped to DSP Slices and Logic Blocks instead. In this section, the impact on various metrics in such a scenario is studied.

For this experiment, the `lstm` benchmark is considered, as it requires the largest number of Tensor Slices. This design is implemented on the baseline architecture and different variations of the proposed architecture. On the

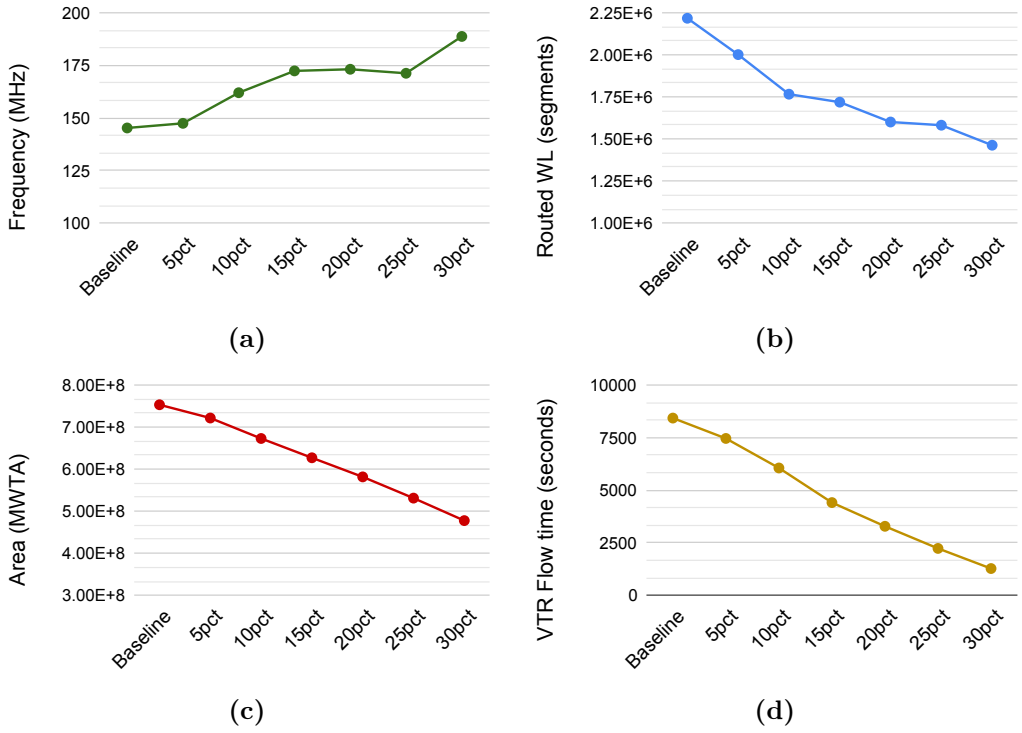


Figure 4.20: Performance results of mapping a DL benchmark to FPGAs with increasing number of Tensor Slices. 5pct, 10pct, etc denote variants of proposed FPGA architecture. The letters "Prop_" are omitted for brevity.

baseline FPGA, all computation is mapped to DSP Slices and Logic Blocks. More and more computation is mapped onto Tensor Slices depending on the number of available Tensor Slices in the variations of the proposed architecture. On the Prop_30pct FPGA, all computation that could be mapped to Tensor Slices is mapped to Tensor Slices. This is shown in Table 4.14.

Figure 4.20 shows the results from this experiment. When more computation is mapped to DSP Slices, (1) the frequency is lower, (2) more area of the FPGA is consumed to implement the circuit, and (3) more routed wire-

length is used. When enough Tensor Slices are not available, computation can be mapped to DSP Slices and Logic Blocks at the cost of reduced performance.

4.3.12 DNN Evaluation

Figure 4.21 shows the speedups obtained for various DNNs using a proposed FPGA with Tensor Slices compared to a baseline FPGA with DSP Slices. Note that as mentioned in Section 4.2.5.1, the proposed FPGA in this case does not contain DSP Slices. The total area spent on DSP Slices in the baseline FPGA is instead spent on Tensor Slices in the proposed FPGA. This leads to ∼27% area of the FPGA being consumed by Tensor Slices.

A geomean speedup of 2.3× is observed across the 5 DNNs. The `lstm_net` workload shows the least speedup because of two reasons: (1) the weights do not fit on the FPGA and loading the weights takes a significant amount of time, and (2) there is not enough reuse of weights to amortize the time for loading the weights. The highest speedup is seen for the `gru` workload.

The peak throughput enhancement for the FPGA with Tensor Slices is 3.5×. The reasons for why the achieved speedup is lower are: (1) some layers in the workloads are limited by DRAM bandwidth, and (2) underutilization of Tensor Slices.

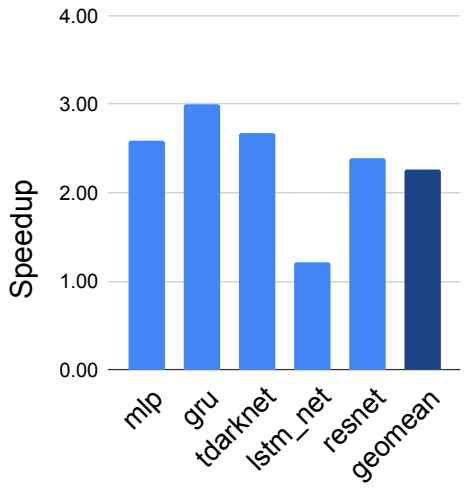


Figure 4.21: Speedup obtained by using an FPGA with Tensor Slices compared to an FPGA with DSP Slices

4.4 Discussion

4.4.1 Benefits and Limitations

Qualitatively, the benefits of adding Tensor Slices to FPGAs can be summarized as:

- Adding Tensor Slices to an FPGA increases the compute density of the FPGA. The area consumed by DL designs is smaller when Tensor Slices are used. This implies larger DL designs can now fit the same-sized FPGA chip.
- Designs using Tensor Slices can achieve faster frequencies compared to those using DSP Slices, because of the reduced dependence on routing/interconnect. This means DL applications can run faster on FPGAs with Tensor Slices.
- Using Tensor Slices leads to a reduction in routing wirelength required to implement a DL design. This implies a reduction in power consumption as well.
- Tensor Slices are large coarse-grained blocks. A design using Tensor Slices can go through the FPGA tool chain (synthesis, packing, placement, routing) faster. This means shorter turn around time for debug iterations.

Overall, Tensor Slices lead to better out-of-the-box performance for DL applications. This can lower the barrier of adoption for FPGAs for DL

acceleration. Note that, currently, to use Tensor Slices, a designer has to manually instantiate a Tensor Slice block in the RTL and connect it. This is commonly how DSP Slices are used as well, other than for simple use cases like multiplication and MAC.

There are some limitations of adding Tensor Slices to FPGAs as well. First, adding Tensor Slices makes an FPGA more heterogeneous (any new type of block added to the FPGA fabric adds to heterogeneity). This increases the complexity in CAD tool algorithms, especially at the implementation stage (pack, place, route). However, this cost is typically transparent to end-users of the FPGA. Secondly, adding Tensor Slices makes an FPGA less generic or flexible compared to a typical FPGA. If a Tensor Slice is not required or can not be used by an application (even in individual PE mode), the Tensor Slices will remain unutilized on the FPGA. But with the abundance of DL applications, DL-specialized FPGA families containing Tensor Slices can be created. And non-DL applications can continue to use non-DL-specialized FPGAs. Thirdly, adding Tensor Slices to an FPGA causes some degradation in performance for non-DL applications, as shown in Section 4.3.

4.4.2 Comparison with DSP Slice

Table 4.15 compares I/O and area related properties of the DSP Slice with the Tensor Slice. The Tensor Slice is about 4.3× in area compared to the implementation of an Agilex-like DSP Slice, but has only 1.5× the number of I/O pins. This means that the Tensor Slice has a low pin density or that

Table 4.15: Comparing I/O and area related metrics for a Tensor Slice with a DSP Slice (numbers based on 22 nm tech node)

Metric	DSP Slice	Tensor Slice	Ratio
Area (um^2)	12597	55218	4.3
Number of I/Os	386	608	1.5
Pin density (pins/um^2)	0.03	0.01	0.3
Logic to routing area ratio	1.5	4.5	3

Table 4.16: Comparing compute throughput related metrics of a Tensor Slice with a DSP Slice (numbers based on 22 nm tech node)

Precision	Metric	DSP Slice	Tensor Slice	Ratio
INT8	Number of MACs	4	64	16
	Freq (MHz)	429	391	0.9
	Throughput (GigaMACs/sec)	1.7	25.0	14.6
	Throughput/area (GigaMACs/sec/mm^2)	136.2	453.2	3.3
INT16	Number of MACs	2	16	8
	Freq (MHz)	429	391	0.9
	Throughput (GigaMACs/sec)	0.8	6.2	7.3
	Throughput/area (GigaMACs/sec/mm^2)	68.1	113.3	1.6
FP16/BF16	Number of MACs	2	16	8
	Freq (MHz)	341	302	0.9
	Throughput (GigaMACs/sec)	0.7	4.8	7.0
	Throughput/area (GigaMACs/sec/mm^2)	54.1	87.5	1.6

the Tensor Slice has a high core-to-pin ratio. The Tensor Slice also has about 3× the logic-to-routing ratio, compared to the DSP Slice. The routing area includes the local crossbar, the switch boxes and the connection boxes. This implies significantly lower routing overhead compared to a DSP slice.

Table 4.16 compares throughput related properties of the DSP Slice with the Tensor Slice. The Tensor Slice has over 14× int8 throughput and over 7× throughput for 16-bit precisions, compared to an Agilex DSP slice. Note that because of quantization/fragmentation, the Tensor Slice can suffer from

under-utilization (reduced effective throughput) in cases where the problem size does not divide up evenly into the dimensions of the Tensor Slice. For example, to compute a 7x7 int8 matrix-matrix dot product using the Tensor Slice, 15 out of the 64 PEs will be effectively wasted. But with the large matrix sizes commonly required for DL applications, this performance loss is not a significant issue in real-world applications.

4.4.3 Comparison with Intel AI Tensor Block

Intel Stratix NX FPGAs contain a new block called the AI Tensor Block [75] [57]. The AI Tensor Block[4] is a replacement of the DSP Slice, in that it exactly matches the I/Os and the area of a DSP Slice. Each Tensor Block contains three dot product units, each of which has 10 multipliers and 10 accumulators. A Tensor Block has 7.5x more int8 compute compared to an Intel Agilex DSP Slice. Multiple blocks can be cascaded to support larger matrices.

AI Tensor Blocks are similar to Tensor Slices in that both these blocks are integrated into the programmable logic. However, there are many differences. A qualitative and quantitative comparison of the two types of blocks is shown below. This notation is used: A matrix-vector multiplication (MVM) involves multiplying M×K matrix with a K×1 vector and a matrix-matrix

[4]Intel Stratix NX FPGAs were announced in June 2020 and were not generally available until 2021. The first submission based on this dissertation for an FPGA with matrix multiplier blocks was made to the ACM International Symposium on FPGAs (ISFPGA) in September 2019 [15] and published in the International Conference on Application-specific Systems, Architecture and Processors (ASAP) conference in July 2020 [8].

multiplication (MMM) involves multiplying a M×K matrix A with a K×N matrix B.

1. AI Tensor Blocks are smaller (1× an actual Stratix 10 DSP) compared to Tensor Slices (4.3× Agilex-like DSP). Note that Tensor Slice's implementation is full-featured, but the DSP Slice implementation does not have all the features that are present in Intel's DSP slices (as mentioned in Section 4.2.2). As an approximation, 4.3x can be used as the ratio of the area of a Tensor Slice and an AI Tensor Block. An AI Tensor Block has 30 int8 MAC units, but a Tensor Slice has 64 int8 MACs. AI Tensor Blocks have lower compute throughput compared to Tensor Slices, but higher throughput per unit area.

2. Information about AI Tensor Block operating frequency is not available in the Intel Stratix 10 datasheet. In [75], the frequency used for calculating peak throughput is 600MHz, but it is not clear if this is the maximum frequency of operation of the AI Tensor Block. Optimistically it can be assumed to be the same as the Stratix 10 DSP Slice, which is 1000 MHz for fixed-point and 750 MHz for floating-point (in 14 nm technology node). Scaling to 22 nm (which is the technology node used for evaluation in this chapter) using equations from [108], this comes out to be 442 MHz for fixed-point and 331 MHz for floating-point. This is faster than the Tensor Slice.

3. Tensor Slices and AI Tensor Blocks support different set of precisions.

The base precisions supported on AI Tensor Block are int8 and int4. There is no native floating point support, but instead, there is shared exponent support for block floating point fp16 and fp12 [127]. Tensor Slices, on the other hand, completely supports int8, int16, fp16 and bf16 precisions. The only common precision between the two is int8.

4. Tensor Slice and AI Tensor Block have different bandwidth requirements. That is, the number of bytes read/written per cycle by each slice to perform a computation is different. For matrix-matrix multiplication, Tensor Slices read the inputs only once because of their systolic architecture. So, the inputs can be streamed in, for example, from the previous layer's logic on the FPGA. However, in case of the AI Tensor Block, one of the input matrices is read multiple times, so it has to be stored and fed to the block. For matrix-vector multiplication, Tensor Slices and AI Tensor Block both only read the inputs once. However, in both cases, to get more speedup, more blocks can be used and that typically involves reading the inputs once and fanning them out to the multiple blocks.

5. Both Tensor Slice and AI Tensor Block suffer significant under-utilization for matrix-vector multiplication, as will be seen in the quantitative comparison later in this section. Tensor Slice has a utilization of 25% for int8 precision, but for 16-bit precisions (int16, fp16, bf16), its utilization is 50%. AI Tensor Block has a utilization of 33% for any precision for matrix-vector multiplication. For matrix-matrix multiplication, 100%

utilization can be achieved on both the blocks. For the Tensor Slice, rounding up or zero padding the problem size to the nearest multiple of 8 is needed for int8 precision and to the nearest multiple of 4 for 16-bit precisions. This is true for all 3 dimensions - M, N, and K. For the AI Tensor Block, rounding up or zero padding to the nearest multiple of 6 along the N dimension and nearest multiple of 10 in the K dimension leads to maximum efficiency.

For quantitative comparison, two workloads are considered - one is matrix-vector multiplication (MVM) and the second is matrix-matrix multiplication (MMM). The problem size considered is such that there are no fragmentation effects in either the AI Tensor Block or the Tensor Slice (i.e. the problem dimensions have to be a multiple of 6, 10 and 8). Table 4.17 shows the properties and metrics for these workloads. The int8 precision is used because that's the only common precision between the two blocks.

An interplay between cycles consumed, number of blocks used and the bandwidth requirements is seen. For the MVM workload, Tensor Slice takes a smaller number of cycles, but more blocks are required compared to the AI Tensor Block. More blocks means more routing interconnect will be required and the frequency of operation will likely be lower. The total bytes read during the process is the same (both matrix and vector are read only once). For the MMM workload, two implementations which differ in how the blocks are cascaded to perform a large MMM are shown. In implementation #1, only

Table 4.17: Comparing Intel AI Tensor Block (AI.T.Block) with Tensor Slice (T.Slice) for 2 workloads: a matrix-vector multiplication (MVM) and a matrix-matrix multiplication (MMM). There are two implementations for MMM: imp #1 and imp #2. They differ in how multiple blocks are cascaded. Note that this is only for int8 precision. The area is in terms of number of DSPs and only includes the area of the blocks, not the routing wires.

	MVM		MMM (imp #1)		MMM (imp #2)	
	AI.T.Block	T.Slice	AI.T.Block	T.Slice	AI.T.Block	T.Slice
M	480	480	480	480	480	480
K	480	480	480	480	480	480
N	1	1	480	480	480	480
Number of blocks used	49	60	49	3600	3920	1800
Approximate area	49	258	49	15480	3920	7740
Clock cycles	674	274	76996	1920	1156	3840
Utilization	33%	25%	100%	100%	100%	100%
Peak i/p BW (elements in 1 cycle)	480	482	490	960	1280	720
Peak o/p BW (elements in 1 cycle)	10	480	30	480	2400	240
Total bytes read	230880	230880	37094400	460800	18892800	691200
Fanout for matrix B (or vector)	1	1	1	1	1	1
Fanout for matrix A	1	2	1	1	80	1

Note that neither a board with an Intel Stratix NX FPGA nor Intel Quartus CAD tool with support for Stratix NX FPGAs was available to be able to design and implement these designs and obtain metrics such as resource usage and achieved frequency.

a few AI Tensor Blocks are used compared to Tensor Slices, and so the AI Tensor Blocks take a large number of cycles, while also reading a much larger number of bytes (matrix A has to be read multiple times). On the other hand, Tensor Slices take a very small number of cycles, but the number of blocks required is very high. In implementation #2, for the AI Tensor Block case, more blocks are used leading to a significant reduction in cycles, but a much higher fanout requirement for matrix A and also higher input (i/p) and output (o/p) bandwidth is required. Higher fanout generally means a lower frequency of operation and higher bandwidth requirement implies more routing interconnect usage. More blocks also means more routing wires required to connect the blocks, which can nullify the area advantage the AI Tensor Block has over Tensor Slice in this case. In implementation #2 for the Tensor Slice case, half the number of blocks compared to implementation #1 are used, which doubles the number of cycles required. Both Tensor Slice based implementations read much smaller number of bytes compared to the AI Tensor Block based implementations because of the systolic architecture of the slice. This implies reduced dynamic power consumption. Tensor Slices consume more area in both implementations.

Let us consider two smaller workloads as well. First one is an MMM of size M=12, K=20, N=12. This workload has no fragmentation when the AI Tensor Block is used. 3 AI Tensor Blocks are required, the number of cycles taken is 60, and the total elements read are 1200. When computed using Tensor Slices, 4 blocks are required, 80 cycles are consumed, and 768 elements

are read. The second workload is an MMM of size M=16, K=16, N=16. This workload has no fragmentation when the Tensor Block is used. 4 Tensor Slices are required, the number of cycles taken is 64, and the total elements read are 512. When computed using AI Tensor Block, 3 blocks are required, 108 cycles are consumed, and 2280 elements are read.

Overall, there is no clear winner and both blocks have their positives and negatives. There are many ways in which both the blocks can be connected to perform MVMs and MMMs and the best solution depends on the requirements of the application. From the point of view of approximate area (i.e. only block area, not the area of routing wires required to connect the blocks), AI Tensor Blocks are a better choice. Either block can be chosen based on cycles. If bytes read (and dynamic power) is a concern, then Tensor Slices are a better option. Note that these comparisons are valid only for int8 precision.

4.4.4 Comparison with Xilinx AI Engine

Xilinx Versal ACAPs (Adaptive Compute Acceleration Platform) add an array of AI engine tiles [132] [134] to the FPGA chip. Each AI engine tile contains an AI engine and a data memory. The AI engine contains a SIMD (Single Instruction Multiple Data) VLIW (Very Long Instruction Word) vector processor. The data memory is 32 Kilobytes. An AI engine can access the data memory of its near neighbors as well. The AI engine array can communicate with the programmable logic via AXI stream and memory mapped interfaces

over a NoC (Network-On-Chip). An analytical comparison of the Tensor Slice and the AI engine is performed in this section:

1. Integrating AI engines on an FPGA is similar to adding CPUs on the same die, whereas integrating Tensor Slices on an FPGA is similar to adding DSP slices in the FPGA's programmable logic. AI engines are not tightly coupled with the FPGA programmable logic. They follow a different computation paradigm. This approach is a software centric approach to acceleration. It does make programming the FPGA easier, but it brings with it the overheads of instruction pipelines like addressing, decode, etc., as well as additional overheads in communicating with compute/control units designed in the programmable logic. Tensor Slice, on the other hand, enables tighter integration with any custom logic implemented in the programmable logic.

2. The precisions supported by the AI engine are 8-bit fixed-point, 16-bit fixed-point, 32-bit fixed-point and IEEE floating point 32-bit. It does not support smaller floating-point precisions like fp16 and bf16, which are commonly used in DL workloads and are supported by the Tensor Slice.

3. For 8-bit fixed-point operands (16-bit accumulation), the AI engine's peak throughput is 128 MACs/clock. For 16-bit fixed-point operands (48-bit accumulation), the AI engine's peak throughput is 32 MACs/-clock. For both these precisions, the throughput of the AI engine is 2×

that of a Tensor Slice.

4. The AI engine tiles work at 1 GHz at 7 nm technology node. Scaling to 22 nm (which is the technology node used for evaluation in this chapter) using equations from [108], this comes out to be around 300 MHz, which is lower than the frequency of the Tensor Slice.

5. The area of the AI engine is not publically available. However, in addition to raw MACs, it has a lot of additional control logic like load and store units, instruction fetch and decode units, debug and trace units, etc. The Tensor Slice, on the other hand, has a high ratio of compute to control logic. With the additional logic in the AI engine and given that it has 128 int8 MACs, it is likely much larger than 2× the area of a Tensor Slice. Therefore, the Tensor Slice has a higher throughput per unit area compared to an AI engine.

4.4.5 Replacing DSP Slices with Tensor Slices vs. Having Both

In Section 4.2.4 and in the experimental results for various benchmark circuits in Section 4.3, the proposed FPGA had both DSP Slices and Tensor Slices. Some area of the baseline FPGA is converted to Tensor Slices. So, the proposed FPGA has fewer LBs, DSPs and BRAMs than the baseline FPGA. However, in the DNN evaluation in Section 4.3.12, the proposed FPGA has only Tensor Slices. The area occupied by DSP Slices in the baseline FPGA is repurposed for Tensor Slices in the proposed FPGA. So, the proposed FPGA has the same number of LBs and BRAMs as the baseline FPGA. This is done

to make evaluation easy. This is similar to what Intel did with Stratix NX FPGAs [75].

There are tradeoffs between both these approaches. Having both DSP Slices and Tensor Slices on an FPGA chip means the FPGA is more generic. That is because the DSP Slice is tuned for non-DL applications. It provides multiplication and multiply-accumulate operations, in several precisions, along with many other features such as the ability to chain DSP Slices using direct interconnect. So, having DSP Slices on the FPGA allows mapping non-DL applications more efficiently. Having both DSP Slices and Tensor Slices, in addition to LBs and BRAMs, increases the heterogeneity of the FPGA fabric because there is one extra block type to handle, making CAD tools more complicated.

On the other hand, having only Tensor Slices makes the FPGA less generic and more DL specific. The Tensor Slice is tuned for DL applications. When only Tensor Slices are available and non-DL operations are required (in a non-DL application or in a DL application), the Tensor Slice's Individual PE mode can be used to map multiplication and multiply-accumulate operation. However, this mode is not as performant as the DSP Slice, as mentioned in Section 4.3.10. But having only Tensor Slices in the FPGA, along with LBs and BRAMs, does not increase the complexity of CAD tools (same heterogeneity as the baseline FPGA) and in fact, reduces the turnaround time for the compilation (synthesis, placement, routing, bitstream generation) for users, as shown in Section 4.3.7.

Common DL applications such as computer vision, natural language processing, healthcare, etc. are not performed in isolation, especially on the edge. There are other non-DL parts of the full application such as data pre-processing, result post-processing, filtering, dynamic control, network functions, etc. So, it is useful to have both DSP Slices and Tensor Slices on the FPGA.

Chapter 5

Adding Compute Capabilities to RAM Blocks in FPGAs

In this chapter, the second contribution of this dissertation is described: converting BRAMs on an FPGA to CoMeFa RAMs. A CoMeFa RAM enables computation within the RAM array, without transferring the data in or out of it. One-bit bit-serial configurable processing elements are added to the output of the sense amplifiers in the RAM block. This transforms the BRAM into a parallel SIMD (Single Instruction Multiple Data) computation unit. The availability of true dual-port mode in FPGA BRAMs [133][58] is exploited to read two operands in one cycle.

Computation in any precision can be easily performed in CoMeFa RAMs without any explicit precision-specific hardware because it uses bit-serial compute [32]. For performing a different operation or for using a different precision, a different instruction sequence needs to be generated and applied to the CoMeFa RAM. CoMeFa RAMs reduce the dependence on routing/interconnect and hence increase the routability of the FPGA. Data movement is reduced because the computation is done in the RAM itself, thereby saving power and reducing energy. Since the data is not moved in/out of the RAM

block, routing interface limitations do not restrict the available bandwidth. Instead, the internal physical geometry of the RAM, which is wider than the interface width, governs the effective bandwidth. The compute throughput and compute density ($GOPS/mm^2$) of the FPGA is increased significantly owing to the massive parallelism that is unlocked because of the existence of numerous RAM blocks on an FPGA. When not computing, CoMeFa RAMs can still function as normal BRAMs to store data.

This part of the dissertation resulted in a paper publication at the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) [9] and a research article at the ACM Transactions on Reconfigurable Technology and Systems [10]. The following contributions from the co-authors of these papers/articles are acknowledged:

- Aatman Borda: Implementation of the GEMV and GEMM benchmarks, and drawing some diagrams
- Tanmay Anand: Implementation of the FIR benchmark
- Atharva Bhamburkar: Experiment to compare BRAM+PE with CoMeFa
- Rishabh Sehgal: Experiment to verify working of CoMeFa using SPICE simulation
- Bagus Hanindhito: Drawing several diagrams

5.1 CoMeFa RAMs

In this section, the architecture and design of CoMeFa RAMs is described. The changes made to BRAMs to convert them to CoMeFa RAMs are explained. As in modern Intel FPGAs, a BRAM size of 20 Kilobits is considered, with support for single port, simple dual port, and true dual port modes, with the 512x40 being the shallowest and widest configuration. This BRAM has a physical geometry of 128 rows x 160 columns with a column multiplexing factor of 4 [79] [114]. Even though BRAM sizes, geometries, and modes from Intel FPGAs are used, the architecture described is not specific to a vendor. BRAMs from other vendors, such as Xilinx, may have slightly different sizes and geometries, but the architecture will work as long as the BRAMs support true dual port modes.

Some Intel FPGAs, such as the Stratix 10 family of FPGAs, implement the true dual port in Block RAMs by time-multiplexing the peripheral circuitry (sense amplifiers and write drivers). This is inferred by looking at the clock frequencies specified in the datasheet [56] - while the single port and simple dual port modes work at 1000 MHz, the true dual port mode works at 600 MHz. Implementing CoMeFa RAMs using such a Block RAM is feasible, but the frequency of operation of the CoMeFa RAM will be lower than the values evaluated later in Section 5.2.4. This will reduce the speedup obtained from CoMeFa RAMs. This architecture is not evaluated in this dissertation.

5.1.1 High Level Operation

At a high level, converting BRAMs into CoMeFa RAMs requires adding processing elements (PEs) to the sense amplifiers inside the BRAM block, as shown in Figure 5.1. The architecture of a PE can vary depending on the type of computations being targeted. The PEs are fed operands by reading multiple wordlines. They perform the required computation, and the result is written back into another wordline. Note that each PE requires two bits (one of each operand) in 1 cycle. All computation is done in a bit-wise manner, using transposed data layout. Figure 5.1 shows how operands are stored, read, computed on and the result stored back. Consider an example of bit-wise `AND`ing of the elements of two arrays (array length = 160 and element width = 4 bits). Each element is stored in a column, 1 bit in 1 row. This is called transposed layout. Elements of array 1 are stored in rows i, $i+1$, $i+2$, and $i+3$. Elements of array 2 are stored in rows j, $j+1$, $j+2$, and $j+3$. A total of 8 rows and 160 columns are required to store both arrays. In one cycle, rows i and j are read, each PE computes the `AND` of two bits, and the result is stored in row k. This process is repeated 4 times with increasing row addresses, and the final result is available, after 4 cycles, in rows k, $k+1$, $k+2$, and $k+3$. Note that 160 operations are done in parallel in each cycle.

5.1.2 Implementation options and changes to the BRAM

To achieve the high-level operation described above, different aspects of the BRAM need to be modified. For each aspect, there are multiple design

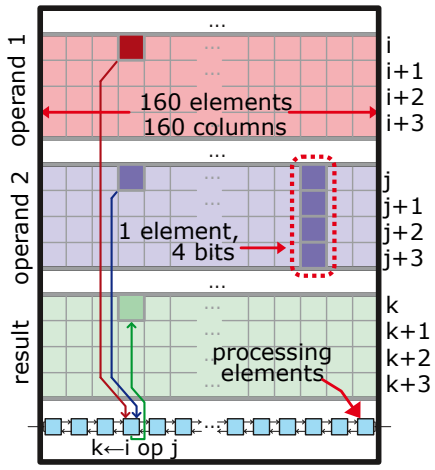


Figure 5.1: High level operation of CoMeFa RAM shown for 4-bit operands and a 4-bit result

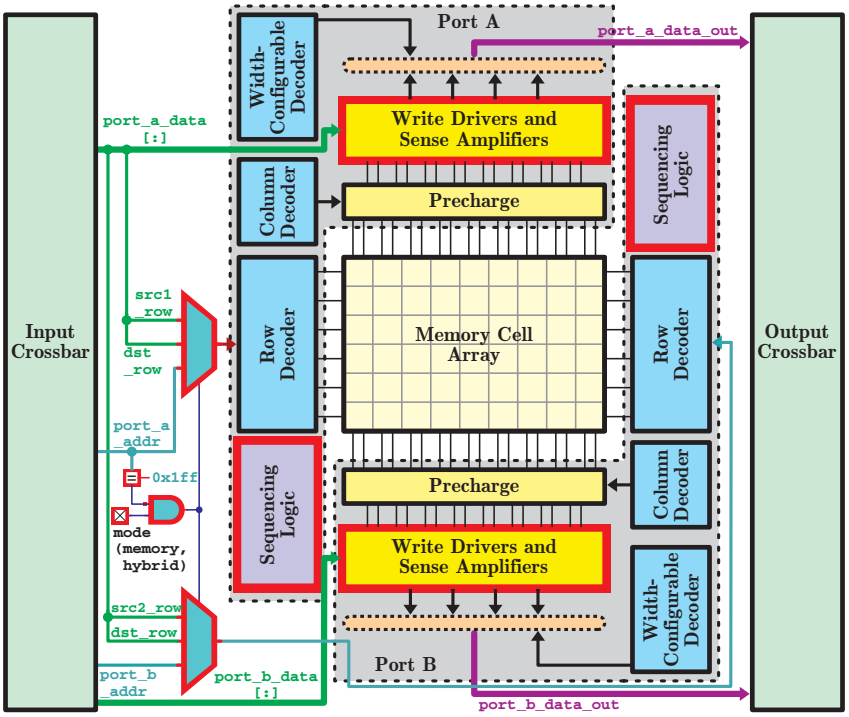


Figure 5.2: Top-level logical diagram of an FPGA BRAM with added/modified blocks for CoMeFa RAM highlighted in red.

Table 5.1: Implementation Options (Option used in this dissertation is in **bold**)

Objective	Options
Processing paradigm	• Bit-parallel • **Bit-serial**
Obtaining two operands	• Activating one wordline and storing bits • Storing operands in separate banks • Activating two wordlines together • **Using dual-ported memory**
Number of PEs and SAs	• **# PEs = # SAs = Number of bitlines** • **# PEs = # SAs = Number of datalines** • Or something in between
Distinguishing between data and instructions	• **Write to a special address** • Add a new signal on the interface
Transposing the data	• **In soft logic** • In DRAM controller • Use RAM with transposable cells
Programming the CoMeFa RAM	• **Workload-specific state machine** • **Stored program**

options. Table 5.1 lists the implementation options considered. The following sections explain each aspect and the design decisions taken.

The goal is to minimize the number of changes done to the BRAM to make the CoMeFa RAM design easily adoptable. Figure 5.2 shows a top-level diagram of an FPGA BRAM, with blocks modified/added for CoMeFa shown with a red outline. The sense amplifiers and write drivers are modified to add and connect the PEs. Sequencing logic that sequences the events of the read/write operations (wordline activation, precharge, sense amp enable, etc.) in the memory is modified. This is done to support reading and writing in one cycle. Some additional logic (comparator, `mode` configuration bit, multiplexers in front of row decoders) is also added. The memory array itself stays unmodified. The following sections explain each change in detail.

5.1.3 Processing Paradigm

There are two paradigms that can be used to convert a BRAM into CoMeFa RAM: Bit-Parallel and Bit-Serial. Bit-Parallel computing is the conventional paradigm in which multiple bits of one data element are processed every cycle. As an example, a conventional bit-parallel processor will take 128 steps to perform an element-wise sum of two arrays with 128 16-bit elements, using 16-bit PEs (adders). Bit-parallel PEs could be added in the RAM [42], for example, 16-bit fixed-point adders or floating-point multipliers. However, this means that the precisions supported by the PE have to be pre-determined, thereby reducing the flexibility of the block. Additionally, using bit-parallel PEs means restricting the location of data to be aligned to certain bitlines. Bit-growth during addition and multiplication operations can cause additional challenges. This paradigm is low in utility because it will not be very different from directly connecting a BRAM and a DSP slice.

Bit serial computing, on the other hand, is commonly used for digital signal processing and has been used on FPGAs as well [74] [73]. The main idea is to process one bit of multiple data elements every cycle. For the example above, a bit-serial processor with 128 processing elements would complete the operation of adding the two arrays in 16 steps as it processes the arrays *bit-by-bit* instead of *element-by-element*. Adding bit-serial PEs in the RAM makes the block a more generic computing unit. The PEs are agnostic to precision, which is useful for evolving applications like DL. The data has to be laid out in a transposed manner (bits of an operand located in a bitline, instead of

a wordline), to feed an operand into the PE one bit at a time. Adding bit-serial PEs to a BRAM converts the BRAM into a SIMD engine with a high vectorization width - up to 160 (in the case of Intel FPGA BRAMs that are considered) when one PE is added for each bitline. The main disadvantage of the bit-serial is that each operation takes many cycles, implying higher latency. However, this latency can be hidden/overlapped with other operations in data-parallel applications like DL. Bit-serial PEs are added in CoMeFa RAMs.

5.1.4 Obtaining two operands

To perform computation, each bit-serial PE needs one bit from each operand. There are multiple ways to achieve this. The first method, based on Computational RAM [35], involves adding flip-flops in the PE (Fig 2.12a). The row (wordline) containing the first operand's bits is read and the bits are stored in the flip-flops in the PEs. The row containing the second operand's bits is read in the next cycle and the computation is then performed. The results are stored back in another row in the third cycle. This increases the area of the PE, and also leads to a multi-cycle operation, reducing the speedup that can be obtained for applications.

In the second method, data can be stored such that each operand exists in a different bank of the RAM. The two banks can be accessed simultaneously. This requires the RAM to be implemented using two banks and also places a restriction on the data layout - that the two operands can not be in the same bank.

The third method is based on Logic-In-Memory [64] (Fig 2.12b). In this method, two wordlines containing bits of the two operands are activated at the same time. This needs changing the memory array, has robustness issues, and as mentioned in Section 2.4, is not very practical on a large scale.

The fourth method proposed in this dissertation uses dual-ported RAMs. Two bits, one from each operand, are read by the two ports' sense amplifiers and fed to the two operands to the PE. This costs additional area for the second port, but FPGA BRAMs are already dual-ported, so this does not add any additional area in the case of FPGAs. Although in the logical diagram of Figure 5.2, the peripheral circuitry of the two ports of the RAM (decoders, write drivers, sense amplifiers, etc.) are shown in diagrammatically opposite parts of the figure, in a typical physical layout of a RAM block, they are adjacent to each other. This ensures the practicality of adding a set of PEs fed by both sets of sense amps.

5.1.5 Modes, Stages and Phases

As shown in Figure 5.2, a new configuration SRAM cell is added which decides the mode of operation of a CoMeFa RAM block. A CoMeFa RAM can operate in two modes:

- **Memory Mode**: In this mode, CoMeFa RAM behaves as a conventional BRAM with no change in functionality. In this mode, the FPGA programmer can flexibly configure the number of ports and the width/depth of the BRAM.

- **Hybrid Mode**: If this mode is enabled at configuration time, the CoMeFa RAM can be used for computation as well as storage. In this mode, the RAM is automatically configured to its maximum width (512x40) to maximize the read/write throughput for populating the memory array with input data and reading the results.

Operations on CoMeFa RAMs typically happen in 3 stages:

- **Data loading stage**. Input data is stored in transposed format into the memory array in this stage.
- **Compute stage**. In this stage, the CoMeFa RAM is instructed to read source operand rows, perform computation in the processing elements and write the results to a destination row.
- **Data unloading stage**. Results can be read out in this stage by reading them from any address in the memory array.

A clock cycle during computation has 3 phases. In the first phase, two rows containing operand bits are read by activating the corresponding word lines. In the second phase, the logic gates in the PE compute the result. In the third phase, the result is stored back by activating a wordline. This leads to a longer clock period, compared to typical BRAM.

5.1.6 Number of processing elements and sense amplifiers

BRAMs typically employ column multiplexing [79] [114] for improving the detection and correction of transient errors in memory cells, and also to reduce the number of signals or wires to the programmable routing in FPGAs.

The memory array layout is roughly square (the number of bitcells in the x-direction is similar to the number of bitcells in the y-direction). When reading, column multiplexers select a smaller number of cells from those along a word-line, based on the address input. When no column multiplexing is present, the number of sense amplifiers in the RAM is equal to the number of bitlines. When column multiplexing is present, the number of sense amplifiers in the RAM is equal to the number of data lines (i.e. number of data signals on the RAM interface). When adding compute-in-memory capabilities into RAMs, the number of PEs and sense amplifiers to provide in the RAM is an architectural design decision involving area-delay tradeoffs. For CoMeFa RAMs, two architectures at the ends of the area-delay design space, are explored, as discussed below.

CoMeFa-D: In this architecture, additional sense amplifiers and write drivers are added to enable reading and writing a row in all columns (bitline pairs) together. A processing element (PE) is added below each column. This is similar to the architecture used in [32], [124] and [3]. During physical design/implementation, PEs (and sense amplifiers and write drivers) should be laid out so that they pitch-match with the SRAM cells for a bitline pair (BL and BLB). This implies longer/skinnier sense amplifiers than those that pitch-match with multiple bitline pairs behind a column multiplexer. This needs careful physical design and can be challenging. There are 160 sense amplifiers and write drivers per port and 160 PEs. This provides a parallelism of 160 operations done in 1 clock cycle (slightly longer than the baseline BRAM's clock

period) at the cost of high area overhead. This architecture is more practical than CCB [124] because multiple wordlines are not activated simultaneously on a port and voltage reduction is not required for robustness.

CoMeFa-A: In this architecture, the number of sense amplifiers and write drivers stays the same as the baseline. A PE (different from the one in CoMeFa-D) is added below each multiplexed column (i.e. one PE for each data line). The physical design of the sense amplifiers and write drivers used in this architecture does not involve any additional challenges than normal SRAM design because they have to be pitch-matched with multiple bitline pairs behind a column multiplexer. An optimization technique called sense amp cycling [109] is employed to sequentially sense multiplexed column bits in an extended clock cycle. There are 40 sense amplifiers and write drivers per port, and 40 PEs in the RAM. This provides a parallelism of 160 operations done in 1 extended clock cycle, thereby trading off delay for area. This extended clock cycle is not a major concern though, since critical paths in most FPGA designs include routing and LBs, and very rarely include BRAMs which can run at much higher frequencies. This architecture has the highest practicality among CCB [124] and CoMeFa variations because it retains column multiplexing.

5.1.7 PE Architecture

The next aspect of converting BRAMs to CoMeFa RAMs is to identify the architecture of the PE. The PE can be a simple logic gate, in which case

the CoMeFa RAM would be capable of only performing logical operations, and hence, would not be very useful for DL applications. Instead, using a bit-serial adder (2 inputs, 1-bit full-adder, 1 carry flip-flop, 1 output) as the PE enables arithmetic operations like addition and multiplication (by repeated addition). This is the core of the PE in [32] and [124]. Additional logic is provided for predication to enable cases where operations in some columns (or bitlines) need to be masked. The PE architecture used in CoMeFa RAM extends from this and adds additional features like configurability and BRAM-to-BRAM connections.

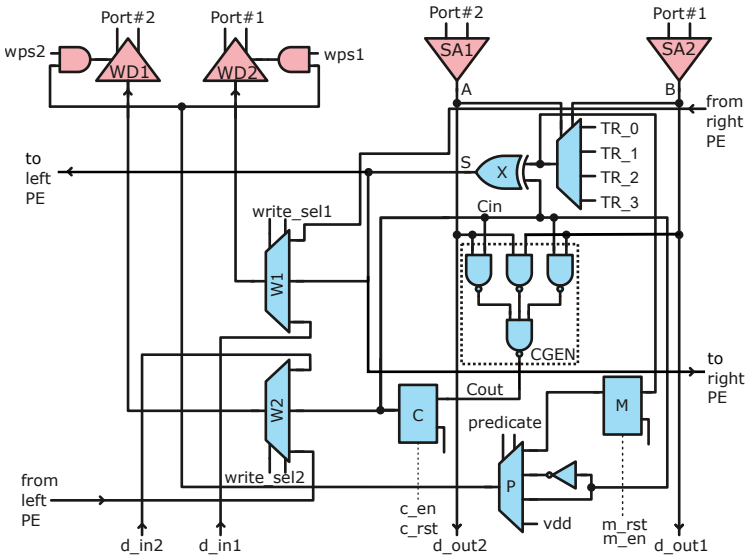
Figure 5.3a shows the structure of PE added to each column of the memory in CoMeFa-D. On the read path, **A** and **B** are the bits of the two operands read from the memory at sense amplifiers **SA1** and **SA2** of the two ports. Multiplexer **TR** evaluates a logical function of **A** and **B**, depending on the inputs **TR0, TR1, TR2, TR3** (truth table). If a 2-bit addition is required, the truth table bits will correspond to that of an XOR gate. The TR mux is basically like a 2-input dynamic LUT that can be configured every cycle. The output of **TR** goes through another XOR gate (**X**) to generate the addition of the input bits (**S**), including the previous cycle's carry. Gates to generate the carry (**CGEN**) are also present. The **carry** is stored in the carry latch (**C**) and can be used in the following cycle's computation. If an addition operation is not required, the carry latch is reset with **C_RST**=1, which enables **X** to pass the output of **TR** transparently to the **S** wire. **C_EN**=0 disables the latch so it keeps the old value. The read outputs **A** and **B** are

also sent to **d_out1** and **d_out2**, which is the normal read path.

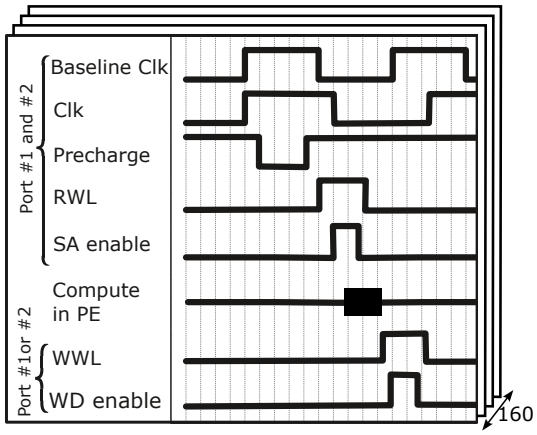
On the write path, 3-input multiplexers **W1** and **W2** are added before the write drivers of the two ports. These multiplexers determine the sources for the write operation. **W1** can select between the **S**, the input data port **d_in1** (normal write operation) and the value read from the right neighboring PE (used during left shift operation). **W2** can select between the **carry**, the input data port **d_in2** (normal write operation) and the value read from the left neighboring PE (used during the right shift operation).

Figure 5.3b shows a waveform view of the sequence of operations in 1 clock cycle for CoMeFa-D. After the bitlines are precharged, the read operation is performed by activating the read word line and asserting sense amplifier enable. The PEs compute on the values read by the sense amplifier. The results are written by activating the write wordline and asserting write driver enable.

The output of multiplexer **TR** is also stored in a special latch called **M** and is called **mask**. Predication logic allows enabling/disabling the write drivers (**WD1** and **WD2**). For this, a multiplexer (**P**) is added to select the signal that will enable/disable the write drivers. The **mask**, **carry**, **not-carry** and **VDD (logic 1; default)** can be selected. This helps CoMeFa RAMs mask writing the results based on various conditions, like the value of the **mask** or the **carry** bit, to support multiplications and floating point operations. The **wps1/2** signals decide which port's write path is activated for a given cycle.

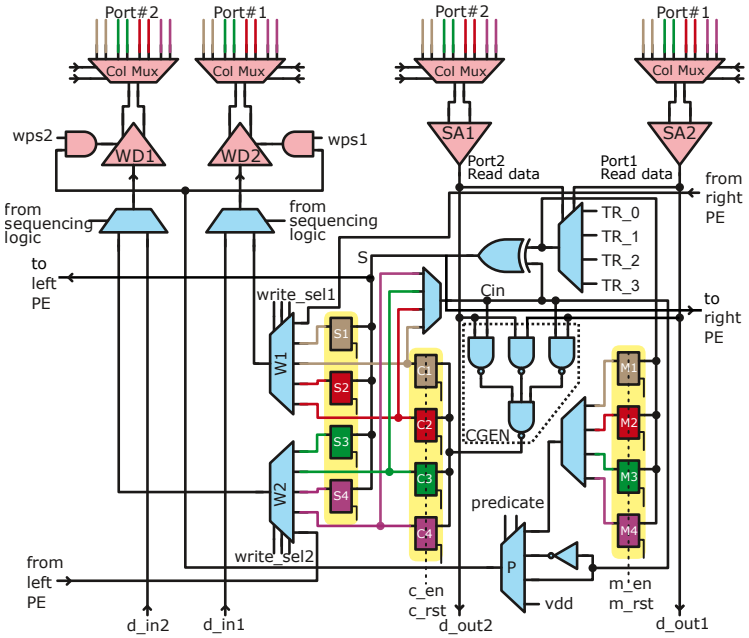


(a) Architecture

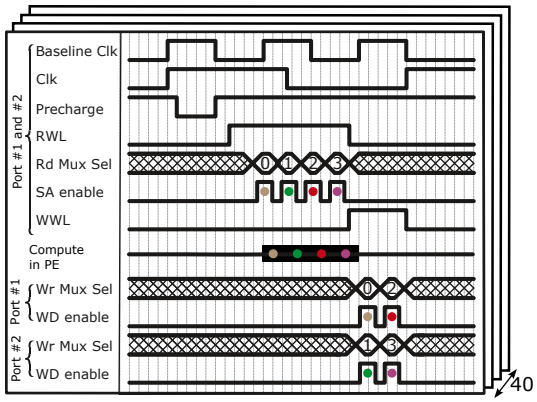


(b) Sequence of operations in one clock cycle

Figure 5.3: Processing element for CoMeFa-D (RWL=Read Word Line, SA = Sense Amplifier, WWL=Write Word Line, WD=Write Driver, Rd=Read, Wr=Write)



(a) Architecture



(b) Sequence of operations in one clock cycle

Figure 5.4: Processing element for CoMeFa-A (RWL=Read Word Line, SA = Sense Amplifier, WWL=Write Word Line, WD=Write Driver, Rd=Read, Wr=Write)

Figure 5.4a shows the structure of PE added to each multiplexed column of the memory in CoMeFa-A. All the labels have the same meaning as the PE described above. The number of **C** and **M** latches changes to 4, and there are 4 additional latches for **S**. On each port, 4 column-multiplexed bits are read and two results are written back in an extended clock cycle. In the read phase of the cycle, the brown bitline pairs from each port are sensed first. The resulting **S** bit is stored in latch **S1**, carry bit **C** is stored in the latch **C1**, and mask bit **M** is stored in latch **M1**. This is repeated for red, green, and purple bitline pairs successively. All **Sn**, **Cn**, and **Mn** latches get updated in this process. Then, in the write phase of the cycle, results for the brown and red bitlines are written using the write drivers of the two ports, followed by the green and purple ones. This is shown in Figure 5.4b. Clocks in the PE are driven by signals derived from the sense amplifier enable pulses. The paths in the PE do not add any additional delay to the extended clock from sense amp cycling.

5.1.8 Instructions

An instruction is defined as a bitvector whose bits tell the CoMeFa RAM and the PE what to do in each cycle. The CoMeFa RAM instruction is 40 bits and the format is shown in Figure 5.5. The field names in the instruction are self-explanatory. They directly drive the corresponding signals in the PE (e.g. `predicate` bits are applied to the select lines of the multiplexer **P**). The `src1_row`, `src2_row` and `dst_row` bits are used for activating the first operand

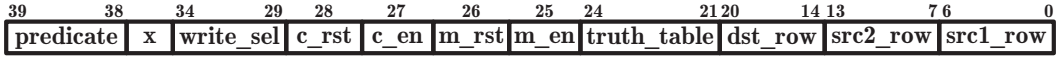


Figure 5.5: Instruction format for CoMeFa RAMs

row on Port A, the second operand row on Port B, and the row at which results will be stored, respectively. These addresses are fed to the appropriate row decoders at the right time in the clock cycle by the sequencing logic in the CoMeFa RAM using the multiplexers shown in Figure 5.2. Instructions are generated using instruction generation logic and fed to the CoMeFa RAMs. Section 5.1.14 will discuss the various ways of achieving this.

5.1.9 Distinguishing between data and instructions

As mentioned in Section 5.1.5, in Hybrid mode, the CoMeFa RAM can be used for computation as well as storage. So, in this mode, a way to distinguish between compute operations (sending instructions to the RAM) and storage operations is required. Two options are considered for this. In the first option, a special address (0x1FF) is reserved to signal sending instructions to the RAM, similar to [124]. Data written to this address is treated as an instruction. Accessing other addresses is done normally; used for storing operands and reading results. A comparator is added to Port A's address signal to check for this address (see Figure 5.2). One limitation of this method is that the special address (0x1FF) cannot hold data anymore. This can be a problem especially when an application needs to store data on all addresses of the RAM. As a workaround, data can be written to another address first and then copied or moved to the special address internally in the BRAM using a

compute instruction.

The second option is to use a dedicated signal on the RAM interface that when asserted indicates that the data written into the RAM be treated as an instruction. This does not need adding an extra signal on the RAM interface because existing signals can be reused. In Hybrid mode, the BRAM is configured into its widest mode (512x40). So, 9 address bits are required. In other modes (like 1024x20 or 2048x10), there are more address bits (10 in 1024x20 and 11 in 2048x10), and those address bits are unused in the widest mode. One of these bits can be reused to denote that the data bus contains instructions.

Both methods do not have any impact on the performance of the CoMeFa RAM. Results from the evaluation are independent of which method is used.

5.1.10 RAM-to-RAM chaining

CoMeFa RAMs provide the capability of performing left-shift and right-shift operations efficiently. Shifts are single operand operations. For a left (right) shift operation, the source operand row is read into the PEs, each PE's **W1** (**W2**) mux is configured to select the bit read from the right (left) neighboring PE, and that bit is written into the destination row. For CoMeFa-A, shifting values from a bitline pair to another bitline pair within the same column multiplexer is also supported, by decoding the `write_sel` bits of the instructions and setting the select lines of **W1** and **W2** muxes appropriately.

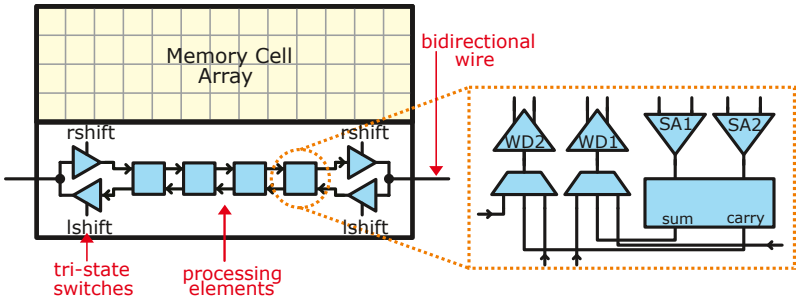


Figure 5.6: CoMeFa RAM supports shifting within a block and across blocks using chaining

Direct links connecting top and bottom neighboring CoMeFa RAMs are provided to allow for shifting data between the corner PEs in each CoMeFa RAM. These connections can provide a much easier way to perform inter-CoMeFa RAM communication and obtain even more parallelism. Figure 5.6 shows the pins on a CoMeFa RAM required to provide these direct connections between CoMeFa RAMs, along with the details of the shift operation support inside each PE. These connections are similar to carry chains in Logic Blocks and cascade chains in DSP Slices in modern FPGAs. Xilinx FPGAs have direct BRAM-to-BRAM interconnections as well. If unidirectional wiring is used, four additional pins would be required on the CoMeFa RAM to allow for shifting in both directions. To minimize this overhead, bidirectional wires controlled by tri-state switches are provided, because at one time, shifting in only one direction is required. The tri-state switches are controlled by a signal decoded from the `write_sel*` and `wps*` bits, because they govern the shift direction.

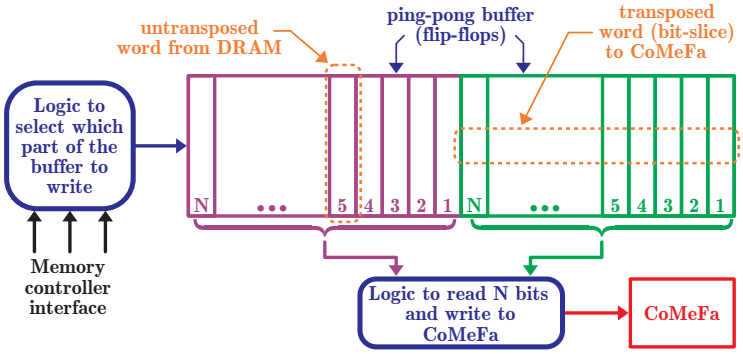


Figure 5.7: Swizzle logic to load non-transposed data from DRAM directly into CoMeFa RAM in transposed layout (N=40)

5.1.11 Transposing the data

As mentioned in Section 5.1.1, data has to be stored in a transposed layout in CoMeFa RAMs for computation. There are multiple methods that can be used for transposing data. The first method is to modify the memory array to use transposable bitcells, similar to [121]. This approach is also referred to as Transpose Memory Units (TMU) in [32]. This requires extensive modification to the RAM and also increases the area significantly, so this approach is not pursued.

Another method is to design logic to transpose data and implement it in soft logic. This is referred to as a **swizzle module** (or **swizzle logic**) that can read data from DRAM, transpose it and write it to a CoMeFa RAM on-the-fly. The architecture of the swizzle logic, shown in Figure 5.7, employs a ping-pong buffer FIFO. Untransposed data read from DRAM is written in order into the ping part of the FIFO (depth = 40 elements). When the ping part is full, a transposed word (a bit slice from 40 elements) can be read and

written into consecutive bitlines on the same wordline in a CoMeFa RAM, and new data from DRAM is written into the pong part. After the pong part is full, transposed data is read from the pong part and written into a CoMeFa RAM, and new data from DRAM is now written into the ping part again. This process continues until the required data has been populated into CoMeFa RAMs. Similarly, transposed data can be read from CoMeFa RAMs and stored into DRAM in untransposed form using swizzle logic. In other words, the swizzle logic is a set of registers which are written in columns (all bits of a column are written with a bit-parallel word) and read in rows (one bit each of many words is read along a row). This method in used in the evaluation.

The swizzle module is mapped to LBs. So, for an application that is already bound by LB usage, this can be a concern. There are multiple ways to reduce dependence on swizzle modules by avoiding the need for transpose, such as OOOR operations (Section 5.1.13), popcount based addition [124], and storing pre-transposed values in CoMeFa RAMs for static data like weights of a neural network during inference.

A third method is based on the realization that transposing data is only needed when reading/writing from/to the DRAM. Modern FPGAs have hardened DRAM controllers integrated into them. So, transpose logic (like the swizzle module) could be hardened into a DRAM controller. However, the current baseline FPGA does not have a hard memory controller.

5.1.12 Variable precision support

Hardware in CoMeFa RAM PEs is not specific to any numerical precision. A different sequence of instructions is all that is required to compute in a different precision. The sequences for fixed-point addition, multiplication, and in-RAM reduction are the same as [32]. Addition for n-bit operands takes $n + 1$ cycles. Multiplication of n-bit operands takes $n^2 + 3n - 2$ cycles. CoMeFa RAMs can natively support floating point precisions as well, as opposed to CCB [124]. Floating point algorithms for addition and multiplication are adopted from FloatPIM [50]. The CoMeFa RAM PE can perform all the steps in the sequences because: (1) **carry** and **not-carry** are used in the predication logic, (2) **mask** is populated from the output of the programmable multiplexer **TR** instead of just **A** or **B**, and (3) operations like XOR can be performed easily using **TR** and the `truth_table` fields in the instruction. The approximate number of cycles consumed for floating point multiplication and addition are $M^2+7M+3E+5$ and $2ME+9M+7E+12$, where M = number of mantissa bits and E = number of exponent bits.

5.1.13 One Operand Outside RAM (OOOR) operations

In Section 5.1.8, two operands are stored inside the RAM. However, in many cases, an optimization can be applied - one of the operands can be outside the RAM. For example, multiplying an array of numbers (stored in the RAM) with a scalar operand (outside the RAM). These are called OOOR operations. This method saves space inside the RAM. Without OOOR, in the

multiplication example, the scalar operand would need to be replicated in each column. This method allows easy inspection of outside operand's bits, thereby enabling efficient algorithms. For example, in the normal shift-and-add based multiplication explained in [32], if a bit in the scalar operand is 0, cycles are still consumed, which can be avoided by using OOOR. In the average case, half of the bits will be 0 and therefore, the number of cycles can be reduced by 50%. Efficient algorithms like booth multiplication can also be deployed. A simple way to perform OOOR operations is to have a row of 1s and a row of 0s in the RAM and use that as a proxy for bits in the operand outside the RAM. Alternatively, appropriate truth table (TR) bits can be sent to the PE in CoMeFa RAMs to achieve the same goal. Overall, OOOR operations make the PEs more powerful by expressing 2 (or 3) operand operations as 1 (or 2) operand operations.

OOOR is applied to design an efficient algorithm for dot-product operations where one of the vector's elements is common to all columns. This is useful in many applications like matrix-vector multiplication and FIR filter. Consider the case where a weight vector $[X, Y, Z, W]$ needs to be multiplied with multiple vectors $[A, B, C, D], [E, F, G, H], ...$ and each vector is stored in a different column of the RAM. The weight vector does not need to be stored in the RAM, but can be outside the RAM and inspected in the instruction generation logic to generate appropriate instructions. To simplify, $AX + BY$ is calculated as the building block operation in one column. Fig 5.8 pictorially shows the evaluation of partial sums PV and PW. In a naive algorithm, PV

will be calculated first and then PW. So, now both PV and PW are in the same column in the RAM. They will be added to get the result. This algorithm is shown in Algorithm 1, and can be done using OOOR with X and Y being outside the RAM, and will provide a speedup of 2x on average assuming half the bits are zeroes.

In the proposed intelligent algorithm (shown in Algorithm 2), $A + B$ is first calculated (call it *temp*). So, now A, B, and $A + B$ (temp) are present in the column. When X and Y are outside the RAM, bits $X[0]$ and $Y[0]$ can be inspected together. If they are 11, $A + B$ (temp) is added to the result. If they are 10, X is added to the result. If they are 01, Y is added to the result. If they are 00, nothing is done. This is successively done for all bit locations of X and Y. When adding to the partial result each time, the correct rows are added to effectively do the shifts required during a normal multiplication. This algorithm provides a speedup of 2x compared to Algorithm 1, and up to 4x compared to the naive multiply-then-add algorithm.

For OOOR operations, the data outside the RAM does not need to be transposed, thereby saving some swizzle logic. There are some disadvantages to using OOOR operations as well. The instruction generation logic becomes more complex. A lesser reduction in energy consumption should be expected, because of additional control logic outside the RAM and because of higher dependency on programmable routing. Since the instruction generation logic takes different paths based on the data, the opportunities of sharing it across many CoMeFa RAMs may drop depending on the application.

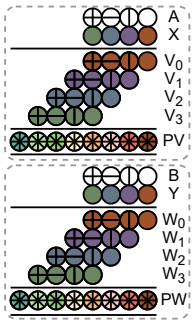


Figure 5.8: Steps to perform A×X and B×Y

Algorithm 1 Naive dot product

Input: A, X, B, Y
Output: Z
  $PV = 0$
  $PW = 0$
  for $i = 0$ to *precision* **do**
    Calculate V_i
    $PW+ = (V_i << i)$
  end for
  for $i = 0$ to *precision* **do**
    Calculate W_i
    $PW+ = (W_i << i)$
  end for
  $Z = PV + PW$

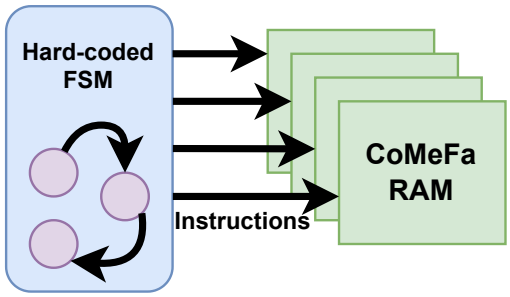
Algorithm 2 Intelligent dot product

Input: A, X, B, Y
Output: Z
  Calculate $temp = A + B$
  for $i = 0$ to *precision* **do**
    if $X_i, Y_i == 2'b11$ **then**
      $Z+ = (temp << i)$
    else if $X_i, Y_i == 2'b10$ **then**
      $Z+ = (A << i)$
    else if $X_i, Y_i == 2'b10$ **then**
      $Z+ = (B << i)$
    else *No Change*
    end if
  end for

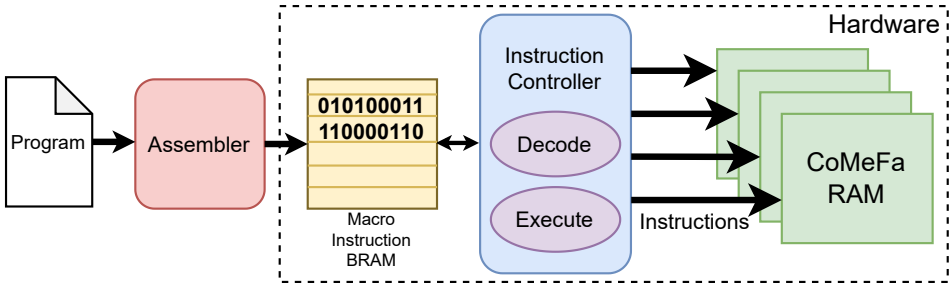
5.1.14 Programming CoMeFa RAMs

Programming a CoMeFa RAM means sending it a sequence of instructions to perform a given operation. Two methods for programming CoMeFa RAMs are considered. In both methods, multiple CoMeFa RAMs can share instruction generation logic to amortize its cost. However, doing so can increase the fanout and reduce frequency.

First, a finite state machine (FSM) implemented in soft logic is used to generate instructions, similar to [124]. This is shown in Fig 5.9a. This method leads to an efficient implementation because the FSM can be customized to (or hardcoded for) specific requirements of an application. However, this method is tedious because designing an FSM to generate instructions for bit-serial operations is not easy. This process could be automated by using High-Level Synthesis, given a high-level language description of the sequence of operations required. This is not done in this dissertation.



(a) An FSM sending instructions to CoMeFa RAMs



(b) An instruction controller decoding macro-instructions and sending instructions to CoMeFa RAMs

Figure 5.9: Programming CoMeFa RAMs

To improve programmer productivity, a stored program method is considered, inspired by Compute RAMs [14]. This is shown in Fig 5.9b. For this method, macro-instructions are defined for the various operations supported by the CoMeFa RAMs. Table 5.2 shows the list of supported macro-instructions (or opcodes). The `ram[x]` notation refers to an operand stored in the CoMeFa RAM at row `x`. The `out[x]` notation refers to an operand outside the RAM. The controller currently supports selecting an operand from 9 values outside the CoMeFa RAM using multiplexing logic. A macro-instruction ending in `_ooor` means that at least one of the operands is outside the RAM. Each operation is done for the precision specified in the macro-instruction (using a field ending in `_prec`). The address specified in the instruction refers to the row number of the least-significant-bit of the operand. The `dot_prod` operations performs $a * x + b * y$. Since there are only 40 bits in a macro-instruction, and more than 40 bits are needed to express all operands of this instruction, an assumption is made that the x is laid out right after a and y is laid out right after b. This is the reason for having `ram[src3+prec]` and `ram[src1+prec]` in the instruction description.

An assembler (written in Python) converts a program written using these macro-instructions into a binary format. This binary data is loaded into a BRAM (either at configuration time or at run time). An instruction controller (implemented in soft logic) fetches macro-instructions from the BRAM, decodes them, and sends instructions to the CoMeFa RAMs.

Not all applications need to use all the macro-instructions supported by

CoMeFa RAMs. E.g, an application that performs elementwise operations may only need the `add` and `mul` macro-instructions. To support this, an instruction controller generator (written in Python) is designed. A user can generate an instruction controller that only supports the macro-instructions they need. This keeps the instruction controller lean and lowers the overhead.

The hardware for executing some macro-instructions such as `reduce` can be complex. Providing support for such macro-instructions in the controller will make it complicated. Instead, for such macro-instructions, the burden is moved to the assembler. The assembler converts these complex macro-instructions into a sequence of simple macro-instructions such as `add` and `shift`. So, the controller only supports simple macro-instructions and stays lean.

In some applications, the program can be very small (a few macro-instructions). In such cases, using a BRAM to store a few instructions is wasteful. A user can map the binary to distributed RAM in LBs instead of a BRAM. On the other hand, in some cases, the program can be very long and may exceed the number of instructions that can be stored in a BRAM. In the applications evaluated in this dissertation, this is never the case. But, in the future, to reduce the size of the program, a macro-instruction (`repeat`) that will implement hardware loops is planned. Support for floating-point operations in the controller is also planned.

To make adoption of CoMeFa RAMs even easier by users, a compiler could be developed that would translate a DNN application written in Python

Table 5.2: Macro-instructions supported by the assembler. The operator +: has a meaning similar to Verilog's index part-select operator. For example, data[24 +: 8] is the same as data[31:24]

Instruction	Operands	Semantics
add	dst, dst_prec, src2, src2_prec, src1, src1_prec	ram[dst +: dst_prec] ← ram[src2 +: src2_prec] + ram[src1 +: src1_prec]
add_ooor	dst, dst_prec, src2, src2_prec, src1, src1_prec	ram[dst +: dst_prec] ← out[src2 +: src2_prec] + ram[src1 +: src1_prec]
mul	dst, dst_prec, src2, src2_prec, src1, src1_prec	ram[dst +: dst_prec] ← ram[src2 +: src2_prec] × ram[src1 +: src1_prec]
mul_ooor	dst_prec, src2, src2_prec, src1, src1_prec	ram[dst +: dst_prec] ← out[src2 +: src2_prec] × ram[src1 +: src1_prec]
logical	dst, src2, src1, prec, op	ram[dst +: dst_prec] ← ram[src2 +: src2_prec] *op* ram[src1 +: src1_prec]
logical_ooor	dst, src2, src1, prec, op	ram[dst +: dst_prec] ← out[src2 +: src2_prec] *op* ram[src1 +: src1_prec]
shift	dst, src, dir, shamt, prec	ram[dst +: prec] ← ram[src +: prec] left or right shifted by *shamt*
dot_prod	dst, dst_prec, src3, src3_prec, src1, src1_prec	ram[dst +: dst_prec] ← ram[src3+src3_prec +: src3_prec] × ram[src3 +: src3_prec] + ram[src1+src1_prec +: src1_prec] × ram[src1 +: src1_prec]
dot_prod_ooor	dst, dst_prec, src4, src3, src2, src1, src_prec	ram[dst +: dst_prec] ← out[src4 +: src_prec] × ram[src3 +: src_prec] + out[src2 +: src_prec] × ram[src1 +: src_prec]
reduce	dst, src, levels, prec	internally reduce operands (each of precision `prec`) located across CoMeFa RAM `levels` times
unload	src, count	unload data from ram[src] to ram[src+count] from the ram
init (set/reset)	dst, pattern, count	ram[dst+count] to ram[dst] ← pattern
set_mask	src	mask register in PE ← ram[src]
nop	count	No operation for `count` cycles

or C into macro instructions. Such a compiler would identify the best parallelism distribution across CoMeFa RAM blocks, perform data allocation in memory rows, keep track of the lifetime of each data, and eventually generate macro instructions (like `add`, `mul`, etc.). The assembler would then convert these macro-instructions to the binary format, which will be then be decoded and executed by the instruction controller. The development of this compiler is left as future work.

5.2 Evaluation Methodology

5.2.1 Tools and Methods Used

The Verilog-to-Routing (VTR) tool flow [85] is used to evaluate and compare FPGA architectures. COFFE [136] is used to obtain the area and delay values for the various components of the FPGA, including CoMeFa RAMs, (to enter them in the FPGA architecture model for VTR experiments). SPICE simulations are performed using FreePDK45 [88] for a circuit containing one bitline pair, two wordline circuits, and two memory cells, with other transistor and wire loads modeled. This is done to get more confidence that the read+compute+write operation of CoMeFa RAMs works, and to validate the numbers obtained from COFFE.

A cycle-accurate behavioral model of CoMeFa RAM is developed to use in functional simulations. This model is written in System Verilog and has the exact same interface (input and output signals) as a CoMeFa RAM hard block in the FPGA architecture model for VTR experiments. Both the storage and compute modes are modeled, by accurately capturing the functionality of the PE and the RAM array. Synopsys VCS and Xilinx Vivado's integrated simulator are used for functional verification of all designs (e.g. designs for benchmarks) used during the evaluation.

An analytical model is used to estimate dynamic energy consumption. Transistor energy and wire energy are considered. Transistor is calculated based on the number of transistors in each block (obtained from the area consumed by the block from VTR). Wire energy numbers (fJ/mm) are taken

from [69], scaled to 22 nm technology node using [108], and multiplied with the total routing wirelength from VTR. An activity factor of 0.1 is used.

5.2.2 Baseline vs. Proposed Architectures

An Intel Arria 10-like FPGA architecture is used as the baseline with the same resources as Arria 10 GX900 [59] (Table 5.3). Arria 10 FPGAs [58] use a technology node (20 nm) similar to the setup used in this chapter (22 nm). Arria 10 GX900 has 96 transceiver channels that support up to 17.4 Gbps [60]. It is assumed that a 4-port full-width soft HMC (Hybrid Memory Cube) controller [51] is implemented on the FPGA to provide a DRAM bandwidth of 2048 bits/clock. Resources consumed by the controller are not used to map the applications to the FPGA. FPGAs with a higher BRAM:DSP ratio will see even more benefits by converting BRAMs to CoMeFa RAMs.

The VTR FPGA architecture used in [12] is used to make a baseline architecture model. COFFE simulations are run on an Arria-10 like DSP to identify its delay and area. Delay and areas of a 20 Kilobit BRAM are obtained from COFFE (by interpolating between 16K and 32K). These results are scaled based on the DSP and BRAM delays specified in [55]. The DSP slice works at 630 MHz in fixed-point mode and 550 MHz in floating-point mode. The BRAM works at 735 MHz in single-port, simple dual-port, and true dual-port modes. The proposed FPGA architecture models (CoMeFa-D and CoMeFa-A) differ from the baseline in having CoMeFa RAMs instead of normal BRAMs.

Table 5.3: Properties of the baseline FPGA architecture (Intel Arria 10 GX 900 like)

Resource	Count	Percentage Area
Logic Blocks	33962	66
DSP Slices	1518	18
Block RAMs	2423	15
DRAM bandwidth	2048 bits/clock	
Channel width	300	

Table 5.4: List of microbenchmarks used for evaluation (CB = Compute bound, OMB = On-chip memory-bandwidth bound, DBB = DRAM bandwidth bound)

Microbenchmark	Domain	Scenario Created	Storage	Precision
GEMV	DL	CB	DRAM	8-bit
GEMM	DL	CB	DRAM	8-bit
Conv2D	DL	CB	DRAM	8-bit
FIR Filter	Signal Processing	CB	DRAM	16-bit
Eltwise Mult	DL	DBB	DRAM	HFP8
Bitwise - Search	Databases	OMB	BRAM	16-bit
Bitwise - RAID	Data Center	OMB	BRAM	20-bit
ReLU	DL	OMB	BRAM	16-bit
Reduction	DL	OMB	BRAM	Multiple

5.2.3 Benchmarks

Verilog designs are created for several diverse applications to use as microbenchmarks (Table 5.4). These include Deep Learning (matrix-vector multiplication (GEMV), matrix-matrix multiplication (GEMM), 2D convolution (Conv2D), reduction, elementwise multiplication (Elt Mul), rectified linear unit activation (ReLU)), signal processing (FIR filter or 1D convolution) and bitwise applications (database search and RAID array data recovery). These applications are manually mapped to CoMeFa RAMs and instantiate

the CoMeFa RAM blocks in Verilog RTL. During functional verification, a simulation model of CoMeFa RAM is used. Different scenarios (compute bound, DRAM bandwidth bound and on-chip memory bound) are created in these applications. Additionally, the impact of adding CoMeFa RAMs on the performance of real-world Deep Neural Networks (DNNs) is evaluated. Three common DNN types are used: Fully Connected Networks (MLP), Recurrent Neural Networks (LSTM and GRU), Convolutional Neural Networks (Tiny Darknet and ResNet).

General Matrix-Vector Multiplication (GEMV) and General Matrix-Matrix Multiplication (GEMM): GEMV and GEMM are fundamental operations in DL applications. They are used in MLPs, LSTMs and many other DNNs. The GEMV workload has a weight matrix of size 2048x512 is multiplied with an input vector of size 512x1. The GEMM workload has a weight matrix of size 1536x512 is multiplied with an input matrix of size 512x32. These are sizes from actual layers in DeepBench benchmarks [87]. 8-bit integer precision with 27-bit accumulation is used. On the baseline FPGA, compute units are implemented using efficient chaining of DSPs. On the proposed FPGA, compute units based on CoMeFa RAMs are additionally deployed, because many RAM blocks are available after mapping the baseline design on the proposed FPGA. The efficient OOOR-based dot product algorithm, described in Section 5.1.13, is used. Partial sums are read out from the CoMeFa blocks and accumulated using a pipelined bit-serial tree [74]. No online data transpose is required - the weight matrix is transposed offline and

pinned into CoMeFa RAM blocks; the input is streamed and does not need to be transposed because it is outside the RAM. Since both DSP based and CoMeFa based compute units are used, a reduction in data movement is not expected.

Convolution: The convolution operation forms the backbone of Convolutional Neural Networks (CNNs). A convolution layer with the following parameters is considered - Input: Height = 72, Width = 72, Channels = 128; Filters: Height = 2, Width = 2, Number = 128; Output: Height = 71, Width = 71, Channels = 128. On the baseline FPGA, dot product units are designed using DSP slices to perform multiplications and additions along the channel dimension, and then the results from the 4 filter locations are added. The filters are stored in BRAMs and the inputs are streamed. A compute unit on the baseline FPGA is made of 64 DSPs and 8 BRAMs. On the proposed FPGAs, CoMeFa RAMs are additionally deployed. Filters are pre-transposed and stored in the CoMeFa RAMs. A compute unit formed by CoMeFa RAMs contains 128 CoMeFa RAMs, along with instruction generation logic. The columns of a CoMeFa RAM are used to store different filters (vectorization across the output channel dimension), whereas the RAMs in a unit are used for vectorization across the input channel dimension. OOOR operations are used to compute dot products. The input data is divided between the compute units formed by DSPs and those formed by CoMeFa RAMs.

Finite Impulse Response (FIR) Filter: FIR filters are a common Digital Signal Processing (DSP) application. An FIR filter with 128

taps is considered. Input operands are streamed onto the FPGA through the DRAM interface. The baseline FPGA uses an efficient implementation of FIR filter using systolic DSP chaining [6]. The proposed FPGA uses CoMeFa RAMs for computation along with DSP chains. Logic blocks are used for control logic. Operands are transposed on-the-fly and loaded into multiple CoMeFa RAMs in parallel. While some CoMeFa RAMs are computing, other CoMeFa RAMs are loaded in a pipelined manner to improve parallelism. When a CoMeFa RAM finishes computing, its results are unloaded and sent to DRAM, and the process starts again until all inputs are processed. This is called the Load-Compute-Unload (LCU) pipeline. In this application, the CoMeFa RAM-to-CoMeFa RAM chaining (Section 5.1.10) feature is used to share inputs between neighboring blocks.

Elementwise multiplication: Elementwise multiplications are commonly used in DL, for example, in normalization layers and Winograd-based convolution layers. An application involving elementwise multiplication of two arrays of 100K elements is considered. Floating point data with a precision of HFP8 [111] is used. The intent is to showcase that CoMeFa RAMs are adaptable to any custom precision. The operands are read from DRAM and the results are written to DRAM. This is a DRAM bandwidth bound application because of low arithmetic intensity. It is observed that the number of LBs used is significantly higher (25x) than in the baseline FPGA. This is because to saturate the DRAM bandwidth available on the chip, many swizzle logic instances are required. However, if the swizzle logic is hardened into a

DRAM controller, as discussed in Section 5.1.11, then this overhead is entirely removed.

Bitwise operations: Bitwise operations (like AND, OR, XOR, XNOR, etc) are commonly used in databases, encryption, DNA sequence alignment, etc. They are also used in Binary Neural Networks (BNNs). CoMeFa RAMs are very efficient at these massively parallel operations because of the presence of mux-based fully configurable PEs. The operands are assumed to be available in BRAMs in the right layout. The speedup seen in these applications is attributed to the effective increase in on-chip memory bandwidth because 160 bits can be operated upon in 1 cycle in a CoMeFa RAM, compared to only 40 bits from a BRAM in the baseline FPGA. Two applications are considered in this category.

Database search: In this application, records matching a key are searched. If a record matches the key, it is replaced with special marker data (like constant 0). Each operand is bitwise XOR'ed with the key. Bitwise OR reduction is performed on the result. And then a bitwise ANDing operation is performed to zero out the operands that match the key. BRAMs are used to store operands. Each row of a BRAM has 2 16-bit elements. On the proposed FPGA, elements are stored in 256 CoMeFa RAMs. 7 data elements are stored in each column and temporary results consume 16 rows in a CoMeFa RAM. The key is outside the RAM.

RAID data recovery: In RAID (Redundancy Array of Independent Disks) arrays, parity protection is used. If a drive in an array fails, the re-

maining data on other drives is combined with the parity data (using XOR) to reconstruct the missing data. These numerous parallel XOR operations with the parity data can be accelerated using an FPGA. Instead of storing operands in a transposed format (bits of one operand in multiple rows), an un-transposed data layout is used where bits of one operand are stored in one row and bits of the second operand are stored in another row. This works for logical operations like bitwise XOR where there is no dependency/communication between consecutive bits and avoids the overhead of transposing data. Performing an XOR operation between operands stored on two rows takes 1 cycle. 256 RAMs are used.

ReLU: Rectified Linear Unit (ReLU) is the most common activation function used in DNNs. Activations usually follow a GEMM or GEMV or CONV operation. The operation involves zero'ing out any negative input, but any positive input stays unchanged. The input data is available in a BRAM (computed by a prior kernel, for example). The precision is 16-bit. In CoMeFa RAMs, the inverted most significant bit (sign bit) of each input is copied into the mask latches in the PEs. The value 0 is written to each row containing the input elements. In some columns, the operation is masked (because the sign bit is 0) implying the values stay unchanged. But in other columns, the values are zero'ed out. In the baseline FPGA, values are read from the RAM, their most significant bit is inspected, the output is generated using simple multiplexing logic and written back into the RAM.

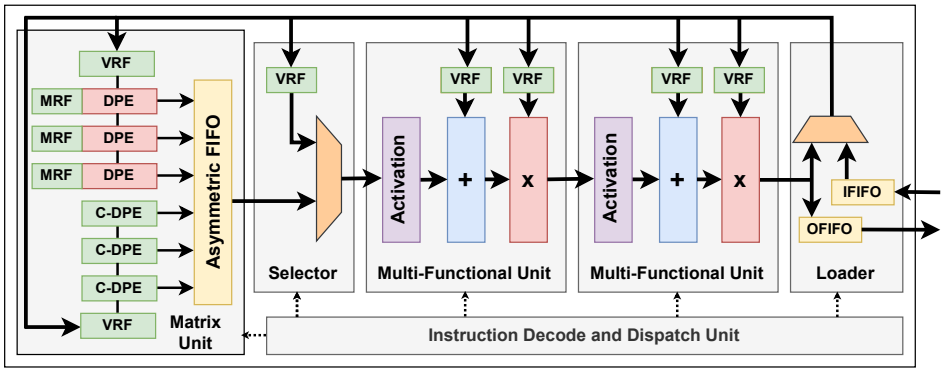
Reduction: Reduction (or accumulation) is heavily used in DL and

DSP applications. This application is designed to create a scenario of an on-chip memory bandwidth limited application. Data is available in transposed format (computed in RAM by a prior kernel, for example). The precision is varied from 4-bit to 20-bit (accumulator size = 32-bit). In the baseline, operands stored in BRAMs are read and successively accumulated using a pipelined adder tree (in LBs). On the proposed FPGA, CoMeFa RAMs store the operands. The reduction algorithm from [32] is used to reduce the elements to 40 partial sums (1 partial sum in each multiplexed column of the RAM). These intermediate results from multiple CoMeFa RAMs are then read out and accumulated using a popcount-based adder [124] to obtain the result. A significantly smaller number of LBs (∼2x-3.5x) is required on the proposed FPGA.

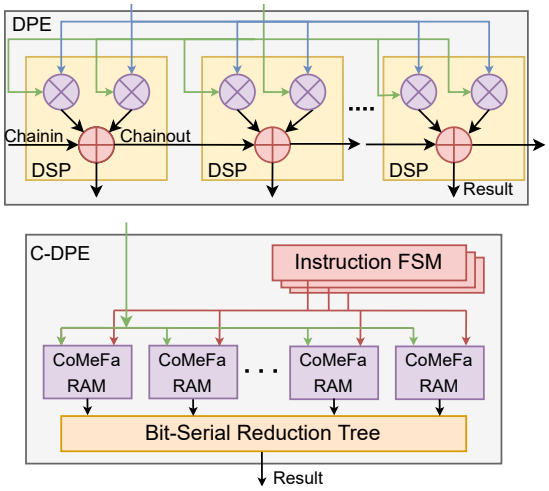
DNNs: To evaluate full neural networks, a Microsoft Brainwave-like accelerator [37] is created based on Boutros et al. [21]. This accelerator consists of five pipeline stages: the matrix unit (MU) for matrix-vector multiplication operations, the selector unit for skipping the MU when necessary, two multi-function units (MFUs) for vector elementwise operations (e.g. activation, addition, multiplication), and the loader (LD) which interfaces with the DRAM to load and unload data. Register files (MRF and VRF) store the data locally. Similarly to CCB [124], two versions of this accelerator are created: one for the baseline FPGA and another for the proposed FPGA. For the baseline FPGA, the MU consists of dot product engines (DPEs) that contain DSP slice cascade chains. Each DPE generates 1 result. For the proposed FPGA,

the MU additionally contains dot product engines that are mapped to CoMeFa RAMs (these are called CoMeFa-DPEs or C-DPEs). The CoMeFa RAMs in C-DPEs receive instructions from instruction generation FSM (duplicated to reduce fanout). A popcount-based bit-serial reduction tree [124] is used to combine the results from various CoMeFa RAMs. Each C-DPE generates 40 results. Figure 5.10 shows the architecture of the accelerator for the proposed FPGA.

An analytical model is written to explore the distribution of data and BRAMs between DPEs and C-DPEs. There are two main knobs in the analytical model - `f_data`, which decides the fraction of workload (in terms of rows of the matrix processed by the MU) processed by DPEs compared to C-DPEs, and `f_arch`, which decides the fraction of BRAMs allocated to DPEs compared to C-DPEs. In addition, the analytical model also varies the number of DSPs per DPE and the number of BRAMs per C-DPE over pre-specified ranges. The analytical model iterates over each layer for each neural network and calculates the cycles consumed for each layer. Then, the results from the analytical model are post-processed using Pandas to find out the best knob (or parameter) settings for each neural network. This results in a different architecture for each neural network. So, instead of having a one-size-fits-all overlay, there is a customized overlay for each neural network. An RTL generator is written to generate the Verilog design for the accelerator with the best hardware parameters identified by the analytical model. Through simulation, sanity verification of the Verilog design and the analytical model's results is



(a) Top-level architecture



(b) Internals of the DPE and C-DPE units

Figure 5.10: Microsoft Brainwave-like accelerator used to evaluate DNN performance

performed.

The Brainwave-like accelerator does not directly support convolutions. So, for CNNs, convolution is expressed as matrix multiplication using the `im2col` operation. It is assumed that the `im2col` operation is performed in hardware. Although this can be optimized by designing an accelerator specifically for convolution, the goal here is to showcase the gains from in-memory computation rather than designing the most efficient accelerator.

For this part of the evaluation, five DNN benchmarks from 3 common DNN types are considered: Fully Connected Networks (Multi-Level Perceptron (MLP)), Recurrent Neural Networks (Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU)), Convolutional Neural Networks (Tiny Darknet) and Residual Neural Networks (Resnet)). The `mlp` network is a 5-layer MLP with each hidden layer having 1024 neurons, with 4M parameters. The `gru` network has a hidden size = 512, embedding size = 512, and time steps = 50. It has 1.5M parameters. The `tdarknet` network is Tiny Darknet, a small image classification network for edge devices. It has 650K parameters. The `lstm` network is an LSTM with hidden size = 1024, embedding size = 1024, and time steps = 50. It has 8.4M parameters. The `resnet` benchmark is the ResNet-50 variation of ResNet. It has 24M parameters.

Two precisions are considered - int8 and int4, and two batch sizes - 1 and 8. The speedup using the two dot product algorithms mentioned in Section 5.1.13 is also evaluated. The FPGA used in the evaluation (Intel Arria 10) is a mid-sized FPGA (47 MBits capacity). Some of the DNNs used for evaluation

Table 5.5: Area breakdown of various RAM blocks

Component	BRAM	CoMeFa-D	CoMeFa-A
Input and output crossbars	5.6	4.5	5.2
Decoders & wordline drivers	7.8	6.3	7.3
Write drivers & sense amps.	6.9	14.0	6.4
Memory cell array	53.4	43.0	49.6
Routing (conn. & switch)	26.0	20.9	24.1
Processing elements	0	11.1	7.1
Total (%)	**100**	**100**	**100**

have weights that do not fit on the FPGA. For int8, `lstm` and `resnet` do not fit. For int4, only `resnet` does not fit. For those cases, the overhead in loading the weights onto the FPGA from DRAM is also considered.

5.2.4 Implementation Details

Area: Table 5.5 shows the area breakdown of both architectures of CoMeFa RAM. For CoMeFa-D, the area overhead is 1546.78 um^2. This represents an increase of 25.4% in the BRAM tile area compared to the baseline. This overhead is mainly attributed to the addition of 160 PEs and the additional 120 sense amplifiers and write drivers. With BRAMs occupying 15% of the die size in the baseline FPGA, this overhead corresponds to only 3.8% increase in the FPGA chip area. The overhead for CoMeFa-A is 493.5 um^2. Compared to the baseline, this represents an increase of 8.1% in BRAM tile area and only 1.2% increase in FPGA chip area. This overhead is mainly attributed to the addition of 40 PEs.

Frequency: COFFE is used to obtain the overhead in the frequency of operation of a CoMeFa RAM in Hybrid mode, compared to a BRAM (735

MHz). For CoMeFa-D, the cycle duration increases to 1.25x (588 MHz). This is mainly attributed to performing read, compute (PE circuitry delay), and write in the same cycle. For CoMeFa-A, the cycle duration increases to 2.5x (294 MHz). This is because 4 reads and 2 writes are done successively as described in Section 5.1.7. In Memory mode, the delay overhead is negligible; there is only one additional mux in the write path and the read path remains unchanged. For experiments, a delay overhead of 1.5% is considered in Memory mode.

Routing: The interface of a CoMeFa RAM block to the programmable routing is not changed compared to that of a BRAM. The only change is the addition of two pins, which are used for direct connections between neighboring BRAMs. These do not impact the programmable interconnect directly, but do increase the pin density.

CCB: The implementation of CCB [124] is based on a BRAM with 128x128 geometry. The area overhead for the CCB block evaluated in [124] does not include the area of the additional sense amplifiers and write drivers. CCB is re-implemented and the total area overhead comes out to be 872.64 um^2, which is a 16.8% increase at the block level and 2.5% at the chip level in the Arria-10-like FPGA used in this study. The frequency of operation of the CCB evaluated in [124] is 1.6x (469 MHz) compared to the baseline BRAM. Table 5.6 shows the differences between CCB and CoMeFa.

Table 5.6: Differences between CCB and CoMeFa

Property	CCB	CoMeFa-D	CoMeFa-A
Activate two wordlines at the same time on one port	Yes	No	No
Additional voltage source required	Yes	No	No
Additional row decoder required	Yes	No	No
Changes in sense amplifiers	Yes	No	No
Additional sense amplifiers	Yes	Yes	No
Sense amp cycling	No	No	Yes
Compute uses dual-port behavior	No	Yes	Yes
Generic/Flexible PE	No	Yes	Yes
Shift between RAM blocks	No	Yes	Yes
Floating point support	No	Yes	Yes
Flip-flops in PE to store operands	No	No	Yes
Parallelism	128	160	160
Application(s) demonstrated	DL	Many	Many
Clock duration overhead	60%	25%	125%
Area overhead (block)	16.8%*	25.4%	8.1%
Area overhead (chip)	2.5%*	3.8%	1.2%
Column multiplexing	No	No	Yes
Practicality	**Low**	**Medium**	**High**

* includes overhead of additional sense amplifiers and write drivers.

5.3 Results

5.3.1 BRAM+PE vs. CoMeFa RAMs

A comparison of CoMeFa RAMs with a normal Block RAM and single-bit bit-serial processing elements implemented in soft logic is performed. The latter is called the "BRAM+PE" architecture. Figure 5.11 shows the block diagram of the BRAM+PE architecture. The BRAM is used in true dual port mode (1024x20) so that each PE can be fed with 2 operands at the same time. Multiplexing logic is provided to allow data to be loaded into the BRAM before the operation, and the results to be unloaded after the operation has finished. When `start` is asserted, the instruction generation FSM (also implemented in soft logic) starts generating instructions. The instruction specifies the BRAM addresses to read to provide data to the PEs. It also specifies the operation to be performed by the PEs. The address to write the results back to is also included in the instruction. After the operation is complete, the `done` signal is asserted and results can be read out from the BRAM.

Qualitatively, the BRAM+PE architecture suffers from the following disadvantages compared to CoMeFa RAMs:

1. More cycles are required because separate cycles are required to read each operand and then write the result, and also in each cycle only 40 bits can be read compared to 160 in CoMeFa RAMs
2. Use of programmable routing/interconnect to transfer data from BRAM to PEs, resulting in higher power consumption

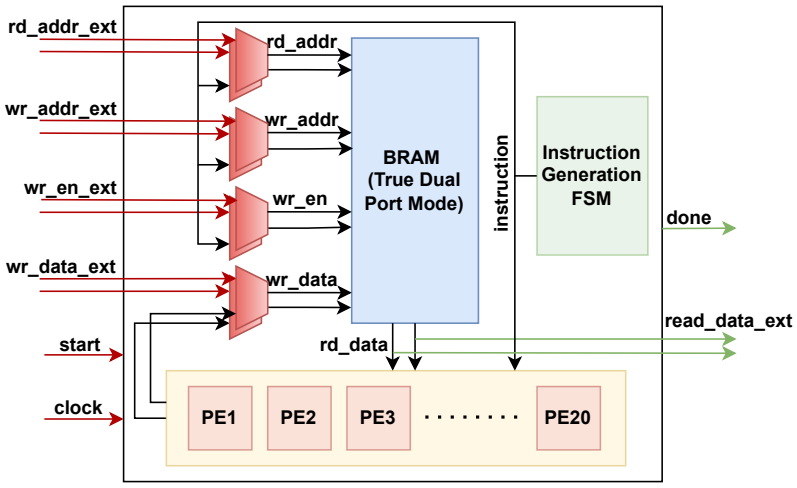


Figure 5.11: Block diagram the BRAM+PE architecture

Parameter	BRAM+PE	CoMeFa
LBs	78	32
RAMs	10	10
Frequency	337.9	536.5
Cycles	64	8
Time (us)	0.19	0.015

Table 5.7: Comparison between CoMeFa RAMs and BRAM+PE

3. Low frequency of operation and higher area because of the PEs and complex control logic being implemented in soft logic

Quantitatively, to compare the BRAM+PE and CoMeFa RAMs, a simple elementwise addition operation is performed on an array of two numbers (precision = 8 bit). Data is laid out inside the RAMs in a transposed manner in both the cases. The results are shown in Table 5.7. It is observed that CoMeFa RAMs use approximately 60% less LBs than BRAM+PE. The BRAM+PE design has the PEs, the multiplexing logic for each RAM interface signal and the instruction generation logic implemented in LBs. On the

other hand, the CoMeFa RAM design does not have any PEs in LBs, and the multiplexing logic and the instruction generation logic is much simpler. The frequency of operation is higher in the CoMeFa RAM case for the same reasons. In the BRAM+PE case, the critical path included routing wires to implement the multiplexing logic, whereas in the CoMeFa RAM case, the critical path is inside an LB. There is a factor of 8 difference in the number of cycles between the two cases. This includes a factor of 4 from the difference in available bandwidth (160 in CoMeFa RAM case vs. 40 in BRAM+PE case) and a factor of 2 from the difference in the number of cycles for each operation (1 cycle to read+compute+write in CoMeFa RAM case vs. 2 cycles to read+compute+write in BRAM+PE case). Overall, the time taken by CoMeFa RAMs is an order of magnitude less compared to the time taken in the BRAM+PE case.

5.3.2 Throughput Comparison

To evaluate the peak throughput, the MAC (multiply-accumulate) operation is considered. MAC operation is the most common operation in DSP and DL applications. Common fixed-point precisions - 4-bit (accumulator=16-bit), 8-bit (acc=27-bit) and 16-bit (acc=36-bit), as well as floating-point precisions - HFP8 ({exp=4, frac=3} and acc={exp=6, frac=9}) [111] and IEEE FP16 (acc=IEEE FP32) are used. The throughput of CoMeFa RAMs is compared to the traditional compute units (LBs and DSPs). For LBs, one MAC is synthesized, placed and routed onto the FPGA, and the operating frequency

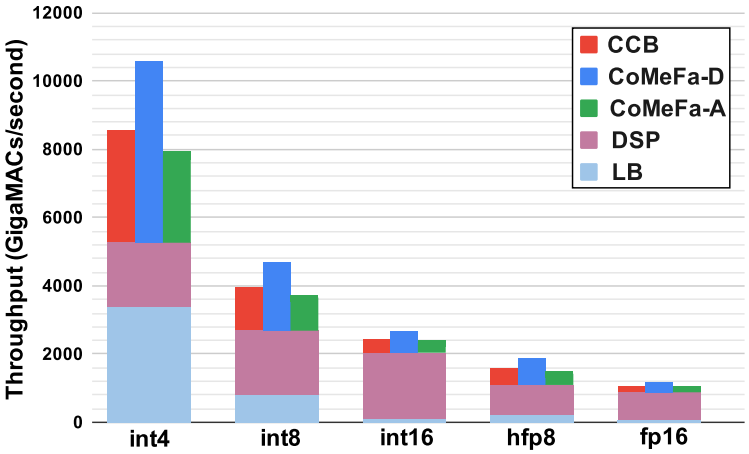


Figure 5.12: Peak throughput for MAC operations for the whole FPGA for various precisions

and resource utilization are determined. Then the throughput is calculated by optimistically assuming that the FPGA can be filled by such MAC units at the same operating frequency. This serves the purpose of evaluating peak throughput. For DSPs, MACs are created and taken through a similar process. The DSPs do not natively support FP16 and HFP8 precisions, so MACs for these precisions are designed using soft logic and DSPs. For CoMeFa RAMs, 160 MACs are implemented in parallel by instantiating one CoMeFa RAM and an instruction generation FSM.

Figure 5.12 shows the peak throughput for each precision obtained from each different computing resource in GigaMACs/second. The throughput of the FPGA increases by 2x, 1.7x, 1.3x, 1.7x and 1.3x for int4, int8, int16, hfp8 and fp16 respectively by adding CoMeFa-D RAMs. Similarly, the throughput of the FPGA increases by 1.5x, 1.36x, 1.16x, 1.36x and 1.15x for int4,

int8, int16, hfp8 and fp16 respectively by adding CoMeFa-A RAMs. CoMeFa RAM throughput reduces as the precision increases, due to the bit-serial nature of computation in CoMeFa RAMs. CoMeFa RAMs can be used for computing in any precision, unlike DSPs. The frequency of operation of CoMeFa RAMs does not change significantly with changing precision, unlike LBs.

Note that the compute throughput enhancement evaluated here is for MAC operations only and does not use OOOR operations. The speedup obtained for different benchmarks can vary from the peak throughput enhancement calculated here because: (1) non-MAC operations like reductions may be needed, (2) clock frequency may be lower because of large designs, (3) cycles may be spent in loading and unloading data to/from CoMeFa RAMs, (4) DRAM reads and writes may bound certain parts of the application, (5) OOOR operations may be used to speed up the operation, and (6) LBs may only be used for control logic and not for computation.

5.3.3 Resource Usage and Frequency

Table 5.8 shows the resource usage and frequency of operation for the various compute bound and DRAM bound microbenchmarks obtained from the VTR flow (averaged over 3 seeds). The table shows the data for the baseline FPGA and the three FPGA variations with compute-enabled BRAMs (CCB, CoMeFa-D and CoMeFa-A). The resource usage for each resource is in percentage of the total resources of that type on the FPGA. The logic block (LB) usage increases significantly for all the microbenchmarks. This

is because of the control logic (instruction generation logic, data loading/unloading logic, etc) required for using the compute-enabled BRAMs. The DSP usage remains the same as baseline, and the usage of RAMs significantly increases. Note RAMs are not used *in-place-of* DSPs, but *additionally*, in order to maximize the usage of the FPGA to exploit the higher compute throughput to obtain speedup. The FIR benchmark uses chaining of RAMs, which is not supported by CCB. Similarly, CCB does not support floating-point operations. So, the FIR and Elementwise Multiplication benchmarks are not implemented on the CCB architecture, and hence the frequency is marked with a "-". For Elementwise Multiplication benchmark, a design with no swizzle modules is constructed, so that the design fits on the FPGA by maximizing the number of CoMeFa RAM for compute. This is done to obtain the theoretical speedup in the case with unlimited DRAM bandwidth (see Section 5.3.4).

Table 5.9 shows the resource usage and frequency of operation of the various on-chip memory bound microbenchmarks obtained from the VTR flow. The resource usage shown here is in absolute numbers. This is because for these benchmarks, a small design is created that uses similar FPGA RAM resources on the baseline FPGA and the FPGA with compute-enabled BRAMs. Bitwise-Search uses more BRAMs in the baseline because of underutilization of the RAM due to the data layout. No DSPs are used for computation in these benchmarks on the baseline FPGA as well the FPGA with compute-enabled BRAMs. A significantly lesser number of LBs are used in the FPGAs with compute-enabled BRAMs because the computation is done internal to the

Table 5.8: Resource Usage (percentage) and Frequency (F, in MHz) for compute and DRAM bound microbenchmarks

	Baseline				FPGA with compute-enabled BRAMs					
Benchmark	**LB**	**DSP**	**BRAM**	**F**	**LB**	**DSP**	**BRAM**	**F(CCB)**	**F(CoMeFa-D)**	**F(CoMeFa-A)**
GEMV	1.6	90.1	43.4	253	27.9	90.1	91.8	231	242	242
GEMM	0.8	92.4	38.6	269	25.5	92.4	86.7	260	267	260
Conv2D	5.0	91.8	28.5	255	35.5	91.8	91.3	245	246	243
FIR	12.8	93.0	3.5	243	53.1	93.0	95.3	-	229	229
Elt Mult	25.8	49.8	38.1	300	21.7*	0	92.6	-	292	288

* does not include LB usage from swizzle modules to capture the infinite DRAM bandwidth case.

Table 5.9: Resource Usage (absolute values) and Frequency (F, in MHz) for on-chip memory bound microbenchmarks

	Baseline				FPGA with compute-enabled BRAMs					
Benchmark	**LB**	**DSP**	**BRAM**	**F**	**LB**	**DSP**	**BRAM**	**F(CCB)**	**F(CoMeFa-D)**	**F(CoMeFa-A)**
Search	2242	0	280	600	1206	0	256	451	465	294
RAID	1538	0	256	702	578	0	256	459	588	294
ReLU	560	0	256	616	301	0	256	445	465	294
Reduction	4072	0	256	445	1184	0	256	453	469	294

BRAMs and LBs are not used for computation. The frequency of operation is very high on the baseline FPGA because the control logic is much simpler compared to the control logic (instruction generation logic) in the FPGAs with compute-enabled BRAMs. An interesting observation is that in the CoMeFa-A case, the frequency of operation is always limited by the frequency of operation of the CoMeFa-A RAM (294 MHz).

5.3.4 Speedup and Energy Benefits

Figure 5.13 shows the speedup obtained by using compute-enabled BRAMs across microbenchmarks. Significant speedups are seen by using CoMeFa RAMs in the compute bound applications because of the augmented

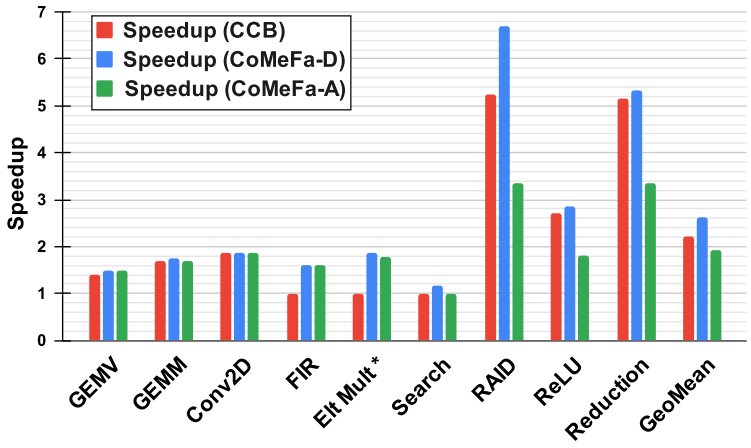


Figure 5.13: Speedups obtained for different FPGA architectures for various benchmarks. $*$ implies no DRAM bandwidth limitation.

compute throughput provided by the FPGA. For GEMV benchmark, speedups of 47.5% are seen in CoMeFa-D and CoMeFa-A. With CCB, the max speedup is 40%. For GEMM, the speedup is 74.5% for CoMeFa-D and 69% for CoMeFa-A and CCB. A speedup of ∼85% is seen in the Convolution benchmark for all 3 architectures, because the frequency of operation is similar in all three, as seen in Table 5.8. A speedup of 59% is seen in the FIR benchmark for both CoMeFa-D and CoMeFa-A. The FIR benchmark uses chaining of RAMs, which is not supported by CCB. So, no speedup is considered compared to the baseline.

Since the Elementwise Multiplication benchmark is limited by DRAM bandwidth, no speedup is seen by using CoMeFa RAMs. CoMeFa RAMs are targeted to improve the compute throughput of the FPGA, not the DRAM bandwidth. If the restriction of DRAM bandwidth is removed and it is assumed that all the compute units (CoMeFa RAMs as well as DSPs/LBs) can

be fed with data, then speedups of 86% and 79% can be seen on CoMeFa-D and CoMeFa-A FPGAs respectively. Since CCB does not support floating-point operations, the speedup for this benchmark for CCB is shown as 0%.

The Search benchmark is sped up by 18% for CoMeFa-D. The design on baseline FPGA has the highest frequency of operation because of very simple operations done in soft logic, as seen in Table 5.9. No speedup is seen using CoMeFa-A RAMs because of the low frequency of operation. This application is not sped up by using CCB either. CCB takes ∼2x cycles compared to CoMeFa RAM because of the inflexibility of the processing elements that only support a few operations. E.g. AND operation can be done in 2 cycles in CCB, compared to 1 cycle in CoMeFa RAM. The RAID application is sped up by 6.7x in CoMeFa-D, 3.35x in CoMeFa-A and 5.2x in CCB. The baseline frequencies are very high in this case also, but the difference in number of cycles enables the significant speedups. In the ReLU benchmark, speedup of 2.7x, 2.85x and 1.8x are seen in CCB, CoMeFa-D and CoMeFa-A respectively. The speedups for the Reduction benchmark (4-bit precision) are 5.3x in CoMeFa-D, 3.3x in CoMeFa-A and 5.1x in CCB.

Results from the energy model are shown in Figure 5.14. The results are similar for the various architectures - CCB, CoMeFa-D, and CoMeFa-A. FIR and Eltwise Mult benchmarks are not run on the CCB architecture as mentioned earlier, so those results are omitted from the figure.

In compute-bound benchmarks (GEMV, GEMM, Conv2D, FIR), an increase in energy consumption is observed. This is because the resource usage

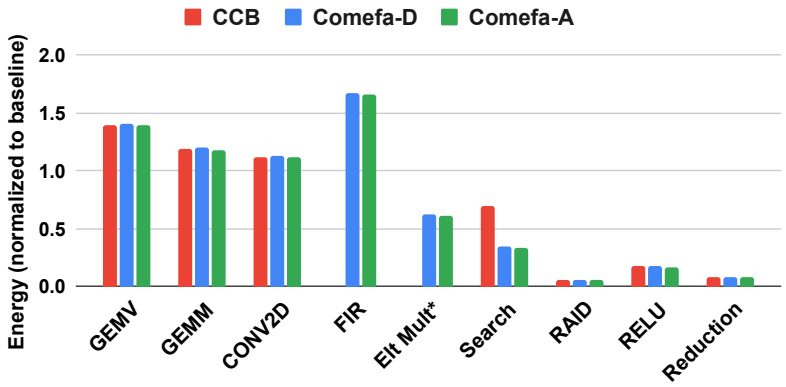


Figure 5.14: Energy consumption for all microbenchmarks.
* implies no DRAM bandwidth limitation.

in these benchmarks is significantly higher compared to the baseline, as seen in Section 5.3.3. E.g., for GEMV, the baseline uses 1.6% LBs, 90.1% DSPs and 43.4% BRAMs, but with CoMeFa RAMs, 27.9% LBs, 90.1% DSPs and 91.8% CoMeFa RAMs are used. The additional LBs are required for control logic to program the CoMeFa RAMs and also for reduction of partial results obtained from CoMeFa RAMs. To reduce fanout from this logic to CoMeFa RAMs (to achieve high frequencies), this control logic has to be replicated multiple times, increasing the LB usage significantly. This increased resource usage leads to a high power consumption. The reduction in time for these benchmarks, compared to the baseline, is less than 2×, as seen earlier in this section from the Speedup results (Fig 5.13). E.g., for GEMV, the speedup is ˜1.47. Since energy is evaluated by combining the power consumption and the time taken, the energy consumption is higher when running these benchmarks on an FPGA with CoMeFa RAMs. These results indicate that using lower precision (e.g.

int4) in these benchmarks could lead to an overall energy reduction because of increased speedup. To confirm this, the GEMV benchmark is implemented with the int4 precision and mapped to an FPGA with CoMeFa-A RAMs. A speedup of 2.83 compared to the baseline is observed. The baseline uses 1.45% LBs, 91% DSPs and 23.8% BRAMs, but with CoMeFa RAMs, 28.8% LBs, 89.5% DSPs and 89.7% CoMeFa RAMs are used. An energy reduction of 24% is observed.

In the memory-bound benchmark (Eltwise Mult), a reduction of 40% is seen in both CoMeFa-D and CoMeFa-A, but note that this excludes the impact from the LBs used for swizzle logic, to showcase the infinite DRAM bandwidth case.

In the on-chip memory bandwidth bound microbenchmarks (Search, RAID, ReLU, Reduction), up to 38% less LBs are used in CoMeFa compared to baseline. That is because no LBs are needed for computation when CoMeFa RAMs are used. Routing WL reduction of up to 68% is seen, which directly correlates to reduction in data movement. This reduces power consumption by up to 56% in CoMeFa-A and up to 52% in CoMeFa-D. With significant reduction in time obtained by using CoMeFa RAMs for these benchmarks as seen in the Speedup results (Fig 5.13), the energy reduction of up to 95% can be seen.

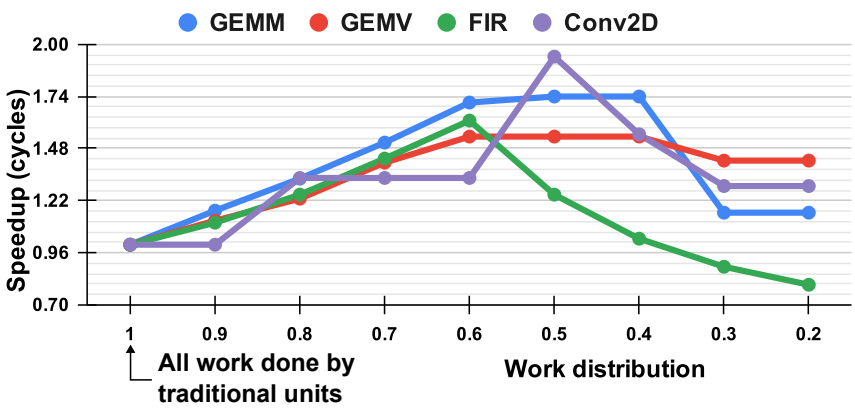


Figure 5.15: Illustration of variation in speedup (based on cycles) by partitioning the application between DSPs and CoMeFa RAMs.

5.3.5 Application Co-mapping

CoMeFa RAMs supplement DSPs and LBs as compute units, and enhance the FPGA's compute throughput. Appropriately dividing the data between CoMeFa RAMs and traditional compute units is key. For the compute bound applications (GEMV, GEMM, FIR, and Conv2D), the effect of varying data distribution between CoMeFa RAMs and DSPs/LBs on the proposed FPGA is explored analytically. The results are shown in Figure 5.15. As more work is given to CoMeFa RAMs, more speedup can be obtained upto a limit, after which the overheads (loading, unloading, serial compute) associated with CoMeFa RAMs can start dominating and reduce the overall speedup. This sweet spot is different for each application. In some cases, mapping a majority of the application onto CoMeFa RAMs can even cause an overall slowdown because of higher latency.

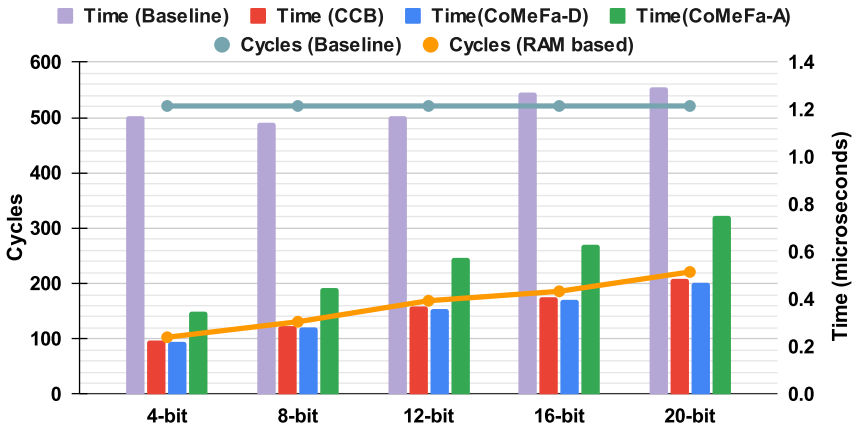


Figure 5.16: Sweeping precision in the Reduction benchmark

5.3.6 Adaptability to Precision

CoMeFa RAMs can be used for efficiently computing in any custom precision. Figure 5.16 shows the results of sweeping the precision from 4-bit to 20-bit in the Reduction benchmark. Speedups ranging from 5.3x (3.3x) to 2.7x (1.7x) are seen with CoMeFa-D (CoMeFa-A) as precision increases. CoMeFa-D is 3% better than CCB owing to the improved frequency achieved by the design. The baseline takes the same number of cycles for each precision because of the bit-parallel nature of compute. But the number of cycles taken increases as the precision increases when CoMeFa RAMs are used. This is because of bit-serial arithmetic, and illustrates that applications using smaller precisions are better suited for CoMeFa RAMs. Note that the frequency of operation stays constant for CoMeFa RAMs because the hardware architecture stays the same. For the baseline, the frequency decreases slightly as the precision increases.

Benchmark	LB	DSP	BRAM
GEMV	49.0	90.1	91.7
GEMM	53.5	92.4	86.7
Conv2D	55.2	91.8	91.4
FIR	63.0	93.0	95.7
Elt Mult	24.2	0	84.2

Table 5.10: Resource usage when instruction generation logic is implemented using the stored program method

Benchmark	Customized FSM	Stored Program
GEMV	1.47	1.07
GEMM	1.69	1.22
Conv2D	1.85	1.45
FIR	1.59	1.41
Elt Mult	1.79	1.21

Table 5.11: Speedup obtained when instruction generation logic is implemented using a customized FSM and using the stored program method

5.3.7 Using stored programs instead of hardcoded FSM

For each microbenchmark, a design that uses the stored program method discussed in Section 5.1.14 is created. Table 5.10 shows the resource usage of benchmarks when the stored program method is used. Comparing this to the resource usage in Table 5.8, it is observed that the LB usage increases significantly. This increase is attributed to the relatively more generic instruction controller logic, compared to the hardcoded FSM logic which can be highly specialized and optimized for a specific benchmark. The DSP usage remains exactly the same, because the instruction controller logic does not use any DSPs. The BRAM usage remains almost the same, because of how the experiment is designed. Some BRAMs used for computation are instead repurposed to be used for storing instructions. The minor differences arise because the

number of CoMeFa RAMs removed from compute units in the benchmark and the number of instruction RAMs required by the remaining compute units may differ.

The speedup obtained by using the stored program method is compared with the speedup obtained using hardcoded FSM based method in Table 5.11. It is observed that the speedup reduces by ~40% on average. But there is a significant hard-to-quantify improvement in programmability of the CoMeFa RAMs by using the stored program method. There are two main reasons for reduction in the speedup. Firstly, instruction storage in the stored program method can consume a significant number of BRAMs, reducing the BRAMs available for compute. The hardcoded FSM method does not use any BRAMs in the instruction generation logic. Secondly, when using the stored program based method, a reduction in frequency of operation of the design is observed. The critical path is in the instruction decoder of the controller.

In the future, improving the speedup obtained with the stored program method is planned by (1) adding pipeline stages in the controller to improve frequency, and (2) mapping macro-instructions to distributed RAMs in LBs to keep the number of BRAMs available for compute the same.

Note that the reduction in speedup is evaluated only for compute-bound and DRAM-bound microbenchmarks here. For the on-chip memory bandwidth microbenchmarks, the experimental setup is such that a small number of BRAMs is used in both baseline and proposed cases. A few extra BRAMs can be used to store instructions and obtain the same speedup as the case with

hardcoded FSM.

5.3.8 DNN evaluation

Fig 5.17 shows the speedup obtained by using the accelerator shown in Fig 5.10, for 5 DNNs along with the geometric mean. The baseline uses an accelerator without C-DPEs, on an FPGA without CoMeFa RAMs. Three knobs or parameters are varied - precision, batch size, and dot product algorithm. The frequency of operation of the accelerator is the same for both cases - using CoMeFa-D and using CoMeFa-A - because the critical path of the design is not in the Matrix Unit of the accelerator. So, these speedups apply to both cases.

In Fig 5.17a, a geomean speedup of 1.26x is seen with int8 precision, and that increases to 2.49x with int4 precision. Because of the bit-serial computation in CoMeFa RAMs, smaller precision exhibit low latencies and higher speedups compared to the baseline. These speedups are for a batch size of 8 using Algorithm 2. Fig 5.17b, the speedups for batch size of 4 and batch size of 8 are compared, for int4 precision and Algorithm 2. The speedup increases with batch size because of improved utilization, higher reuse, and amortization of weight loading (when needed). In Fig 5.17c, the speedups obtained by using Algorithm 1 and Algorithm 2 are compared, for a batch size of 8 and precision of int4. Algorithm 1 is slower than Algorithm 2 because Algorithm 2 takes advantage of inspecting one bit each from two operands outside the RAM, and reduces cycles by up to 2x.

In Fig 5.18, the trends of varying the knobs `f_arch` and `f_data` (discussed in Section 5.2.3) can be seen, based on the results of the analytical model. The number of cycles consumed (normalized) for each DNN is plotted along the y-axis. In the top chart, `f_data` is kept constant at 0.5, implying 50% of the workload (matrix rows) is assigned to the DPEs and the rest 50% is assigned to the C-DPEs. For higher values of `f_arch`, the number of cycles consumed is high because enough BRAMs are not available for the 50% of the workload assigned to C-DPEs. On moving left along the x-axis, `f_arch` reduces, and more BRAMs become mapped to C-DPEs, which in turn means the part of the workload assigned to C-DPEs can be executed efficiently, reducing the overall cycles consumed.

In the bottom chart in Fig 5.18, `f_arch` is kept constant at 0.5, implying that 50% of the BRAMs are assigned to DPEs and 50% are assigned to C-DPEs. When `f_data` is low (on the left of the x-axis), only a small amount of the workload is assigned to C-DPEs, so there isn't much speedup. But on moving right along the x-axis, more of the workload is assigned to C-DPEs, achieving a lower number of cycles. But on moving further right, the cycles start to increase again because the latency of the C-DPEs starts to dominate. Medium values of `f_data` give the highest speedup.

5.3.9 Integration into an open acceleration framework

To demonstrate using CoMeFa RAMs with already existing acceleration frameworks, a CoMeFa RAM based acceleration unit is integrated into

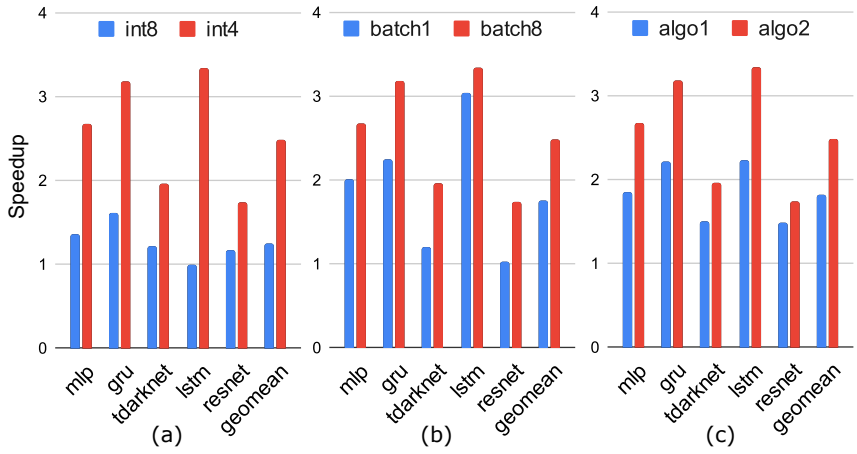


Figure 5.17: Variation of speedup for DNNs with precision, batch size, and dot product algorithm

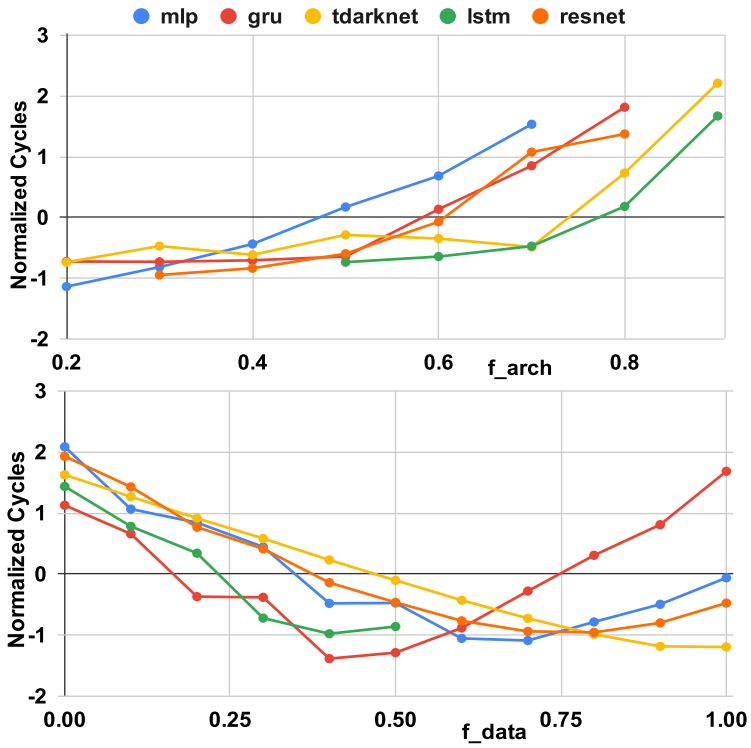


Figure 5.18: Variation of cycles with changing f_arch and f_data

CFU Playground [97] from Google and Harvard. Figure 5.19 shows the overall architecture of the system. CFU stands for Custom Functional Unit. CFU Playground is a collection of software and hardware to make it easy for everyone, including software engineers, to accelerate ML/DL inferencing. Overall, the system provides a soft RISC-V based SoC that can be mapped to any FPGA, with a simple C-based programming interface, along with the capability to design a CFU that is easily hooked up to the CPU. In the original CFU Playground framework, the CFU is tightly coupled to the pipeline of the CPU. Both commands and data to the accelerator are sent from the CPU interface. This limits the acceleration that can be achieved because of the large amount of data to be transferred using a narrow interface. The framework is enhanced by adding a direct-memory-access (DMA) path from the accelerator unit shown using red arrows in the figure (currently only using a simulation model).

A C program is written to first populate the instruction RAM in the accelerator. Then the accelerator is triggered by writing into a control register. This trigger initiates the instruction controller in the accelerator to fetch data into the CoMeFa RAMs using the DMA path via the swizzle logic. A status register is read to ensure that the data transfer has been completed. Then another command is sent by writing to a control register in the accelerator. This command initiates the controller to fetch the instructions from the instruction RAM in the accelerator, and decode and execute them on CoMeFa RAMs. After the execution has finished, another status register is set. The

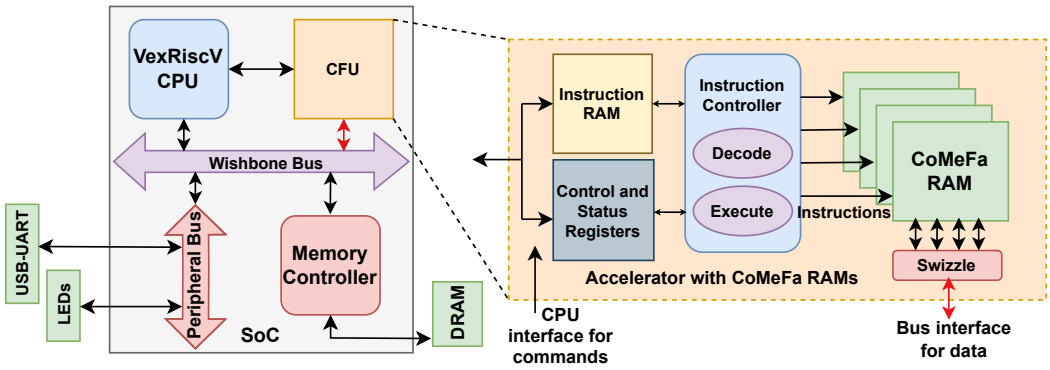


Figure 5.19: Integrating a CoMeFa RAM based unit into an SoC using an open-source accelerator framework (CFU Playground)

CPU busy-waits on this status register until the execution is finished (there are no interrupts currently).

Kernels are deployed to perform elementwise addition and multiplication on large arrays of data, using this framework. With the C-based interface of the RISC-V CPU and the instruction based interface of the CoMeFa RAMs, it is very easy to develop and use this accelerator. Comparing the cycles required to perform the same kernels using DSP-based accelerators, the speedup obtained is similar to that in Section 5.3.4.

5.3.10 Impact on non-DL benchmarks

Some of the microbenchmarks used for evaluation of CoMeFa RAMs in the previous sections are non-DL, for example, FIR filter and the bitwise (search and RAID) benchmarks. This illustrates that CoMeFa RAMs can be used for compute (by configuring them in their Hybrid mode) in non-DL

benchmarks as well. However, in applications like legacy non-DL designs, CoMeFa RAMs are not used in their Hybrid mode (for compute). They are used in the Memory mode. To comprehensively evaluate the impact of adding CoMeFa RAMs on non-DL benchmarks which use CoMeFa RAMs in memory mode, VTR benchmarks [85] are run on 3 FPGA architectures: (1) Baseline FPGA architecture with Block RAMs, (2) Proposed FPGA architecture with CoMeFa-D RAMs, and (3) Proposed FPGA architecture with CoMeFa-A RAMs.

VTR is run with auto layout enabled (meaning the grid size expands based on the resources required by the design), the default timing-driven routing option with a maximum of 150 routing iterations, and a fixed channel width of 300 wires. When running VTR, an SDC (Synopsys Design Constraints) file is provided in which the target clock period is set to 0 (i.e. VTR will optimize the design for maximum clock frequency), and timing analysis for paths to/from the FPGA IOs is disabled.

VTR reports the resoure usage, achieved frequency, and routing wirelength for each benchmark. The total area consumed by a circuit on an FPGA is the sum of the logic area and the routing area. Logic area is available in the VTR output report, but routing area is not. The routing area is estimated approximately by adding the routing area of all tiles that had atleast one operation mapped to. To reduce CAD noise, the results of each benchmark are averaged first over six different seeds. Then, the results of each benchmark are geometrically averaged to get the overall results plotted in Fig 5.20.

Resource usage remains the same for all 3 cases (the only difference is in the type of RAM used). Fig 5.20 compares the area, frequency and routing wirelength. For area, increases of 2.3% and 0.75% are seen when CoMeFa-D RAMs and CoMeFa-A RAMs are used respectively. CoMeFa-D RAMs are larger than CoMeFa-A RAMs, which are larger than BRAMs, so this difference is expected.

The change in frequency is negligible (less than 0.5%) for both CoMeFa-A and CoMeFa-D cases. In fact, for some benchmarks, the achieved frequency is higher than the baseline for some seeds. At the block level, the baseline Block RAM and the CoMeFa RAMs operate at significantly high frequency than the overall frequency of operation of the circuit. As mentioned in Section 5.2.4, Block RAM operates at 735 MHz, and this frequency is reduced by only 1.5% *in memory mode* for CoMeFa-D and CoMeFa-A. The overall circuit frequencies are unaffected by this minor reduction in frequencies of CoMeFa RAMs. If the degradation in frequency in memory mode was significant, the CoMeFa RAMs could have been in the critical path and reduced the frequency of operation in some benchmarks.

The change in routing wirelength is negligible (less than 0.5%) for both CoMeFa-A and CoMeFa-D. This is because in the evaluation done here, the number of rows and columns in the FPGA grid consumed by a CoMeFa RAM remains the same as a Block RAM. Only the width of a CoMeFa RAM column is increased to accommodate for the increase in area. However, VTR does not model grid columns of different widths.

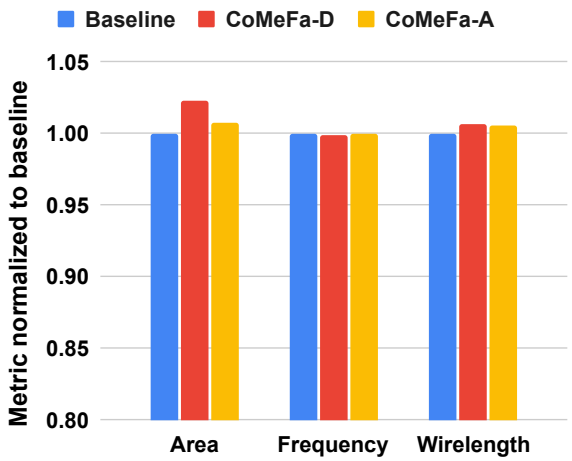


Figure 5.20: Impact of adding CoMeFa RAMs to an FPGA on non-DL benchmarks

5.4 Discussion

5.4.1 Benefits and Limitations

Qualitatively, the advantages of using CoMeFa RAMs can be summarized as:

1. CoMeFa RAMs transform BRAMs, which can only be used for storage, into highly parallel SIMD engines. This parallelism provides significant throughput boost for an FPGA. When not computing, CoMeFa RAMs can still function as normal BRAMs (data does not need to be transposed).
2. Because the computation happens inside the memory block, data movement between various blocks on the FPGA is significantly reduced. The requirement of precious routing/interconnect resources is reduced. This leads to reduction in energy consumption, reduced routing congestion, and potentially faster frequencies.
3. Any custom operation with any custom precision can be supported by a CoMeFa RAM block. For performing a different operation or for using a different precision, a different instruction sequence needs to be generated and applied to the CoMeFa RAM. This makes CoMeFa RAMs universally applicable for any application domain.
4. CoMeFa RAMs increase the effective bandwidth available for computation. For Intel BRAMs, in one cycle, 160 data bits can be processed in parallel, instead of 40 (an external block like DSP/LB can only access 40 bits from a BRAM at a time).

5. Replacing BRAMs in an FPGA with CoMeFa RAMs increases the compute density of FPGAs ($GOPS/mm^2$). A smaller Silicon area will potentially be required for a given circuit. More Silicon area can be dedicated to compute an application resulting in significant speedups.

There are some limitations of converting FPGA BRAMs to CoMeFa RAMs. First, because a CoMeFa RAM is larger in area than a BRAM, the proposed FPGA is larger in size compared to the baseline FPGA. There is an area overhead of 3.8% for an FPGA with delay-optimized CoMeFa RAMs and of 1.2% for an FPGA with area-optimized CoMeFa RAMs. Secondly, some overhead is seen on non-DL applications (upto 2.3% in area, less than 0.5% in frequency, less than 0.5% in routed wirelength). Thirdly, CoMeFa RAMs take longer for one operation because of their bit-serial nature, leading to higher latency. In many cases, this latency can be hidden or overlapped with other operations to obtain speedup, similar to how speedup is achieved for benchmarks in Section 5.3. Another drawback of CoMeFa RAMs is the new programming model. In this dissertation, a stored program method for programming CoMeFa RAMs is proposed, which makes them much easier to program. Even then, a new block means tools need to be updated and libraries need to be created that can enable users to use them easily.

5.4.2 Comparison with other FPGA blocks

CoMeFa RAMs are universal blocks and can be used for accelerating any application. CoMeFa RAMs are not replacements of DSPs or LBs, but

can work together and complement them. In some ways, CoMeFa RAMs can be thought of as blocks that fuse together Logic Blocks and BRAMs. They provide a more structured way of computation compared to Logic Blocks, along with the storage capability of BRAMs. Compared to DSPs, CoMeFa RAMs provide infinite flexibility in terms of precisions (because of their bit-serial nature) even after the chip has been designed. DSPs can support multiple precisions too, but the precisions have to be hardened at the time of designing the FPGA, and adding more precisions increases DSP area. CoMeFa RAMs support more operations than DSPs (which mainly just support multiplication and addition) because of the configurable processing element in them. They also allow flexibility in which algorithm to use for multiplication and addition, through bit-serial and OOOR operations. With DSPs, a user is forced to use the multiplier or adder architecture that is designed into it.

5.4.3 Applications

Applications that are well-suited for deploying CoMeFa RAMs are:

1. Applications that have significant SIMD parallelism, e.g., Deep Learning and signal, image and video processing.
2. Applications that do not require a lot of communication between processing elements, e.g., elementwise and bitwise operations.
3. Applications that use reduced and/or custom numerical precisions, e.g., Deep Learning.

5.4.4 Organizing data for computation

When performing computation using CoMeFa RAMs, data is laid out in a transposed manner, i.e. bits of an element are stored along a bitline. In addition to operands and results, intermediate results need to be stored in the same column. If the intermediate results are not required at a later stage, they can be overwritten to improve RAM utilization. For example, consider the case where 4 operands a, b, c, d are stored in a column and the operation required is $e = a*b+c*d$. The intermediate results $a*b$ and $c*d$ are calculated bit-serially, and then added. The final result can reuse the same rows as those storing the intermediate results by overwriting them.

In some cases, complex operations can be split over multiple columns and the final results can be obtained by reducing intermediate results from different columns. For example, in the example of calculating $e = a*b+c*d$, if the precision of operands is 18-bit fixed point, total available rows (128) would not be enough to store the operands, intermediate results, and the final result. Splitting independent operations over multiple columns exposes more parallelism and can achieve better speedups. However, reducing intermediate results stored in different columns involves serialization and can reduce the obtained speedup. Bit-slicing [124] is another method in which individual elements are split over multiple columns. For example, a 16-bit number can be sliced into 2 8-bit chunks and stored in 8 rows in two columns. Operations can be performed independently on the two bit-slices and then concatenated and reduced appropriately to get the final result.

There are tradeoffs in RAM utilization, data reuse, and latency. Having a smaller number of operands in one column (bitline) means reduced utilization, but low latency, because it will take fewer cycles to perform the operation (because there is only 1 PE per column). However, storing more operands in a column means that the RAM will have higher utilization, but the latency will be higher. Also, more operands in a column allow for more reuse. Consider an example where one operand needs to be multiplied by 2 other operands and then the partial products need to be added. Having all three operands in one column allows this operation to happen locally. Otherwise, two multiplications will need to be done in separate columns (or separate BRAMs) and then reduced. Smaller precisions allow for more elements to co-exist in a column.

Consider an operation where elements stored in all 160 columns of a CoMeFa RAM need to be reduced to get one final result. Performing in-CoMeFa RAM reductions will proceed in a tree fashion, where after the first reduction step, 80 columns will have the new partial results. Then, these will be reduced further into 40 columns. At some stage, performing further in-CoMeFa RAM reductions can significantly degrade the compute throughput, as only a successively smaller portion of the CoMeFa RAM columns are actively performing compute during each reduction iteration. To avoid this, reduction can be performed in soft logic outside the CoMeFa RAMs. 40 partial results can be read out one bit-slice at a time (bit 0 of 40 partial results in 1 cycle, bit 1 of these partial results in the next cycle, and so on) and reduced externally. A popcount based external reduction [124] can be used for this

addition/accumulation.

5.4.5 Parallelism

The parallelism CoMeFa RAMs provide (SIMD) is different from the pipeline parallelism that is commonly used with Logic Blocks and DSP Slices. Although SIMD parallelism can be achieved with LBs and DSP Slices as well, CoMeFa RAMs provide it in a more efficient and compact form. In applications where SIMD parallelism from CoMeFa RAMs is used to obtain speedup (e.g. the on-chip memory boundary bound applications in Section 5.2), reduced data movement will typically be observed leading to a significant reduction in energy.

In addition to SIMD parallelism, data parallelism (splitting the data to be processed between traditional DSP and LB based compute units, and CoMeFa RAM based compute units) is used to exploit the additional compute throughput provided by CoMeFa RAMs. Consider the case where time T is spent on processing D chunks of data on a baseline FPGA (using only traditional compute units like LBs and DSPs). However, if a part of the data (e.g. $D/3$ chunks) is processed by CoMeFa RAMs and if the rest of the data ($2D/3$) is processed by traditional compute units in parallel, then the total time taken would be less than T (say $2T/3$). This achieved speedup depends on the distribution of work between traditional units and CoMeFa RAM based units, and the best case would be when both types of units finish in an approximately equal amount of time. In applications where data parallelism

is used to obtain speedup (e.g. compute bound applications in Section 5.2), energy consumption may not reduce because more hardware is used to solve the problem.

Different applications may need different types of parallelism. Even parts of one application may be suited for different types of parallelism. So, adding CoMeFa RAMs to FPGAs opens the door to new ways to exploit parallelism efficiently.

Chapter 6

DL Benchmarks for FPGA Architecture Research

In this chapter, the third contribution of this dissertation is described: an open-source benchmark suite of DL acceleration benchmark circuits for FPGA architecture and CAD research called Koios[1]. This benchmark suite is currently in its second version (2.0). The Koios suite consists of 40 benchmarks that capture a wide variety of accelerated neural networks, design sizes, numerical precisions, and circuit characteristics. To maximize the utility of these benchmarks, they are made compatible with the Verilog-to-Routing (VTR) flow [85], which is the most widely-used FPGA architecture and CAD research framework. Researchers can use these benchmarks seamlessly with VTR and, with minor modifications, can also use them with other tool chains.

The Koios benchmarks are representative of modern DL workloads; many of them are re-created from prior works and some are replicas of industrial benchmarks. In addition to being more pipelined and heavily using FPGA hard blocks, these benchmarks have higher usage of structures like wide buses, large reduction trees, hard block dedicated cascade routing and

[1] *Koios* (also written as *Coeus*) is the Titan of intelligence in Greek mythology. Unlike the Titan benchmarks [86], Koios focuses on deep learning.

large fanouts. This makes the Koios benchmarks better suited for DL-targeted FPGA architecture exploration than other non-DL benchmark suites.

This part of the dissertation resulted in a paper publication at the IEEE International Conference on Field-Programmable Logic and Applications (FPL) [12] and a research article at the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) [11]. The following contributions from the co-authors of these papers/articles are acknowledged:

- Andrew Boutros: Implementation of `clstm` and `intel_dla` benchmarks, comparison with Intel Quartus, running some experiments, drawing some figures/charts, and significant help with writing the paper/article
- Seyed Alireza Damghani: ODIN and Yosys support/enhancements
- Karan Mathur: Implementation of proxy and `deepfreeze` benchmarks
- Vedant Mohanty: Implementation of `tpu_like.ws` benchmarks
- Tanmay Anand: Implementation of `bwave_like` benchmarks
- Daniel Rauch: Implementation of `tdarknet_like` benchmarks
- Aishwarya Rajen: Implementation of `lstm` and `attention_layer` benchmarks
- Aatman Borda: Implementation of `spmv` benchmark
- Samidh Mehta: Implementation of `robot_rl` benchmark
- Sangram Kate: Implementation of `tpu_v2` benchmark (not included in Koios)
- Pragnesh Patel: Implementation of `softmax` benchmark

6.1 Koios Benchmarks

6.1.1 Overview

The Koios benchmark suite is a DL-specific benchmark suite for FPGA research. It consists of 40 benchmarks covering a diverse representative space, coming from various applications within the DL domain. Table 6.1 provides an overview of the benchmarks and their properties. These benchmarks are completely open-source, and both Verilog HDL source codes and BLIF netlists are provided.

Table 6.1: The Koios Benchmarks (in decreasing order of number of netlist primitives)

Benchmark	Description	Implementation	Network	Precision	Acc. Paradigm	2D Systolic	Winograd/FFT	Reduction	Buffers	DSP usage	Cent. buffers	Based on	Other Properties
dla_like (S/M/L)	Intel-DLA-like accelerator	RTL	CNN[2]	int8/16	Overlay	✓		✓[3]	✓[4]	✓		[16][22]	Daisy chain
clstm_like (S/M/L)	CLSTM-like accelerator	RTL	RNN	int18	Overlay	✓		✓[3]	✓			[123]	Circular compression
deepfreeze	ARM FixyNN design	RTL	CNN	int4	Layer		✓	✓	✓			[126]	Hardcoded weights
tdarknet_like (S/L)	Accelerator for Tiny Darknet	HLS	CNN[12]	fp16	Custom			✓[3]	✓			[100]	Fused layer pairs
bwave_like	Microsoft-Brainwave-like design	RTL	Any	int8, bfp11	Overlay		✓	✓	✓[4]			[37]	Mat-vec mult unit
lstm	LSTM engine	RTL	RNN	int16	Layer		✓	✓	✓				Streaming dataflow
bnn	4-layer binary neural network	HLS	MLP[1]	binary	Custom				✓			[89]	int16 act/norm
lenet	Accelerator for LeNet-5	HLS	CNN	int8	Custom			✓	✓			[76]	5x5 conv layers
dnnweaver	DNNWeaver accelerator	RTL	Any	int8	Overlay	✓		✓[3]	✓	✓		[104]	DDR and PCIe intf
tpu_like.ws (S/L)	Google-TPU-v1-like accelerator	RTL	Any[12]	int8	Overlay	✓		✓	✓	✓		[65]	Weight stationary MMU
tpu_like.os (S/L)	Google-TPU-v1-like accelerator	RTL	Any[12]	int8	Overlay	✓		✓	✓	✓		[65]	Output stationary MMU
gemm_layer	Matrix multiplication engine	RTL	MLP	bfloat16	Layer	✓			✓	✓			AXI interface
attention_layer	Transformer self-attention layer	RTL	RNN	int16	Layer		✓	✓[3]	✓			[118]	GEMV based
conv_layer	GEMM based convolution	RTL	CNN	int16	Layer	✓		✓	✓	✓			3x3 filters
robot_rl	Robot+maze application	RTL	RL	int8/16/32	Custom			✓	✓	✓		[106] [28]	Q-learning algo
reduction_layer	Add/max/min reduction tree	RTL	Any	int16	Layer		✓	✓		✓			Reduces 128 inputs
spmv	Sparse matrix vector multiplication	RTL	MLP	int8	Layer			✓	✓	✓		[38] [130]	COO sparsity enc.
eltwise_layer	Matrix elementwise add/sub/mult	RTL	Any	bfloat16	Layer			✓	✓	✓			Broadcast heavy
softmax	Softmax classification layer	RTL	Any	fp16	Layer			✓		✓		[125]	LUT based exp/log
conv_layer_hls	Sliding window convolution	HLS	CNN	fp16	Layer				✓	✓			1x1 filters
proxy	Proxy benchmarks	RTL	-	-	-	-	-	-	-	-	-	Sec 6.1.4	-

[1] Has Normalization layer [2] Has pooling layer [3] Uses double buffering [4] Has DSP cascade chains

6.1.2 Diversity and Representativeness

The Koios benchmarks cover a wide variety of design sizes, implementation styles, target neural networks, acceleration paradigms, numerical precisions, and circuit properties.

- **Design Size**: The smallest design has 12,097 netlist primitives while the largest has 1,608,867. Any latch, gate or hard block resulting from logic synthesis counts as a netlist primitive. Some benchmarks, such as `clstm_like, dla_like, tpu_like`, have multiple size variants (i.e. small, medium, large). In these cases, the size indicates the parallelism factor used in the design. Bigger designs create a more challenging optimization problem for the CAD tools, while smaller ones have faster compilation time, suitable for early-stage experiments.
- **Implementation Style**: Although all the designs in the benchmark suite are provided to users in the form of Verilog HDL implementations, some are originally implemented in RTL while others are automatically generated from higher-level language descriptions using high-level synthesis (HLS) tools. HLS-generated designs typically have specific design characteristics that are not very common in hand-coded RTL designs, such as widely distributed control signals and complex state machines.
- **Target Neural Network**: Koios benchmarks cover all major classes of neural networks. These include: multi-layer perceptrons (MLPs), convolutional neural networks (CNNs), recurrent neural networks (RNNs), and reinforcement learning (RL). These different classes have different compute

and memory requirements, which reflects on the resource breakdown and routing patterns of their corresponding benchmark circuits. Some designs are also generic and can be used to accelerate any type of network.

- **Acceleration Paradigm**: FPGAs are used for acceleration of DL workloads in different ways. One way is to design a flexible software-programmable overlay architecture that can execute different DL models without the need to reprogram the FPGA with a new bitstream similar to the Microsoft Brainwave [37] architecture. These designs tend to have instruction decoders and more complicated control logic to enable this level of flexibility. In other cases, a custom network-specific dataflow architecture is mapped to an FPGA to maximize efficiency, similar to [47]. The control logic of these circuits is usually hard-coded and implemented as relatively simple state machines. Another approach is to implement layer-specific accelerators that are invoked by software running on a host or an embedded CPU. These circuits are mostly streaming-style datapaths with simple or even no control paths. Koios benchmark suite contains designs from all three acceleration paradigms.
- **Numerical Precisions**: One of the main advantages of using FPGAs to accelerate DL workloads is the ability to design hardware for custom numerical precisions, which is a commonly used technique in accelerating DL workloads [101]. The designs in Koios use various precisions, including: binary (`bin`), different fixed point types `int4/8/16/18/32`, brain floating point (`bfloat16` [122]), IEEE half-precision floating point (`fp16`), and block

floating point (`bfp11` [37]). This diversity is useful for exploring new DSP block architectures and different hard arithmetic circuitry.

- **Circuit Properties**: Koios benchmarks have varying circuit styles that can exercise different components of the CAD tools in different ways. For example, regular structures like systolic arrays can be used for optimizing placement algorithms, large reduction trees can form local routing congestion that stress the routing algorithms, long cascades (or chains) of hard blocks impose harder placement constraints, etc. The benchmarks are also highly heterogeneous (i.e. use different types of FPGA resources) with varying degrees. They utilize a large number of DSP blocks and BRAMs. DSPs are often used to form dot product units and memory structures like double-buffered RAMs and FIFOs are commonly used to store on-chip weights and activations.

6.1.3 Curating the benchmark suite

The designs in the benchmark suite are chosen keeping representativeness and diversity in mind. These designs are implemented (either handcoded or script generated or using HLS) and tested using commercial FPGA tools for ease of development and debugging. Then, many modifications are done to these designs to ensure their compatibility with the VTR flow. Vendor-specific and architecture-specific IP cores (e.g. floating point adders and multipliers, RAM macros) are replaced with ones that are compatible with VTR and the FPGA architecture file used for experiments. This process is especially challenging for the designs generated from HLS tools which tend to be non-

human-readable in many cases. After that, various experiments are run to ensure the suitability of these benchmarks.

6.1.4 Proxy benchmarks

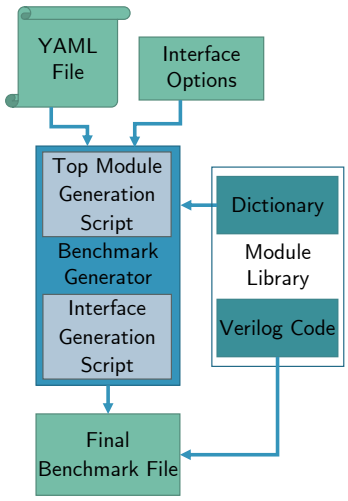
Having a larger set of benchmark circuits is desirable for most FPGA architecture and CAD research. Obtaining real world designs and curating them to be used as FPGA benchmarks is a tedious process as it requires re-creating designs that are not publicly available or modifying existing ones to be compatible with open source CAD tools. Hence, deriving inspiration from other fields [94, 105], a framework for generating synthetic DL benchmark circuits is created. The synthetic benchmarks generated by this framework have similar properties and circuit compositions to real DL benchmarks as described in the previous section. Since these benchmarks can be used as proxies of real DL designs for FPGA architecture and CAD research, they are referred to as *proxy benchmarks*. Unlike the other benchmarks in Koios, the generated proxy benchmarks are not functional DL accelerators – they instead mimic the composition of key components of DL accelerators. Statistical analysis is performed on the properties of real designs and synthetic designs generated from this framework, and compare them in Section 6.2.4.

Proxy benchmarks are generated using design components that are commonly present in real DL designs. Different components from the existing benchmarks are extracted and parameterized, and new components are designed to create a library of modules that can be used in the generation of

Table 6.2: Circuit components used to generate proxy benchmarks

Type	**Properties**
Adder Tree	Adder tree levels {3,4,5}, Precision {16, 8, 4, fp16}
DSP Chain	# DSPs {2,3,4}, Precision {16, fp16}
Systolic Array	Array size {4x4, 8x8}, Precision {4, 8, fp16}
Activations	# of logical LUTs {32}, Precision {8, 16}
Dot Product	Dot product length {10}, Precision {8, bf16}
RAM	Depth {2048, 4096}, Width {40, 60}, # Ports {1, 2}
Double Buffer	Depth {2048, 4096}, Widths {40, 60}
FIFO	Depth {256, 512}, Widths {40, 60}

proxy benchmarks, as listed in Table 6.2. This library can be easily extended to increase the diversity of the generated proxy benchmarks. In addition to the Verilog implementation of these components, the library also contains a Python dictionary of the various components along with their properties (e.g. size, precision, width) and the resource usage of each module for the FPGA architecture used for evaluation. Fig. 6.1a shows how the proxy benchmark generation framework works. The benchmark generator takes as input a YAML format file which specifies the graph structure the user desires (i.e. the specific hardware components and the connections between them). A snippet from a sample YAML file is shown in Fig. 6.1b. For each component, its specific parameters (e.g. type, size, precision) are also specified in the YAML file. The generator goes through the graph structure described by the user, instantiates the corresponding components in the top-level module, and automatically generates the interconnections between them to generate the Verilog file of the proxy benchmark.



(a) Flow overview

```
fifo1:
  type: "fifo"
  size: 256
  precision: 40
  number: 4
  inputs: [top]
  outputs: [dsp_ch1]
dsp_ch1:
  type: "dsp_chain"
  size: 4
  precision: 18
  number: 8
  inputs: [fifo1, add_tree2]
  outputs: [activ1]
activ1:
  type: "sigmoid"
  size: 16
  precision: 16
  number: 1
  inputs: [dsp_ch1]
  outputs: [sarray2,dbram2]
```

(b) YAML file snippet

Figure 6.1: Proxy benchmark generation

Since the YAML input file specifies components connected to each other regardless of the number of output and input bits of each component, the generator inserts some interface logic between the component instances. For example, if the YAML file specifies that component A with 40 output bits feeds component B that has 20 input bits, some interface logic needs to be generated that can enable connecting 40 signals to 20 signals. There are three cases that can arise:

1. The input bits are equal to the output bits and can be directly connected.
2. The input bits are less than the output bits. In this case, input bits are fanned out to match the number of outputs.
3. The input bits are greater than output bits. In this case, a reduction of bits is performed by inserting reduction trees of logical operations (e.g. xor/and/or gates, 2:1 multiplexers). The user can specify the mix of

gates and multiplexers to be used via command line options ("Interface options" in Fig. 6.1a).

The interface logic always adds a register stage between components. Note that these choices are also governed by circuit properties observed in real world designs; deep pipelines, high fanouts and reduction trees are common in DL designs. The proxy generator also generates statistics, such as the number of I/Os, the expected number of logic blocks, DSPs and BRAMs used by the generated benchmark. Comparing these numbers with the numbers obtained after running the generated benchmark through the VTR flow can be useful for verifying the validity of the generated benchmark.

For the Koios suite, 8 proxy benchmarks are generated. They have varying sizes ($14 - 43K$ netlist primitives) and contain different mixes of components from the module library. The generator and the YAML files of the 8 benchmarks are open-sourced, so a user can generate more designs, if required.

6.1.5 Enhancements to the VTR Flow

The VTR flow has traditionally been using Odin II [63] as its synthesis front-end. Some of the benchmark circuits could not be synthesized using Odin II as it only supports a subset of the Verilog-2005 standard. Therefore, support for some of the commonly used Verilog constructs was added to Odin II and some benchmarks were re-written using only the supported subset of the Verilog-2005 standard. However, this was a very tedious and labor-intensive process that restricted the extension of the Koios suite to include more bench-

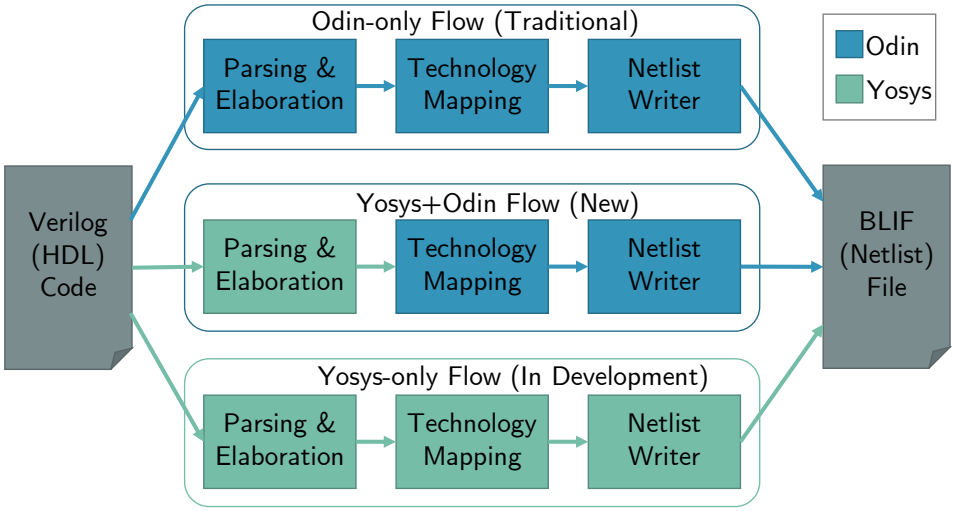


Figure 6.2: Various synthesis front-ends supported by VTR

marks that were written in SystemVerilog or used unsupported Verilog syntax. Hence, a new effort was undertaken to improve the language coverage of the VTR synthesis front-end using a combination of the Yosys synthesis tool and Odin II [29].

Yosys is an open-source synthesis engine with extensive Verilog-2005 and SystemVerilog support [129]. Whereas most commercial synthesis tools are closed source, Yosys offers a flexible and open-interface synthesis process which is valuable for developing new and customized synthesis algorithms. However, Yosys is totally agnostic to the target FPGA architecture and thus limits opportunities for architecture-aware logic inference (i.e. automatically inferring logic that can be mapped to hard blocks). Therefore, in this newly developed Yosys+Odin flow, Yosys provides better language coverage support and performs HDL elaboration followed by coarse-grained optimizations. After

that, Odin II performs partial technology mapping based on the target FPGA architecture using a mix of genetic algorithms [30] and trade-off analysis of hard vs. soft logic inference [71], and then writes out the final netlist as illustrated in Fig. 6.2.

The Koios suite includes a variety of benchmarks that use Verilog constructs that are not supported by Odin II, and therefore was used for developing and testing the new hybrid synthesis front-end. During this process, the Koios benchmarks helped identify and fix several issues such as:

- Unlike the Odin-only flow, the new hybrid front-end produced netlists with randomly-generated net names that cannot be traced back to their HDL declarations, which made debugging CAD flow errors significantly harder.
- Yosys used different names for the clock and reset signals of netlist atoms that can be packed into the same hard block (e.g. two multiplications to the same DSP block) which prevented VPR from packing them together even though they are actually connected to the same clock and reset signals, leading to higher resource utilization compared to the Odin-only flow.
- The new hybrid front-end generated a single black-box module definition for each of the hard block models specified in the VTR architecture description file. In some cases when the hard block has different modes of operation with different interface widths, this would result in technology mapping failures.

More recently, a modified version of Yosys, adapted from the Verilog-to-Bitstream tool [49], is also being integrated into the VTR flow to provide the option of performing all the synthesis steps solely using Yosys as shown in Fig. 6.2. However, this flow is not currently used for the Koios benchmarks.

While developing them, the Koios benchmarks were useful in exercising different parts of the tool flow that were not extensively tested before and identifying subtle bugs/issues in them. For example, multiple of the Koios circuits make heavy use of cascaded chains of multiple DSP blocks in a column to implement efficient dot product operations for DL. This presents additional placement constraints since the DSP blocks in a chain have to be initially placed and then moved around during placement optimizations as a single combined molecule to maintain the placement legality. Since VPR picks the device grid size based on the number of required blocks of each type without considering these additional placement constraints for cascaded DSPs, some of the Koios benchmarks (`tpu_like.small.os`, `dla_like.large` and `bwave_like.fixed.large`) were failing at the initial placement stage due to the absence of a legal solution given the predetermined device grid size. For example, a design can have a cascaded chain of N DSP blocks left to place and the device still has enough DSP blocks available but split across different columns (i.e. not in consecutive locations along a column). This results in a failure since no legal solution exists at this device grid size which was decided earlier in the flow based solely on the number of required blocks. A straightforward workaround is to manually specify a slightly bigger grid size

for the failing benchmarks or a maximum resource utilization target for VPR when automatically sizing the device grid (the latter workaround is currently used for running the failing Koios benchmarks). It can also be fixed by iteratively increasing the grid size during initial placement in such cases until a legal solution is found. This issue was flagged and its suggested solution will be implemented by the VPR team in a future release.

6.1.6 Availability and Usage

The Koios benchmarks are available at: `https://tinyurl.com/vtrkoios`. They have been tested and work out-of-the-box with the VTR flow. Scripts to automatically run and generate QoR (Quality of Results) for these benchmarks are also provided. In addition, the Titan flow [86] is used to generate the netlist (BLIF) files of the Koios benchmarks for the Stratix-IV FPGA architecture. These netlists can be used to directly run placement and routing using VPR without the need of an Intel Quartus license for running the synthesis front-end in the Titan flow. The BLIF files can be downloaded separately from this link: `https://tinyurl.com/koiosblif`.

These benchmarks are implemented and curated in this suite to be used for FPGA architecture exploration and CAD tool optimization. They aim to accurately capture all the different circuit structures and compositions, but should not be expected to be deployed as standalone functional designs. These circuits are structurally correct, and their high-level functionality have been verified. However, full functional verification on many different test cases

is considered beyond the scope of this dissertation.

6.2 Benchmark Results

6.2.1 Experimental Setup

The latest version of VTR is used for all experiments. When running VTR, an SDC (Synopsys Design Constraints) file is provided in which the target clock period is set to 0 (i.e. VTR will optimize the design for maximum clock frequency). Timing analysis for paths to/from the FPGA IOs is disabled; only register-to-register paths are analyzed. Unless stated otherwise, VTR is run with auto layout enabled (meaning the grid size expands based on the resources required by the design), the default timing-driven routing option with a maximum of 150 routing iterations, and a fixed channel width of 300 wires. All reported results are the average of three runs with different seeds. For experiments where VTR flow runtime and peak memory usage is reported, an Intel Xeon CPU E5-2430 running at 2.5 GHz with 64 GB of memory is used.

6.2.2 FPGA Architecture Used

A new FPGA architecture description file is developed to capture some relevant features of modern FPGAs. This architecture description file is also open sourced along with the benchmark suite. The delays and areas of all the FPGA blocks, including the DSP tiles, are obtained from COFFE [136] using a 22 nm technology node from PTM [117]. The circuits in this architecture are optimized for area-delay product, which leads to relatively higher delays compared to performance-optimized commercial FPGAs such as the Arria 10 family. Figure 6.3 shows a representation of this architecture, which

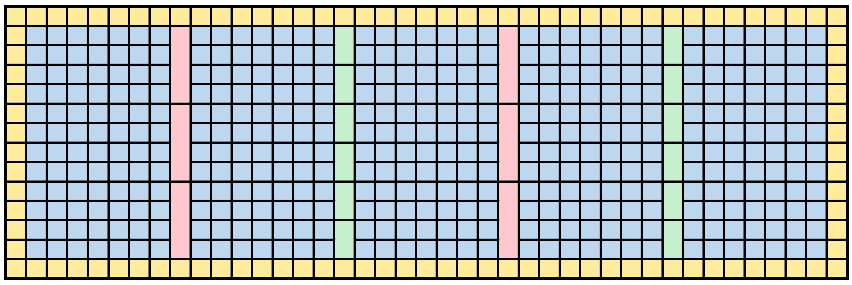


Figure 6.3: FPGA architecture (not to scale) used for experimenting with Koios benchmarks. Blue = Logic Block, Green = Block RAM, Red = DSP Slice, Yellow = Input Output Block

is described in the rest of this subsection.

6.2.2.1 Floorplan

The FPGA contains columns of logic blocks, DSPs and BRAMs. Both DSP and BRAM columns repeat every 16 columns and are interleaved such that every 8th column is a DSP or a BRAM. The DSP and BRAM tiles are 4 and 2 rows high respectively, and the IO pads are arranged along the perimeter of the FPGA.

6.2.2.2 Routing Architecture

The architecture uses unidirectional routing with wire segments of length 4 (260 out of 300 wires) and length 16 (40 out of 300 wires). The length 16 wires do not directly connect to block pins and are only accessible from the length 4 wires. Switches appear after every 4 blocks on the length 16 wires. The switch blocks use a custom switching pattern based on the Stratix-IV-like architecture used in the Titan flow [86]. The input and output flexibility of

connection blocks are set to 0.15 and 0.1, respectively.

6.2.2.3 Logic Blocks

Each logic block (LB) contains 10 basic logic elements (BLEs) similar to that in the Intel Stratix-10-like architecture from [33]. Each block has 60 input pins, 40 output pins, and a 50% sparsely populated local input crossbar. Each BLE has a 6-input LUT which can be fractured into two 5-input LUTs. The BLE also has 2 flip-flops and 2 bits of arithmetic with dedicated carry chains between LBs. Each BLE has 8 inputs and 4 optionally registered outputs.

6.2.2.4 DSP Slices

This architecture has a complex DSP block that supports most of the operating modes in the state-of-the-art Intel Agilex DSP block [53]. Multiple fixed point (9x9, 18x19, 27x27) and floating point (IEEE 32-bit (`fp32`), IEEE 16-bit (`fp16`) and Brain floating point (`bfloat16`)) precisions are supported. In addition, the DSP block has dedicated output chains for cascading several DSP blocks in the same column for efficient dot product structures.

6.2.2.5 BRAMs

BRAM blocks have a capacity of 20 Kb and have registered inputs and outputs. True and simple dual port modes are supported. In the simple dual port mode, a BRAM can be configured as: 512×40b, 1024×20b and 2048×10b, while the widest mode is not supported in true dual port mode. The delays

and areas of a BRAM block are obtained by interpolation between the values obtained from COFFE for 16 Kb and 32 Kb BRAMs.

Some benchmarks in Koios use advanced DSP features that are available in the FPGA architecture described above. This is done by instantiating DSP hard macros directly into the RTL when implementing natively-supported `fp16` multiplications or DSP cascaded structures for example. Similarly, BRAMs are also instantiated as hard macros in the RTL. Although these hard macros are architecture-specific, users can still use the Koios benchmarks with other FPGA architectures by replacing these RTL instantiations with their alternatives. To improve the usability of the Koios benchmarks, the same functionality of the architecture-specific hard macros is also implemented using behavioral Verilog. This allows users to switch between the hard macro and behavioral implementations using pre-processor directives (i.e. `ifdef`s). By disabling the `complex_dsp` and `hard_mem` directives, the benchmarks become completely architecture-agnostic and can be used with any FPGA architecture description file. In this case, the synthesis tool infers the hard blocks to be used and generates a netlist containing hard macro instances available in the user's FPGA architecture. If no hard blocks are available in the FPGA, the code will just be mapped to FPGA soft logic. The benchmarks have been verified to run without these directives for the FPGA architecture described in this section and the VTR flagship architecture as well.

This makes the Koios benchmarks also suitable for evaluating the addition of new hard blocks to an FPGA architecture, similar to some recent

DL-optimized FPGAs [75, 7]. To perform such studies, users can either: (1) modify the synthesis engine to automatically extract specific patterns from the Verilog designs and map them to the new blocks, or (2) modify the benchmarks to instantiate these new blocks as hard macros (defined in the VTR architecture file).

6.2.3 Results of the Koios Benchmarks

Table 6.3 shows the VTR results for the Koios benchmarks when running them with the FPGA architecture described in Section 6.2.2. The results show that these designs, with sizes ranging from 12K to 1.6M netlist primitives, are deeply pipelined with 27 out of the 40 benchmarks having critical paths with 6 or less logic levels on them. The benchmarks are also highly diverse in heterogeneity, with varying circuit compositions between soft logic, DSPs, and BRAMs. For example, some designs do not utilize any BRAMs since they either implement only the workload datapath (e.g. `gemm_layer` and `softmax`) or use distributed registers for storage (e.g. `bnn`). On the other hand, there are other BRAM-intensive designs such as `tdarknet_like.large` with $4,400$ BRAMs utilized. Similarly, with DSPs, there are some designs that use very few or no DSPs (e.g. `conv_layer_hls` and `reduction_layer`) as they mostly implement other non-multiplication operations in DL workloads such as pop-count or max/min/add reduction. Other designs are DSP-intensive (e.g. `deepfreeze.style2`) with over $1,700$ DSP blocks. Table 6.3 also shows that different types of resources are the grid size limiting factor for different

Table 6.3: VTR results of the Koios benchmarks

Benchmark	Netlist Primitives	Logic Depth	Used IOs	Used LBs	Used DSPs	Used BRAMs	Max. Freq.	Routed Wirelength	Elapsed Time	Peak Memory
dla_like.large	1608867	5	819	28201	**1376**	864	107.4	11445	1140.0	15733.5
clstm_like.large	1083855	3	1518	26341	**961**	739	105.6	5785	842.7	12901.0
deepfreeze.style3	759656	3	540	18499	340	**3489**	116.3	5380	289.3	16131.7
clstm_like.medium	743071	3	1230	17854	**661**	498	113.9	3767	400.4	8805.7
deepfreeze.style1	687669	3	540	15115	700	**1999**	135.2	4673	243.0	10172.1
dla_like.medium	600492	5	411	10656	**400**	312	140.6	2920	209.0	5408.4
deepfreeze.style2	470421	3	540	12896	**1762**	1387	62.6	3466	246.3	15574.3
proxy.2	439725	8	574	8921	330	**1099**	130.9	3293	228.9	5796.2
clstm_like.small	402331	3	942	9396	**361**	257	131.3	1821	100.8	4739.3
tdarknet_like.large	391291	5	46	13574	367	**4400**	72.7	4173	775.4	18456.7
proxy.4	391195	7	2392	7768	**757**	1189	101.3	4510	401.9	7439.1
proxy.1	358143	7	1113	5989	**1037**	619	125.3	4325	206.8	9503.0
bwave_like.float.large	310527	6	1093	9699	**640**	1182	93.9	4440	114.5	6522.4
proxy.3	304125	10	1036	9585	107	**847**	96.8	2491	124.7	4569.3
dla_like.small	260199	5	207	4799	**128**	132	160.7	998	59.3	2143.0
proxy.7	248950	7	498	4937	**302**	492	114.2	2167	135.9	3214.4
lstm	247060	7	2677	5060	**610**	305	121.8	2129	272.2	5767.3
proxy.6	206539	3	1025	3403	**300**	406	134.7	1720	174.3	3053.7
bnn	204601	3	382	**5694**	63	0	131.0	1184	17.2	2171.0
lenet	190809	34	140	7417	**497**	820	53.9	3250	671.4	5850.0
dnnweaver	189706	6	3531	5552	**288**	**1139**	82.4	2921	49.7	5258.4
tdarknet_like.small	157431	6	46	6974	**90**	**3978**	63.8	2657	217.4	16043.7
proxy.8	150264	7	1002	3047	**367**	378	110.9	1266	67.2	3325.1
proxy.5	147618	7	785	3199	**283**	236	108.1	1227	70.2	2768.5
bwave_like.float.small	84893	6	200	2625	144	**358**	129.1	936	14.2	1802.7
tpu_like.large.ws	78335	8	1190	3011	**1066**	116	100.2	961	87.9	8848.8
tpu_like.large.os	70946	5	1188	1596	**1064**	64	120.4	2028	95.4	8826.8
gemm_layer	64765	4	1779	2001	**200**	0	173.9	789	17.6	1897.6
bwave_like.fixed.large	54871	6	328	1299	**562**	511	104.2	1816	32.4	5938.8
attention_layer	54865	7	1089	1455	**137**	194	124.5	480	18.6	1328.6
conv_layer	37268	4	156	938	**42**	56	218.6	245	6.5	562.0
robot_rl	30529	6	387	1285	18	**96**	148.8	232	6.0	522.9
tpu_like.small.ws	27097	7	646	1034	**278**	58	118.8	288	15.1	2407.2
tpu_like.small.os	21962	5	644	538	**276**	32	156.7	416	13.9	2381.5
reduction_layer	18323	6	54	805	0	**52**	147.4	183	1.9	340.2
spmv	17734	6	99	503	32	**232**	178.4	221	4.0	946.1
bwave_like.fixed.small	16632	5	198	404	**139**	170	132.7	397	5.2	1293.1
eltwise_layer	16187	4	249	355	**50**	72	249.1	193	2.6	472.8
softmax	13177	10	552	512	**53**	0	114.6	126	2.3	492.1
conv_layer_hls	12097	3	**3299**	1717	12	21	151.1	102	12.2	3983.8

Frequency is in MHz, Routed Wirelength is 1000 length-1 segments, Elapsed Time is in minutes, and Peak Memory is in MBs.

benchmarks in Koios. The majority of the designs are bound by hard blocks, as indicated by the bold entries in the table, which emphasizes that these benchmarks can be useful for exploring new DSP and BRAM architectures.

Most of the designs in the Koios suite can achieve reasonably high operating frequencies up to 249 MHz and an average of 124 MHz. The FPGA architecture used for experiments is not very fast. The delays in the architecture are based on area-delay-optimized PTM models (with raw delays similar to 40 nm Stratix-IV). Changing the delays of FPGA resources to those typical of a high-speed ($\leq$14 nm) device would increase the frequency by >2×. The `lenet` design is a clear outlier with a frequency of 53.9 MHz. This design is generated by HLS and has a very high logic depth of 34. The total routed wirelength of the benchmarks are largely correlated with the circuit size and ranges from 102K up to 11.4M units of length 1 wire segments.

The top graph in Fig. 6.4 plots the VTR flow runtime for each of the Koios benchmarks. The trendline shows that the runtime grows almost linearly with the number of netlist primitives in the circuits. There are some notable exceptions; `lenet` and `tdarknet` designs have very high runtime for their number of netlist primitives. Also, looking at the components of runtime, in most benchmarks, ABC (the tool that performs logic optimization and techmapping in the VTR flow) takes more time compared to Odin/Yosys and VPR (the tool that performs packing, placement and routing in the VTR flow). The bottom graph in Fig. 6.4 plots the VTR flow peak memory usage for the Koios benchmarks. The trendline shows a sub-linear growth in peak

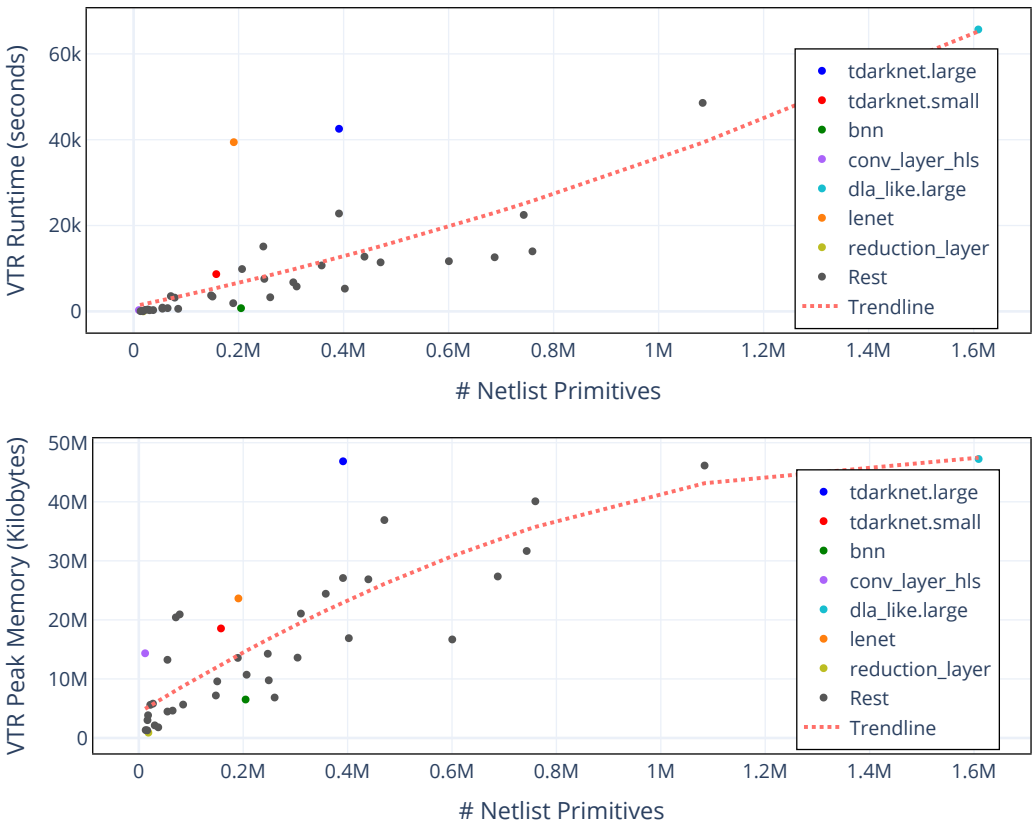
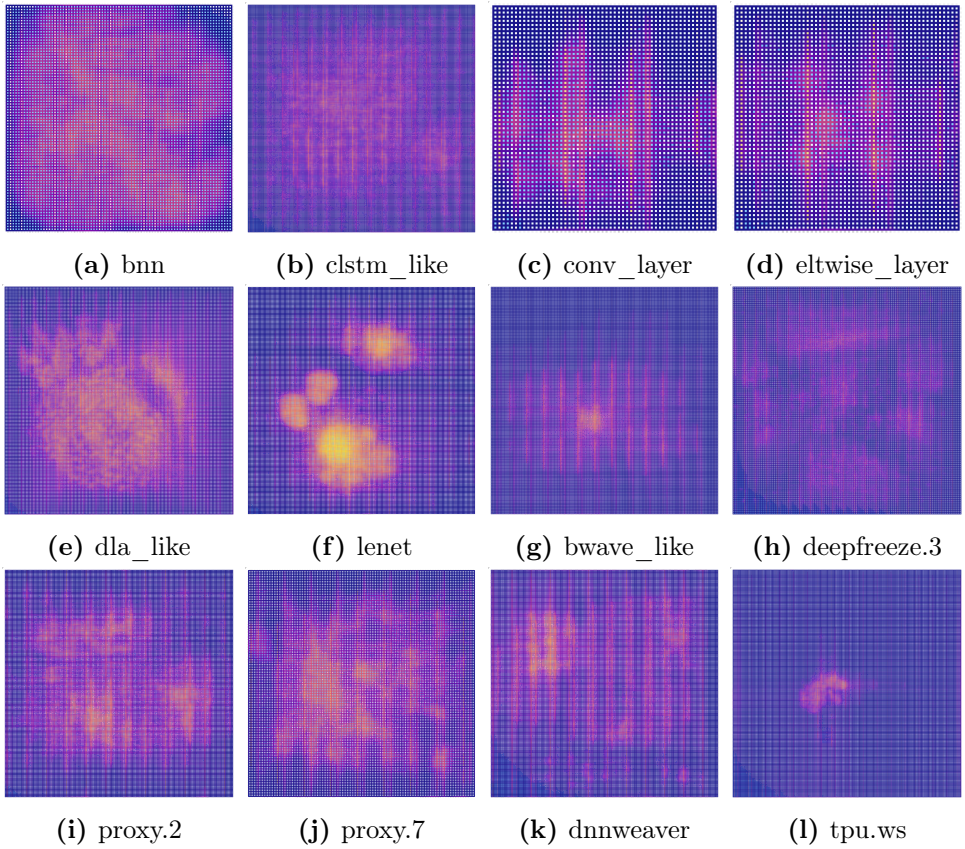


Figure 6.4: VTR runtime (top) and peak memory usage (bottom) for the Koios benchmarks

memory requirement as the number of netlist primitives increases. The `lenet` and `tdarknet` designs again have very high memory usage for their size, and VPR consumes the majority of used memory compared to Odin/Yosys and ABC.

The routing heat maps for some of the Koios benchmarks are shown in Fig. 6.5, where the lighter color correspond to higher routing congestion. The routing heat maps look very different for different designs; this highlights



(a) bnn **(b)** clstm_like **(c)** conv_layer **(d)** eltwise_layer

(e) dla_like **(f)** lenet **(g)** bwave_like **(h)** deepfreeze.3

(i) proxy.2 **(j)** proxy.7 **(k)** dnnweaver **(l)** tpu.ws

Figure 6.5: Routing utilization heatmaps for some Koios benchmarks

the diversity in routing requirements and patterns of the benchmarks, which exercises the placement and routing algorithms in different ways. Some benchmarks have a very regular pattern (e.g. `bnn`), which implies heavy usage of LBs (soft logic). In other benchmarks, high routing congestion is seen along columns of hard blocks (e.g. `dnnweaver`).

6.2.4 Statistical Analysis

To perform statistical analysis on the Koios suite, a large number of metrics (∼75 metrics) are collected for all benchmarks. Some metrics other than those in Table 6.3 are logic depth, max non-global fanout, average wire segments per net, max routing channel utilization, number of near critical connections, number of blocks before and after clustering, and the maximum number of wire segments used by a net. However, it is difficult to manually investigate the data and conduct meaningful analysis. Hence, principal component analysis (PCA) [31, 95] is performed on the collected data, which converts N variables into a smaller group of m linearly uncorrelated variables known as the *principal components (PCs)*. $m = 4$ is used in the evaluation. Each PC is a linear combination of different features or variables with a certain weight. The first PC covers the majority of the variance, and subsequent PCs cover diminishing variances. By eliminating components with lower variance values, the dimensionality of the data set can be reduced. Benchmark similarity is examined by hierarchically clustering them. The Euclidean distance of various metrics (or variables) is used to calculate how similar two benchmarks are. The output of this clustering can be displayed as a tree or dendrogram in which smaller linkage distance between two benchmarks indicates higher similarity between their metrics.

Fig. 6.6 shows the dendrogram plot for the Koios benchmarks. The x-axis shows the linkage distance between the different benchmarks on the y-axis. The absolute value of the distance does not matter, but the relative

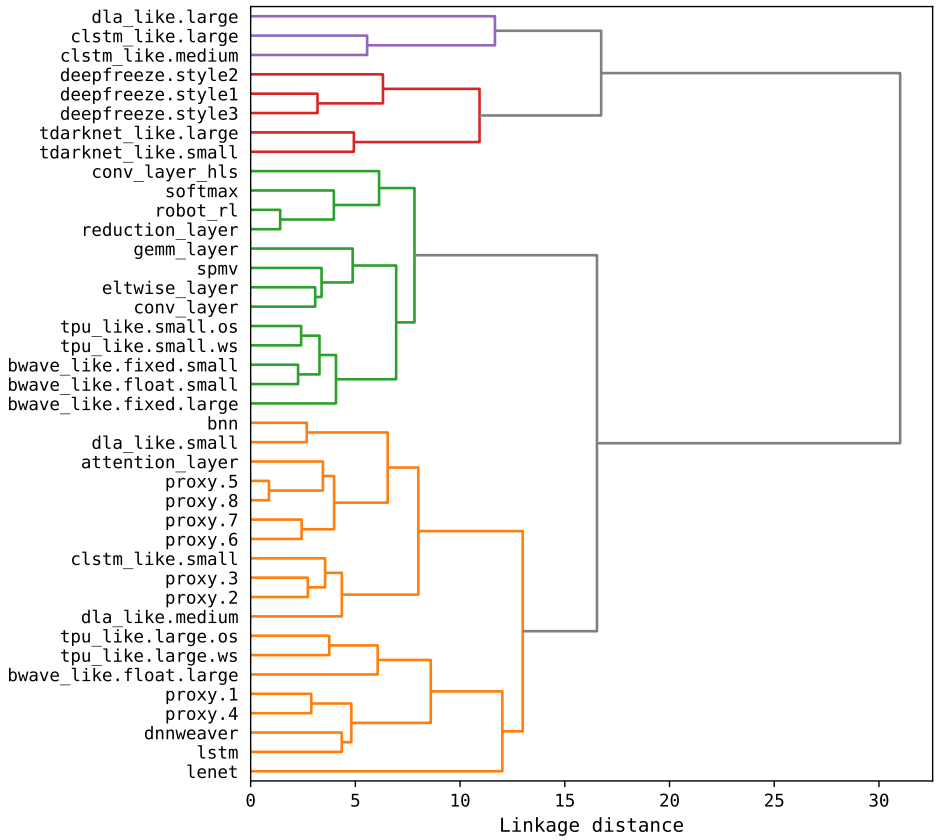


Figure 6.6: Dendrogram showing similarity between Koios benchmarks

value between benchmarks does, and the ordering of benchmarks on the y-axis has no special significance. There is no one benchmark that is particularly unique. If a vertical line is drawn at linkage distance = 15 (for example), the benchmark suite is divided into the 4 subsets shown in different colors. A user with limited compute resources or in early stage experiments can choose one or a few benchmarks from each subset to get the maximum coverage for their experiments. As another example, this analysis shows that among the 8 proxy benchmarks, {1,4}, {2,3} and {5,6,7,8} are three groups that have very

similar characteristics across the circuits in each of them. This means that a user could choose one proxy benchmark from each group as a representative benchmark in case of limited resources/time.

Fig. 6.7a shows a scatter plot of all the Koios benchmarks based on the first two PCs covering 65% of the variance (50% in PC1 and 15% in PC2), and Fig. 6.7b shows a similar plot based on the next two PCs of the metrics covering another 14% of the variance (7.5% in PC3 and 6.5% in PC4). The PCA analysis provides coefficients for each of the metrics to identify the main contributors to each PC. PC1 is mainly dominated by metrics related to the size (netlist primitives, CLB usage, routing wirelength, runtime). PC2 is dominated by average net length, near critical connections, device size and frequency. PC3 is dominated by the logic depth and maximum routing channel utilization. PC4 is dominated by max non-global fanout and max net length. The 4 benchmarks at the extreme opposites of PC1 and PC2 in Fig. 6.7a are `tdarknet_like.small`, `eltwise_layer`, `bnn` and `dla_like.large`. These 4 benchmarks belong to different groups from the dendrogram in Fig. 6.6. The proxy benchmarks appear towards the center of the scatter plots, implying that they represent the common benchmarks of the suite. Thus, if a user is constrained on resources, a representative subset of the benchmark suite could be the 4 extreme benchmarks and one or more of the proxy benchmarks for example. In Fig. 6.7b, the outliers are `clstm_like.large`, `clstm_like.medium` and `lenet`. The `lenet` benchmark (on the far right) has an abnormally large logic depth of 34, while the `clstm_like.large` and `clstm_like.medium` have

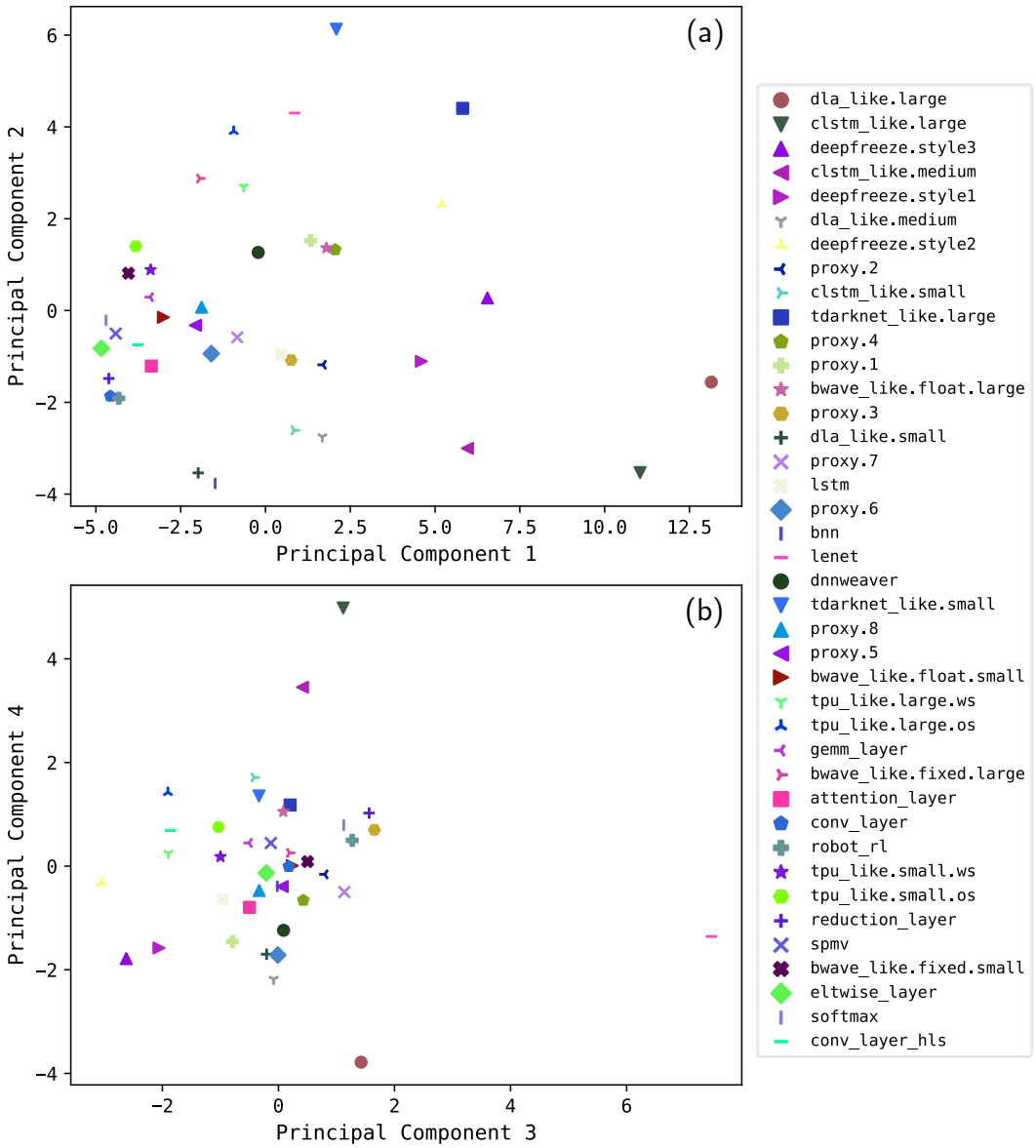


Figure 6.7: Analyzing the Koios benchmarks using PCA

much higher numbers of non-global fanouts.

6.3 Comparison to Other Benchmark Suites

6.3.1 Methodology

In this section, various properties of Koios benchmarks are compared to those of other existing non-DL-targeted benchmarks that are commonly used to drive FPGA architecture and CAD research. The most relevant suite for comparison is the VTR benchmark suite, because these are compatible with the same fully open source VTR flow. Other existing suites are either too small and do not represent realistic modern use cases of FPGAs or depend partially on commercial CAD tools. For this comparison, only the VTR benchmarks with more than $10,000$ netlist primitives (9 benchmarks) are used. This is a common practice in CAD-related studies [34]. Smaller designs are not representative of realistic benchmarks and they cannot be used to derive any reliable conclusions. The same VTR settings and architecture file as in Section 6.2 is used.

In addition, the Koios benchmarks are also compared to the Titan23 benchmarks [86]. The Titan benchmarks are not compatible with the fully open source VTR flow and depend on the Intel Quartus tool to perform logic synthesis and generate netlist BLIF files. Therefore, they can only be placed and routed using the Stratix-IV-like architecture capture in VTR, which limits their usability for FPGA architecture studies. However, they are commonly used as large representative benchmarks for FPGA CAD research and for evaluating QoR of different CAD algorithms/flows. For this comparison, the BLIF netlists of the Titan benchmarks provided in the v1.3.1 release of Titan

are run through VPR and the Koios HDL benchmarks through the end-to-end VTR flow. The same VPR settings from the official VTR Titan regression tests are used for running both sets of benchmarks.

Finally, a QoR comparison between VPR and Quartus using the Koios benchmarks implemented on the Stratix-IV FPGA architecture is also presented, since this is currently the only Intel FPGA architecture with a corresponding architecture capture in VTR. In this experiment, Intel Quartus Prime 20.1 is run using the default compiler effort settings (i.e. `STANDARD_FIT` mode). For a fair comparison to VPR with auto layout, the Stratix-IV device in Quartus is set to `AUTO` which automatically selects the smallest Stratix-IV device that can fit the given design. On the other hand, the Koios benchmarks are synthesized for Stratix IV using Intel Quartus Prime 20.1 and then the `vqm2blif` tool from the Titan flow is used to generate Koios BLIF netlists. Then, they are run through VPR with the same settings used for evaluating QoR in [85]. The placement `inner_num` is set to 1.0, the router `astar_fac` is set to 1.0, and the number of router iterations is set to 400. Both Quartus and VPR are also given equivalent timing constraints with an aggressive 1ns clock period target and paths to/from external IOs constrained on a virtual IO clock as in [85].

6.3.2 Comparison to the VTR Benchmarks

Fig. 6.8a shows a scatter plot of the DSP and BRAM to LB ratios for both the Koios (red) and VTR (blue) benchmarks as metrics for their DSP

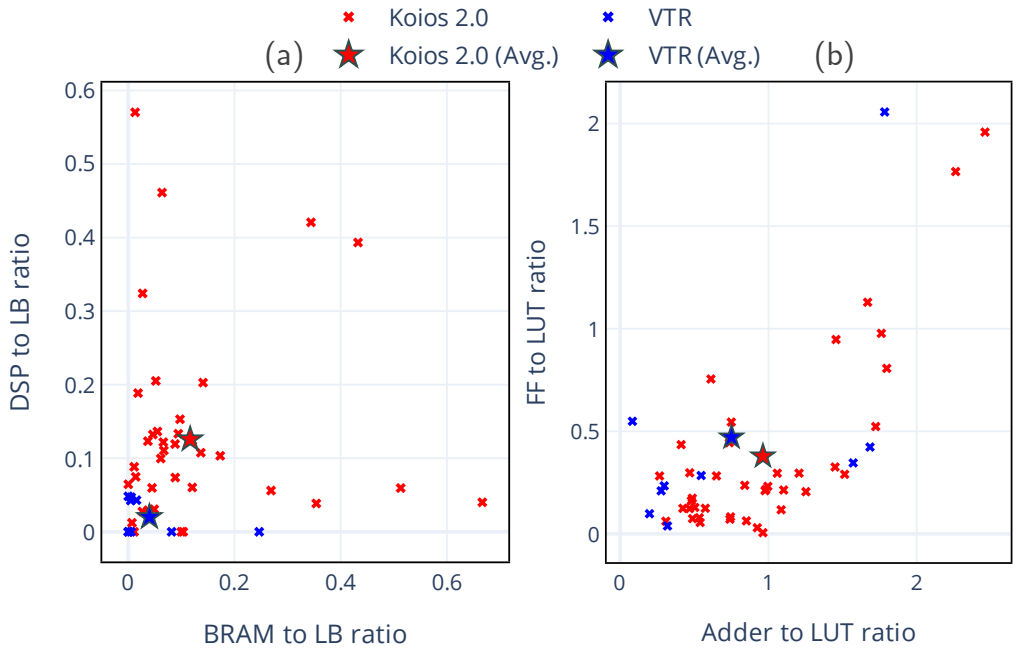


Figure 6.8: Comparing circuit compositions of Koios & VTR benchmarks: (a) DSP/BRAM to LB and (b) FF/adder to LUT ratios

and memory density. The individual ratios for each of the benchmarks are shown by (×) symbols, while the average across the whole benchmark suite is marked by the stars. The figure shows that, on average, the Koios benchmarks are more DSP and memory rich than the VTR benchmarks; it has 2.9× and 6.2× higher DSP to LB and BRAM to LB ratios, respectively. The individual benchmarks of the Koios suite are also more scattered and varying across the spectrum of DSP and BRAM compositions. More importantly, it shows that most of the VTR benchmarks have very low DSP and BRAM densities (except for the only `stereovision2` outlier circuit), making them inadequate for evaluating any DSP or BRAM architecture modifications.

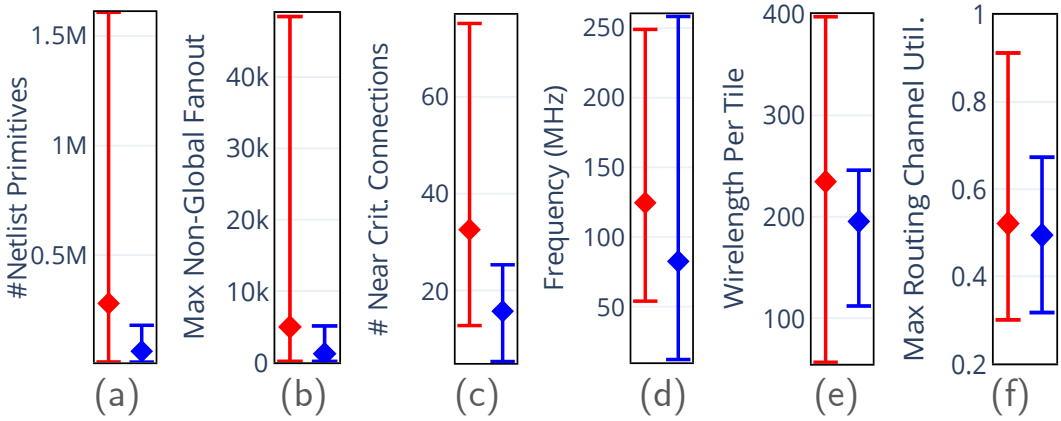


Figure 6.9: Averages and ranges of key metrics of Koios (Red) & VTR (Blue) suites.

Fig. 6.8b has a similar plot for FF and single-bit adder to LUT ratios. It shows that the Koios suite has 1.28× higher ratio between FFs and LUTs which reflects their deeply pipelined nature, and 20% lower adder to LUT ratio compared to the VTR suite. However, the average adder to LUT ratio of the VTR suite is significantly skewed by a single benchmark (`stereovision2`) which has 60,753 1-bit adders and only 29,541 LUTs. Excluding this outlier, the Koios benchmarks have a 1.4× higher average adder to LUT ratio.

Fig. 6.9 illustrates averages and ranges of key metrics for both the Koios and VTR benchmark suites. Fig. 6.9a-d show that the Koios benchmarks have 4.5× more netlist primitives, 4.9× larger non-global fanouts, 2.07× more near (top 10%) critical connections, and 1.5× higher frequencies on average compared to the VTR benchmarks. The Koios benchmarks are also scattered across a much wider range of values for each of those metrics. Fig. 6.9e shows that Koios circuits have 20% higher average routed wirelength per tile

compared to VTR benchmarks. Fig. 6.9f shows that the circuit with the highest max routing utilization in Koios has 35% higher utilization compared to the circuit with max routing utilization in VTR benchmarks. Koios designs also have an average of 6 logic levels on the critical path, compared to 30 levels for the VTR benchmarks. This reflects the deeply pipelined nature of these benchmarks which is a key property of modern FPGA designs.

6.3.3 Comparison to the Titan Benchmarks

Only 22 out of 23 Titan benchmarks could be successfully placed and routed. The largest circuit (`gaussianblur`) fails with runtime exceeding 4 days, and therefore is excluded from the comparison. On the other hand, some of the Koios benchmarks consume more resources than that available in the largest Stratix-IV device, and thus are excluded for a fair comparison (since the Titan benchmarks have to be synthesized through Quartus to a real Stratix-IV device). 6 out of the 40 Koios circuits are excluded for DSP/BRAM limitations and another 11 are excluded for IO limitations, leaving 23 Koios benchmarks valid for this comparison.

Both Titan and Koios suites are heterogeneous - they have a large number of DSPs and BRAMs. On average, the 23 Koios designs have 2.17× DSPs, 0.66× memory bits and 0.51× routed wirelength, compared to the 22 Titan designs. Fig. 6.10a shows that the Koios benchmarks are smaller; there are 2.06× more netlist primitives on average in Titan benchmarks. Koios benchmarks have 3× lower max non-global fanout (Fig. 6.10b) than Titan

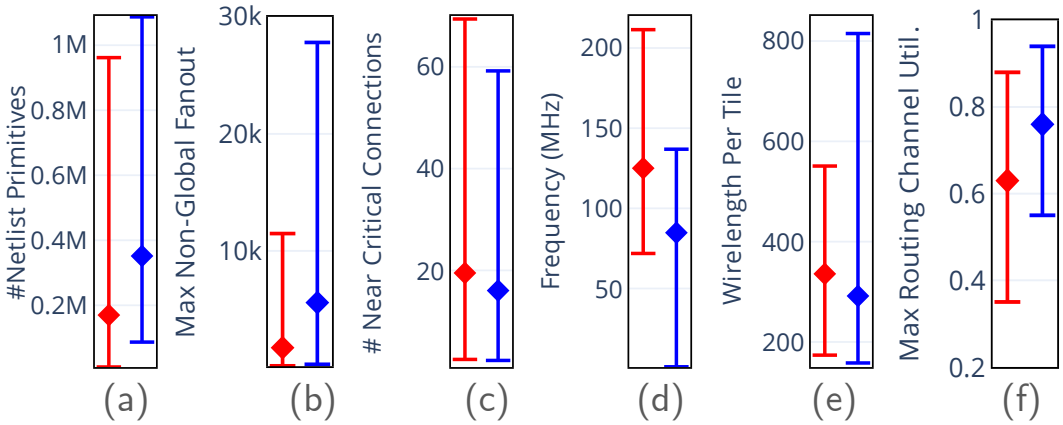


Figure 6.10: Averages and ranges of key metrics of Koios (Red) & Titan (Blue) suites.

benchmarks. However, Koios benchmarks have higher min, max and average number of near-critical connections, compared to Titan benchmarks, as seen in Fig. 6.10c. Koios benchmarks run at significantly faster frequency (Fig. 6.10d) compared to Titan benchmarks. The fastest Koios design runs at 1.55× higher frequency compared to the fastest Titan design. There are designs with very low frequency (minimum=1.1MHz) in the Titan suite. The average wirelength per tile (Fig. 6.10e) is 15% higher in Koios benchmarks. Fig. 6.10f shows that Koios benchmarks have a much wider range of maximum routing utilization, compared to Titan benchmarks, although the max routing utilization is high on average in Titan benchmarks. Overall, both suites pose challenging problems to CAD flows, but there are some peculiar characteristics of Koios benchmarks, like high frequency and heavy DSP usage, owing to them belonging to the DL domain.

6.3.4 QoR Comparison of VPR and Quartus

Table 6.4 presents the detailed QoR comparison of VPR and Quartus for a subset of the Koios benchmarks that could fit on Stratix-IV devices. For most of the benchmarks, VPR packed denser logic clusters, resulting in 33% less Logic Blocks (column 'LB') on average. Although denser logic clustering in VPR was previously reported in [86] and later reduced in [85] resulting in better critical path delays, the Koios benchmarks show a much bigger difference in logic packing density between VPR and Quartus compared to the 5% difference in [85] which uses an older version of Quartus. VPR also uses 42% more DSP Slices (column 'DSP') than Quartus, with some benchmarks (e.g. `bwave_like.fixed.large`) using up to 4×. The reason is that VPR, due to its generality, cannot efficiently map multiplication primitives to DSP blocks in its complex modes of operation. On the other hand, Quartus searches for specific patterns in the circuit netlists that can be efficiently mapped to the target device DSP blocks. For BRAMs, the results show that VPR rarely makes use of the bigger 144 Kb BRAMs (column 'M144K') which it uses in only one benchmark (`lenet`). In contrast, Quartus uses these bigger BRAMs in 7 other circuits as indicated by the BRAM counts in brackets in Table 6.4. For these benchmarks, VPR maps all logical memories to the smaller 9 Kb BRAMs (column 'M9K') resulting in a 1.9× higher utilization of these blocks when averaged across the 7 benchmarks, which translates to a 24% increase across the whole suite. VPR also results in 1.46× higher total routed wirelength (column 'WL') and 1.36× longer critical path delays (column 'CP') compared to Quar-

Table 6.4: VPR and Quartus QoR comparison on Koios. Numbers are ratios of VPR:Quartus results, '–' represents unutilized resource for both, and numbers in brackets are the absolute count of resources used by Quartus when VPR used none.

Benchmark	LBs	DSPs	M9Ks	M144K	WL	CP
attention_layer	0.67	0.69	1.07	–	0.73	1.03
bnn	0.84	–	–	–	0.54	1.33
bwave_like.fixed.large	0.86	4.00	1.00	–	2.23	1.58
bwave_like.fixed.small	0.37	4.00	1.00	–	2.18	1.59
bwave_like.float.large	0.76	3.00	1.00	–	1.23	1.02
bwave_like.float.small	0.53	2.96	1.00	–	2.62	1.29
conv_layer	0.53	0.89	1.00	–	1.41	1.7
dla_like.large	1.15	1.73	1.27	(24)	1.35	1.25
dla_like.medium	1.10	1.67	1.42	(12)	2.04	1.14
dla_like.small	0.95	0.76	1.58	(6)	1.54	1.04
eltwise_layer	0.57	0.50	1.00	–	1.30	1.36
lenet	0.80	0.85	0.77	1.00	1.37	1.55
proxy.1	0.89	3.00	1.00	–	1.06	1.14
proxy.2	0.68	1.23	1.87	(60)	2.52	2.24
proxy.3	0.52	0.92	2.10	(60)	1.54	1.15
proxy.5	0.46	0.93	2.99	(16)	1.33	1.64
proxy.7	0.66	0.88	2.77	(48)	1.36	1.44
reduction_layer	0.76	–	1.00	–	0.97	1.44
robot_rl	0.78	2.00	1.00	–	1.35	1.15
softmax	0.68	0.63	–	–	1.01	1.17
spmv	0.62	1.00	1.00	–	1.54	1.23
tpu_like.small.os	0.35	2.79	1.00	–	2.87	1.88
tpu_like.small.ws	0.50	1.89	1.00	–	2.16	1.55
Geomean	**0.67**	**1.42**	**1.24**		**1.46**	**1.36**

tus. These gaps are higher than the 1.26× higher total routed wirelength and 1.2× longer critical path delay reported in [85] on the less heterogeneous and less DSP-intensive Titan benchmarks. These bigger gaps can be attributed to the less efficient packing and mapping of hard blocks discussed above which are more heavily used in the Koios benchmarks, and also their deeply pipelined nature. This highlights the value of having more challenging benchmarks that can exercise the CAD tools in different ways.

6.4 Case Studies

Koios benchmarks are architecture-agnostic and do not depend on commercial tools for any portion of the FPGA CAD flow. Thus, they can be used to perform flexible FPGA architecture using the fully-open-source VTR flow. In this section, two example case studies to demonstrate this are presented.

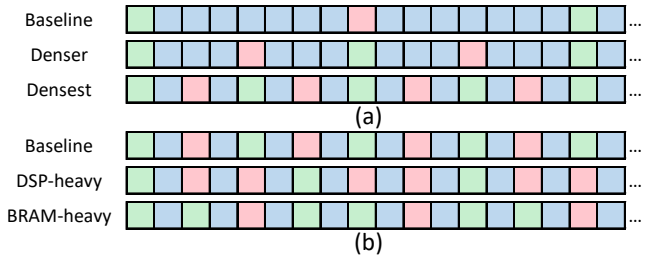


Figure 6.11: FPGA layouts for the architectures used in the case studies. Blue = Logic Block, Green = Block RAM, Red = DSP Slice

6.4.1 Case Study 1: Hard Blocks to Soft Logic Ratio

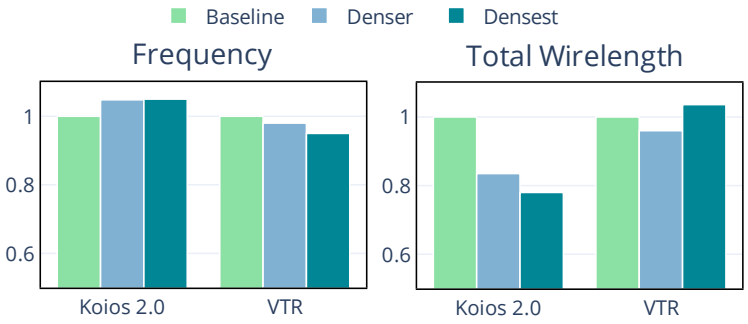


Figure 6.12: Effect of varying the density of DSPs and BRAMs on Koios and VTR benchmark suites

As shown in Table 6.3, Koios' DL-focused circuits are highly heterogeneous (i.e. DSP and BRAM intensive). Thus, in the first case study, the

density of these hard blocks with respect to soft logic is varied. Three different density levels are considered, as shown in Fig. 6.11a, with 1:7, 1:3, and 1:1 ratio between hard block and soft logic columns for the baseline, denser, and densest architecture variations, respectively. All three architecture variations are evaluated using both Koios and VTR benchmarks. Fig. 6.12 shows the geomean frequency and total routed wirelength for both suites. For the DL-oriented Koios benchmarks, the frequency increases and wirelength decreases as the density of hard blocks increases. Since these benchmarks heavily utilize these blocks, increasing their density in the FPGA grid brings them closer to each other, which in turn reduces the critical path delays and total length of used wires. The densest architecture variation results in a 5.2% increase in frequency and 22% reduction in total wirelength on average across all benchmarks in the Koios suite. For the VTR benchmarks, wirelength is slightly improved for the denser variation (4% lower), before getting worse for the densest architecture. The frequency degrades for both denser and densest architectures. These results show that a higher density of DSPs and BRAMs is favorable for building DL-optimized FPGAs, at the cost of a slight or no degradation in QoR for the general VTR benchmarks (in the densest and denser architecture variations respectively).

6.4.2 Case Study 2: DSP to BRAM Ratio

In the first case study, the ratio of hard blocks to soft logic is varied, while keeping a fixed 1:1 DSP to BRAM ratio. For the second case study, the

Table 6.5: Effect of varying the FPGA's DSP to BRAM ratio

Metric	Arch.	Geo-mean	DSP-heavy tpu_like(L)	BRAM-heavy tdarknet_like(L)
Freq.	**Baseline**	125.6	102.5	86.2
	DSP-heavy	124.9	**110.4**	92.0
	BRAM-heavy	126.2	106.0	**114.6**
WL	**Baseline**	1065K	749K	3105K
	DSP-heavy	1065K	**720K**	3343K
	BRAM-heavy	1098K	781K	**3106K**
Grid	**Baseline**	109×109	224×224	190×190
	DSP-heavy	110×110	**210×210**	232×232
	BRAM-heavy	111×111	228×228	**167×167**

Frequency is in MHz, Wirelength (WL) is in units of length 1 wires.

best architecture variation for DL benchmarks from the first case study (i.e. densest) is carried over. However, the DSP to BRAM ratio is varied between 2:1 and 1:2 to create DSP-heavy and BRAM-heavy variations respectively (in addition to the baseline with 1:1 ratio), as shown in Fig. 6.11b. Table 6.5 presents the results of this experiment. It shows the geomean frequency, routed wirelength, and FPGA grid size for the whole Koios suite, as well as the results for a DSP-intensive benchmark (medium `tpu_like.large.ws`) and a BRAM-intensive benchmark (`tdarknet_like.large`). The geomean results do not show a strong trend that clearly favors a specific architecture. However, the DSP-heavy `tpu_like.large.ws` design has 7.7% higher frequency and 4% lower wirelength when implemented on the DSP-heavy architecture compared to the baseline. Similarly, the BRAM-heavy `tdarknet_like.large` benchmark has 33% higher frequency and requires a 23% smaller device when implemented on the BRAM-heavy architecture compared to the baseline. This highlights that Koios strikes a good balance between different circuit compo-

sitions and can be reliably used for DL-optimized FPGA architecture exploration.

Chapter 7

Conclusion

7.1 Summary

This dissertation proposes optimizing the architecture of FPGAs to improve their performance and energy efficiency for Deep Learning acceleration. Tensor Slices are new DL-optimized blocks that are added to FPGAs. The Tensor Slice efficiently performs common operations used in today's neural networks like matrix-matrix multiplication, matrix-vector multiplication and element-wise matrix addition, subtraction and multiplication. Converting about 10% of the area of an Intel Agilex-like baseline FPGA to Tensor Slices increases the peak compute throughput (GigaMACs/sec) by $\sim$1.86$\times$ for 8-bit fixed point precision and $\sim$1.42$\times$ for 16-bit fixed-point, IEEE Half-Precision Floating Point (fp16) and Brain Floating Point (bf16) precisions. Adding Tensor Slices on the FPGA significantly benefits DL benchmarks in terms of metrics like frequency, area, routing wirelength, etc. On an FPGA architecture with Tensor Slices, 1.63$\times$ improvement in frequency and a 55% reduction in area and routing wirelength is observed, averaged across several DL benchmarks, compared to an Intel Agilex-like baseline FPGA. The impact of adding Tensor slices to an FPGA on non-DL applications is also studied. A reduction of 2.3% in frequency and an increase of 7.7% in routing wirelength is observed

on the FPGA with the most amount of area (30%) spent on Tensor Slices, compared to the baseline FPGA, averaged across several non-DL benchmarks. Replacing DSP Slices on an FPGA with Tensor Slices, an average speedup of 2.3× is obtained on common DNNs.

Additionally, this dissertation proposes converting BRAMs on FPGAs to CoMeFa RAMs. CoMeFa RAMs utilize the true dual-port nature of FPGA BRAMs and contain multiple configurable single-bit bit-serial processing elements. CoMeFa RAMs can be used to compute with any precision, which is extremely important for applications like Deep Learning (DL). Adding CoMeFa RAMs to FPGAs significantly increases their compute density, while also reducing data movement. Two architectures of these RAMs are presented: CoMeFa-D (optimized for delay) and CoMeFa-A (optimized for area). Compared to prior works, CoMeFa RAMs do not require changing the underlying SRAM technology like simultaneously activating multiple wordlines on the same port, and are practical to implement. CoMeFa RAMs are especially suitable for parallel and compute-intensive applications like DL; these versatile blocks also find applications in diverse applications like signal processing, databases, etc. By augmenting an Intel Arria-10-like FPGA with CoMeFa-D (CoMeFa-A) RAMs at the cost of 3.8% (1.2%) area, and with algorithmic improvements and efficient mapping, a geomean speedup of 2.55x (1.85x) is observed across microbenchmarks from various applications and a geomean speedup of up to 2.5x is seen across multiple Deep Neural Networks.

Compute throughput ratio between ASICs and traditional FPGAs is

$\sim$ 24:1 [22]. Based on the studies presented in this dissertation, the compute throughput of a DL-optimized FPGA containing both Tensor Slices and CoMeFa RAMs is 5$\times$ compared to a traditional FPGA. Hence, a DL-optimized FPGA containing both Tensor Slices and CoMeFa RAMs reduces the performance gap with ASICs significantly.

In the process of exploring FPGA architecture for Deep Learning, benchmark circuits are an essential component; the QoR achieved on a set of benchmarks is the main driver for architecture and CAD design choices. However, current academic benchmark suites are inadequate, as they do not capture any designs from the DL domain. This dissertation presents a suite of DL acceleration benchmark circuits for FPGA architecture and CAD research, called Koios. This suite of 40 circuits covers a wide variety of accelerated neural networks, design sizes, implementation styles, abstraction levels, and numerical precisions. These benchmarks include 32 DL designs and 8 synthetic (proxy) benchmarks. The Koios benchmarks are larger, more data parallel, more heterogeneous, more deeply pipelined, and utilize more FPGA architectural features compared to existing open-source benchmarks. This enables researchers to pinpoint architectural inefficiencies for this class of workloads and optimize CAD tools on more representative benchmarks that stress the CAD algorithms in different ways. In this dissertation, the Koios designs are described, their characteristics are compared to prior FPGA benchmark suites, and results of running them through the Verilog-to-Routing (VTR) flow using a recent FPGA architecture model are presented. Finally, case studies show-

ing how exploration of DL-optimized FPGA architecture and CAD algorithms can be performed using this new benchmark suite are presented.

DL-optimized FPGAs can reduce the gap between ASICs and FPGAs for DL acceleration. With the abundance of DL applications, making DL-optimized FPGAs is an attractive proposition.

7.2 Future Work

There are a number of future research directions that can be taken to further improve FPGA architecture for Deep Learning, or to improve the usability of the FPGA architecture proposed in this dissertation.

Currently, to use Tensor Slices, a user has to manually instantiate a Tensor Slice block in the RTL and connect it. However, one future direction is to develop tools that can map higher level descriptions directly to Tensor Slices. For example, a tool could directly consume a DNN description and map fully-connected layers to Tensor Slices. FPGA synthesis tools can be enhanced to infer code patterns in RTL and map the computation to a Tensor Slice. Pragmas can be used to aid synthesis tools in mapping code to Tensor Slices. High-Level Synthesis (HLS) tools can generate RTL that instantiates Tensor Slice blocks.

Additionally, libraries of hardware components that utilize Tensor Slices could be created and users could directly instantiate them in their designs. For example, a parameterized hardware IP (Intellectual Property) block for

matrix-matrix multiplication or matrix-vector multiplication, that makes use of Tensor Slices, can be created and distributed with the FPGA tool chain. Users can directly use these blocks in their designs.

The current implementation of Tensor Slice only supports dense matrix operations. However, matrices in DL are sparse, with the amount of sparsity varying greatly across applications and across network types. The Tensor Slice architecture could be enhanced to efficiently support sparse matrix computations.

Other future enhancements to Tensor Slice to improve its frequency of operation is to break down the local input crossbar into smaller crossbars, possibly one per FPGA grid location occupied by the Tensor Slice. Finding the optimal value of Fc_{in} and Fc_{out} for Tensor Slice can also be undertaken. Exploring different switchblock configurations for Tensor Slices could be undertaken.

For CoMeFa RAMs, this dissertation explores two architectures at the ends of the area-delay design space: CoMeFa-A (area optimized) and CoMeFa-D (delay optimized). CoMeFa-D requires additional sense amplifiers to be added to the RAM block, and the number of processing elements added is equal to the number of bitline pairs. The frequency of operation of the RAM reduces by 25%. CoMeFa-A does not require any additional sense amplifiers to be added to the RAM and one PE is added for every 4 bitline pairs. The frequency of operation of the RAM reduces by 125%. Other candidates in this space can be explored in the future. For example, an architecture with one

processing element for every two bitline pairs. This will have a moderate area overhead and a moderate reduction in clock frequency.

Data needs to be stored in a transposed manner in CoMeFa RAMs for computation to be performed. But at the application level, transposing data is only needed when reading/writing from/to the DRAM. Modern FPGAs have hardened DRAM controllers integrated into them. So, transpose logic (like the swizzle module described in this dissertation) can be hardened into a DRAM controller in the future. This will eliminate any requirement of swizzle logic in soft logic, thereby implying that Logic Blocks could be used for other purposes.

The stored program method used to program CoMeFa RAMs makes it easier to program CoMeFa RAMs compared to a designing an FSM. However, it is still difficult to program CoMeFa RAMs. Developing a compiler can make this much easier. The compiler can take an application written in higher level languages like, say, a DL application written in Pytorch or TensorFlow, and generate code for the instruction RAMs that can then be decoded by the instruction controller shown in this dissertation.

Non-SRAM technologies like ReRAM or STT-MRAM have been proposed to be used for FPGA BRAMs instead of SRAMs [26] [113] [66]. Simultaneously, compute-in-memory has been explored with these technologies as well [61] [50]. In the future, adding compute-in-memory capabilities to FPGA BRAMs based on these technologies can be undertaken. The CoMeFa RAM operation is agnostic to the underlying technology.

Koios benchmarks are a set of 40 benchmarks. More benchmarks can be added to this benchmark suite to help build a better and bigger set of DL benchmarks that can guide the design of future FPGA architectures and CAD algorithms. New benchmarks such as designs that support new DNN layers can be added. More proxy benchmarks can be added. In Koios benchmarks, currently, there are 8 proxy benchmarks. Analysis presented in Section 6.2.4 shows that in the future, the proxy generator can be used to design unique proxy benchmarks that have higher linkage distances (i.e. more diverse benchmarks could be generated).

Currently, the user has to specify the exact structure of the proxy benchmark in the input YAML file. However, this framework can be enhanced so that the user only needs to specify an approximate mixture of components they desire. A YAML file can then be automatically generated and passed as an input to the existing framework, which enables easier and faster generation of proxy benchmarks from circuit properties.

In this dissertation, the compute and memory components of an FPGA are modified to design DL-optimized FPGAs. Other ways to improve the architecture of FPGAs for DL include specializing the programmable interconnect on the FPGA for DL. This can be done by analyzing the connectivity patterns between various blocks on the FPGA and changing the routing structure in a way that reduces the switches required for common connectivity patterns. For example, direct DSP to DSP connections in two dimensions could be provided, direct DSP to BRAM connections could be provided, etc.

Bibliography

[1] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu. DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, page 411–4117, Aug 2018.

[2] Achronix. Speedster7t FPGAs. `https://www.achronix.com/product/speedster7t/`, 2019.

[3] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute Caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–492, 2017.

[4] Amogh Agrawal, Akhilesh Jaiswal, Chankyu Lee, and Kaushik Roy. X-SRAM: Enabling In-Memory Boolean Computations in CMOS Static Random Access Memories. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(12):4219–4232, 2018.

[5] Jonathan Allen. UMass RCG HDL Benchmark Collection. `http://www.ecs.umass.edu/ece/tessier/rcg/benchmarks/`, 2006.

[6] Altera. Designing Filters for High Performance. https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01260-stratix10-designing-filters-for-high-performance.pdf, 2015.

[7] A. Arora, S. Mehta, V. Betz, and L. K. John. Tensor Slices to the Rescue: Supercharging ML Acceleration on FPGAs. In *2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2021.

[8] A. Arora, Z. Wei, and L. K. John. Hamamu: Specializing FPGAs for ML Applications by Adding Hard Matrix Multiplier Blocks. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 53–60, 2020.

[9] Aman Arora, Tanmay Anand, Aatman Borda, Rishabh Sehgal, Bagus Hanindhito, Jaydeep Kulkarni, and Lizy K. John. CoMeFa: Compute-in-Memory Blocks for FPGAs. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, page 1–9, May 2022.

[10] Aman Arora, Atharva Bhamburkar, Aatman Borda, Tanmay Anand, Rishabh Sehgal, Bagus Hanindhito, Pierre-Emmanuel Gaillardon, Jaydeep Kulkarni, and Lizy K. John. CoMeFa: Deploying Compute-in-Memory on FPGAs for Deep Learning Acceleration. *ACM Transactions on Reconfigurable Technology and Systems*, 2023.

[11] Aman Arora, Andrew Boutros, Seyed Alireza Damghani, Karan Mathur, Vedant Mohanty, Tanmay Anand, Mohamed Elgammal, Kenneth B. Kent, Vaughn Betz, and Lizy K. John. Koios 2.0: Open-Source Deep Learning Benchmarks for FPGA Architecture and CAD Research. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[12] Aman Arora, Andrew Boutros, Daniel Rauch, Aishwarya Rajen, Aatman Borda, Seyed Alireza Damghani, Samidh Mehta, Sangram Kate, Pragnesh Patel, Kenneth B. Kent, Vaughn Betz, and Lizy K. John. Koios: A Deep Learning Benchmark Suite for FPGA Architecture and CAD Research. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021.

[13] Aman Arora, Moinak Ghosh, Samidh Mehta, Vaughn Betz, and Lizy K. John. Tensor Slices: FPGA Building Blocks For The Deep Learning Era. *ACM Transactions on Reconfigurable Technology and Systems*, 15(4):1–34, Dec 2022.

[14] Aman Arora, Bagus Hanindhito, and Lizy K. John. Compute RAMs: Adaptable Compute and Storage Blocks for DL-Optimized FPGAs. In *2021 55th Asilomar Conference on Signals, Systems, and Computers*, page 1156–1163, Oct 2021.

[15] Aman Arora, Zhigang Wei, and Lizy John. The Case for Hard Matrix Multiplier Blocks in an FPGA. In *Proceedings of the 2020 ACM/SIGDA*

International Symposium on Field-Programmable Gate Arrays, FPGA '20, page 323, New York, NY, USA, 2020. Association for Computing Machinery.

[16] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. An OpenCL Deep Learning Accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 55–64, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Vaughn Betz and Jonathan Rose. VPR: A New Packing, Placement and Routing Tool for FPGA Research. In *International Conference on Field-Programmable Logic and Applications (FPL)*, 1997.

[18] A. Boutros, S. Yazdanshenas, and V. Betz. Embracing Diversity: Enhanced DSP Blocks for Low-Precision Deep Learning on FPGAs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 35–357, 2018.

[19] Andrew Boutros and Vaughn Betz. FPGA Architecture: Principles and Progression. *IEEE Circuits and Systems Magazine*, 21(2):4–29, 2021.

[20] Andrew Boutros, Mohamed Eldafrawy, Sadegh Yazdanshenas, and Vaughn Betz. Math Doesn't Have to Be Hard: Logic Block Architectures to Enhance Low-Precision Multiply-Accumulate on FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-*

Programmable Gate Arrays, FPGA '19, page 94–103, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C. Hoe, Vaughn Betz, and Martin Langhammer. Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, page 10–19, Maui, HI, USA, Dec 2020. IEEE.

[22] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. You Cannot Improve What You Do Not Measure: FPGA vs. ASIC Efficiency Gaps for Convolutional Neural Network Inference. *ACM Trans. Reconfigurable Technol. Syst.*, 11(3), December 2018.

[23] Robert Brayton and Alan Mishchenko. ABC: An Academic Industrial-Strength Verification Tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, page 24–40, Berlin, Heidelberg, 2010. Springer.

[24] Fredrik Brosser, Hui Yan Cheah, and Suhaib A. Fahmy. Iterative floating point computation using FPGA DSP blocks. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–6, 2013.

[25] Daniel W. Chang, Christipher D. Jenkins, Philip C. Garcia, Syed Z. Gilani, Paula Aguilera, Aishwarya Nagarajan, Michael J. Anderson,

Matthew A. Kenny, Sean M. Bauer, Michael J. Schulte, and Katherine Compton. ERCBench: An Open-Source Benchmark Suite for Embedded and Reconfigurable Computing. In *2010 International Conference on Field Programmable Logic and Applications*, pages 408–413, 2010.

[26] Yi-Chung Chen, Wenhua Wang, Hai Li, and Wei Zhang. Non-volatile 3D stacking RRAM-based FPGA. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, page 367–372, Aug 2012.

[27] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39, 2016.

[28] Lucileide M. D. Da Silva, Matheus F. Torquato, and Marcelo A. C. Fernandes. Parallel Implementation of Reinforcement Learning Q-Learning Technique for FPGA. *IEEE Access*, 7:2782–2798, 2019.

[29] Seyed Alireza Damghani and Kenneth B. Kent. Yosys+Odin-II: The Odin-II Partial Mapper with Yosys Coarse-Grained Netlists in VTR. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '22, page 157, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Seyed Alireza Damghani, Jean-Philippe Legault, and Kenneth B. Kent. Desired Footprint by Technology Mapping Modification using a Genetic Algorithm in Odin II. In *2020 International Workshop on Rapid System Prototyping (RSP)*, pages 1–7, 2020.

[31] G.H. Dunteman. *Principal Components Analysis*. Sage Publications, 1989.

[32] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. Neural Cache: Bit-Serial in-Cache Acceleration of Deep Neural Networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 383–396. IEEE Press, 2018.

[33] Mohamed Eldafrawy, Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. FPGA Logic Block Architectures for Efficient Deep Learning Inference. *ACM Trans. Reconfigurable Technol. Syst.*, 13(3), June 2020.

[34] Mohamed A. Elgamma, Kevin E. Murray, and Vaughn Betz. Learn to Place: FPGA Placement Using Reinforcement Learning and Directed Moves. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 85–93, 2020.

[35] D.G. Elliott, M. Stumm, W.M. Snelgrove, C. Cojocaru, and R. Mckenzie. Computational RAM: Implementing Processors in Memory. *IEEE Design Test of Computers*, 16(1):32–41, 1999.

[36] Hadi Esmaeilzadeh, Soroush Ghodrati, Jie Gu, Shiyu Guo, Andrew B. Kahng, Joon Kyung Kim, Sean Kinzer, Rohan Mahapatra, Susmita Dey Manasi, Edwin Mascarenhas, Sachin S. Sapatnekar, Ravi Varadarajan, Zhiang Wang, Hanyang Xu, Brahmendra Reddy Yatham, and Ziqing Zeng. VeriGOOD-ML: An Open-Source Flow for Automated ML Hardware Synthesis. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, page 1–7, Nov 2021.

[37] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A Configurable Cloud-scale DNN Processor for Real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 1–14, Piscataway, NJ, USA, 2018. IEEE Press.

[38] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, page 36–43, Boston, MA, USA, May 2014. IEEE.

[39] Sean Fox, Julian Faraone, David Boland, Kees Vissers, and Philip H.W. Leong. Training Deep Neural Networks in Low-Precision with High

Accuracy Using FPGAs. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–9, 2019.

[40] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx Adaptive Compute Acceleration Platform: Versal Architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, page 84–93, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 100–113, New York, NY, USA, 2019. Association for Computing Machinery.

[42] R. Gauchi, V. Egloff, M. Kooli, J.-P. Noel, B. Giraud, P. Vivet, S. Mitra, and H.-P. Charles. Reconfigurable tiles of computing-in-memory SRAM architecture for scalable vectorization. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, page 121–126, Boston Massachusetts, Aug 2020. ACM.

[43] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. Processing-In-Memory: A Workload-Driven Perspective. *IBM Journal of Research and Development*, 63(6):3:1–3:19, 2019.

[44] GraphCore. GraphCore IPU. `https://www.graphcore.ai/products/ipu`, 2022.

[45] Habana. HABANA GAUDI 2 WHITE PAPER. https://habana.ai/wp-content/uploads/pdf/2022/gaudi2-whitepaper.pdf, 2022.

[46] Mathew Hall and Vaughn Betz. From TensorFlow Graphs to LUTs and Wires: Automated Sparse and Physically Aware CNN Hardware Generation. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, page 56–65, Dec 2020.

[47] Mathew Hall and Vaughn Betz. HPIPE: Heterogeneous Layer-Pipelined and Sparse-Aware CNN Inference for FPGAs. *arXiv preprint arXiv:2007.10451*, 2020.

[48] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[49] Eddie Hung. Mind the (Synthesis) Gap: Examining Where Academic FPGA Tools Lag Behind Industry. In *International Conference on Field Programmable Logic and Applications (FPL)*, 2015.

[50] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 802–815, New York, NY, USA, 2019. Association for Computing Machinery.

[51] Intel. Hybrid Memory Cube Controller IP Core User Guide v16.0. https://www.intel.com/content/www/us/en/docs/programmable/683854/

16-0/introduction.html, 2016.

[52] Intel. BFLOAT16 – Hardware Numerics Definition. https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf, 2018.

[53] Intel. Intel Agilex FPGAs and SOCs. https://www.intel.com/content/www/us/en/products/programmable/fpga/agilex.html, 2019.

[54] Intel. Intel Agilex Variable Precision DSP Blocks User Guide. https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/agilex/ug-ag-dsp.pdf, 2020.

[55] Intel. Intel Arria 10 Device Datasheet. https://www.intel.com.tw/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_datasheet.pdf, 2020.

[56] Intel. Intel Stratix 10 Device Datasheet. https://www.intel.com/content/www/us/en/docs/programmable/683181/current/device-datasheet.html, 2020.

[57] Intel. Intel Stratix 10 NX FPGA Technology Brief. https://www.intel.com/content/www/us/en/products/programmable/stratix-10-nx-technology-brief.html, 2020.

[58] Intel. Intel Arria 10 Device Overview. https://www.intel.com/content/www/us/en/docs/programmable/683332/current/device-overview.html, 2021.

[59] Intel. Intel Arria 10 Product Table. https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-10-product-table.pdf, 2021.

[60] Intel. Intel Arria 10 Transceiver PHY User Guide. https://www.intel.cn/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/ug_arria10_xcvr_phy.pdf, 2021.

[61] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Computing in Memory With Spin-Transfer Torque Magnetic RAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):470–483, Mar 2018.

[62] Peter Jamieson, Tobias Becker, Peter Y. K. Cheung, Wayne Luk, Tero Rissa, and Teemu Pitkänen. Benchmarking and Evaluating Reconfigurable Architectures Targeting the Mobile Domain. *ACM Transactions on Design Automation of Electronic Systems*, 15(2), mar 2010.

[63] Peter Jamieson, Kenneth B. Kent, Farnaz Gharibian, and Lesley Shannon. Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 149–156, 2010.

[64] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory. *IEEE Journal of Solid-State Circuits*, 51(4):1009–1021, 2016.

[65] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a Tensor Processing Unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[66] Lei Ju, Xiaojin Sui, Shiqing Li, Mengying Zhao, Chun Jason Xue, Jingtong Hu, and Zhiping Jia. NVM-Based FPGA Block RAM With Adaptive SLC-MLC Conversion. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2661–2672, Nov 2018.

[67] Mingu Kang, Sujan K. Gonugondla, and Naresh R. Shanbhag. Deep In-Memory Architectures in SRAM: An Analog Approach to Approximate Computing. *Proceedings of the IEEE*, 108(12):2251–2275, 2020.

[68] Mingu Kang, Min-Sun Keel, Naresh R. Shanbhag, Sean Eilert, and Ken Curewitz. An energy-efficient VLSI architecture for pattern recognition via deep embedding of computation in SRAM. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 8326–8330, 2014.

[69] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 2011.

[70] Shreyas Kolala Venkataramanaiah, Yufei Ma, Shihui Yin, Eriko Nurvithadhi, Aravind Dasu, Yu Cao, and Jae-Sun Seo. Automatic Compiler Based FPGA Accelerator for CNN Training. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 166–172, 2019.

[71] Georgiy Krylov, Jean-Philippe Legault, and Kenneth B. Kent. Hard and

Soft Logic Trade-offs for Multipliers in VTR. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 40–43, 2020.

[72] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2 2007.

[73] Aaron Landy and Greg Stitt. Revisiting Serial Arithmetic: A Performance and Tradeoff Analysis for Parallel Applications on Modern FPGAs. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16, 2015.

[74] Aaron Landy and Greg Stitt. Serial Arithmetic Strategies for Improving FPGA Throughput. *ACM Trans. Embed. Comput. Syst.*, 16(3), jul 2017.

[75] Martin Langhammer, Eriko Nurvitadhi, Bogdan Pasca, and Sergey Gribok. Stratix 10 NX Architecture and Applications. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA'21, page 57–67, New York, NY, USA, 2021. Association for Computing Machinery.

[76] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[77] Guy Lemieux and David Lewis. Using Sparse Crossbars within LUT. In *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, FPGA '01, page 59–68, New York, NY, USA, 2001. Association for Computing Machinery.

[78] David Lewis, Vaughn Betz, David Jefferson, Andy Lee, Chris Lane, Paul Leventis, Sandy Marquardt, Cameron McClintock, Bruce Pedersen, Giles Powell, Srinivas Reddy, Chris Wysocki, Richard Cliff, and Jonathan Rose. The Stratix Routing and Logic Architecture. In *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, FPGA '03, page 12–20, New York, NY, USA, 2003. Association for Computing Machinery.

[79] David Lewis, David Cashman, Mark Chan, Jeffery Chromczak, Gary Lai, Andy Lee, Tim Vanderhoek, and Haiming Yu. Architectural Enhancements in Stratix V. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, page 147–156, New York, NY, USA, 2013. Association for Computing Machinery.

[80] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–301, 2017.

[81] Yuanfang Li and Ardavan Pedram. CATERPILLAR: Coarse Grain Reconfigurable Architecture for accelerating the training of Deep Neural Networks. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–10, 2017.

[82] Zhiqiang Liu, Yong Dou, Jingfei Jiang, Qiang Wang, and Paul Chow. An FPGA-based processor for training convolutional neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 207–210, 2017.

[83] Jian Meng, Shreyas Kolala Venkataramanaiah, Chuteng Zhou, Patrick Hansen, Paul Whatmough, and Jae-sun Seo. FixyFPGA: Efficient FPGA Accelerator for Deep Neural Networks with High Element-Wise Sparsity and without External Memory Access. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 9–16, 2021.

[84] Yehdhih Ould Mohammed Moctar, Guy G. F. Lemieux, and Philip Brisk. Routing algorithms for FPGAs with sparse intra-cluster routing crossbars. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 91–98, 2012.

[85] Kevin E. Murray, Oleg Petelin, Sheng Zhong, Jai Min Wang, Mohamed ElDafrawy, Jean-Philippe Legault, Eugene Sha, Aaron G. Graham, Jean Wu, Matthew J. P. Walker, Hanqing Zeng, Panagiotis Patros, Jason Luu, Kenneth B. Kent, and Vaughn Betz. VTR 8: High Performance

CAD and Customizable FPGA Architecture Modelling. *ACM Trans. Reconfigurable Technol. Syst.*, 2020.

[86] Kevin E. Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap between Academic and Commercial CAD. *ACM Trans. Reconfigurable Technol. Syst.*, 8(2), mar 2015.

[87] Sharan Narang. Baidu DeepBench. `https://svail.github.io/DeepBench/`, 2016.

[88] NCSU. FreePDK45. `https://www.eda.ncsu.edu/wiki/FreePDK45:Contents`, 2018.

[89] Jennifer Ngadiuba, Vladimir Loncar, Maurizio Pierini, Sioni Summers, Giuseppe Di Guglielmo, Javier Duarte, Philip Harris, Dylan Rankin, Sergo Jindariani, Mia Liu, Kevin Pedro, Nhan Tran, Edward Kreinar, Sheila Sagear, Zhenbin Wu, and Duc Hoang. Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml. *Machine Learning: Science and Technology*, 2(1):015001, Dec 2020.

[90] E. Nurvitadhi, D. Kwon, A. Jafari, A. Boutros, J. Sim, P. Tomson, H. Sumbul, G. Chen, P. Knag, R. Kumar, R. Krishnamurthy, S. Gribok, B. Pasca, M. Langhammer, D. Marr, and A. Dasu. Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent

RNNs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 199–207, 2019.

[91] Eriko Nurvitadhi, Jeffrey Cook, Asit Mishra, Debbie Marr, Kevin Nealis, Philip Colangelo, Andrew Ling, Davor Capalija, Utku Aydonat, Aravind Dasu, and Sergey Shumarayev. In-Package Domain-Specific ASICs for Intel Stratix 10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 106–1064, 2018.

[92] NVIDIA. NVIDIA Tesla V100 GPU Architecture. `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`, 2017.

[93] University of California Berkeley. Berkeley Logic Interchange Format (BLIF), 1996.

[94] Reena Panda and Lizy Kurian John. Proxy Benchmarks for Emerging Big-Data Workloads. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.

[95] Reena Panda, Shuang Song, Joseph Dean, and Lizy K. John. Wait of a Decade: Did SPEC CPU 2017 Broaden the Performance Horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282, 2018.

[96] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel paterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 389–402, New York, NY, USA, 2017. Association for Computing Machinery.

[97] Shvetank Prakash, Tim Callahan, Joseph Bushagour, Colby Banbury, Alan V. Green, Pete Warden, Tim Ansell, and Vijay Janapa Reddi. CFU Playground: Full-Stack Open-Source Framework for Tiny Machine Learning (tinyML) Acceleration on FPGAs. (arXiv:2201.01863), Oct 2022. arXiv:2201.01863 [cs].

[98] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. W. Leong. PIR-DSP: An FPGA DSP Block Architecture for Multi-precision Deep Neural Networks. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 35–44, 2019.

[99] Seyedramin Rasoulinezhad, Siddhartha, Hao Zhou, Lingli Wang, David Boland, and Philip H. W. Leong. LUXOR: An FPGA Logic Cell Architecture for Efficient Compressor Tree Implementations. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 161–171, New York, NY, USA, Feb 2020. Association for Computing Machinery.

[100] Joseph Redmon. Tiny Darknet. `https://pjreddie.com/darknet/tiny-darknet/`, 2018.

[101] Bita Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, Alessandro Forin, Haishan Zhu, Taesik Na, Prerak Patel, Shuai Che, Lok Chand Koppaka, Xia Song, Subhojit Som, Kaustav Das, Saurabh Tiwary, Steve Reinhardt, Sitaram Lanka, Eric Chung, and Doug Burger. Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[102] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 273–287, 2017.

[103] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 14–26, 2016.

[104] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[105] Shuang Song, Meng Li, Xinnian Zheng, Michael LeBeane, Jee Ho Ryoo, Reena Panda, Andreas Gerstlauer, and Lizy K. John. Proxy-Guided Load Balancing of Graph Processing Workloads on Heterogeneous Clusters. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 77–86, 2016.

[106] Sergio Spanò, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Marco Matta, Alberto Nannarelli, and Marco Re. An Efficient Hardware Implementation of Reinforcement Learning: The Q-Learning Algorithm. *IEEE Access*, 7:186340–186351, 2019.

[107] Marius Stan, Mathew Hall, Mohamed Ibrahim, and Vaughn Betz. HPIPE NX: Boosting CNN Inference Acceleration Performance with AI-Optimized FPGAs. In *2022 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–9, 2022.

[108] A. Stillmaker and B. Baas. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration, the VLSI Journal*, 58:74–81, 2017. `http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIintegration.TechScale/`.

[109] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache Automaton. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 259–272, New York, NY, USA, 2017. Association for Computing Machinery.

[110] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-Optimized OpenCL-Based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 16–25, New York, NY, USA, 2016. Association for Computing Machinery.

[111] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[112] Masakazu Tanomoto, Shinya Takamaeda-Yamazaki, Jun Yao, and Yasuhiko Nakashima. A CGRA-Based Approach for Accelerating Convolutional Neural Networks. In *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 73–80, 2015.

[113] Kosuke Tatsumura, Sadegh Yazdanshenas, and Vaughn Betz. Enhancing FPGAs with Magnetic Tunnel Junction-Based Block RAMs. *ACM Transactions on Reconfigurable Technology and Systems*, 11(1):1–22, Mar 2018.

[114] Jeffrey Tyhach, Mike Hutton, Sean Atsatt, Arifur Rahman, Brad Vest, David Lewis, Martin Langhammer, Sergey Shumarayev, Tim Hoang, Allen Chan, Dong-Myung Choi, Dan Oh, Hae-Chang Lee, Jack Chui, Ket Chiew Sia, Edwin Kok, Wei-Yee Koay, and Boon-Jin Ang. Arria 10 device architecture. In *2015 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8, 2015.

[115] Yaman Umuroglu, Yash Akhauri, Nicholas James Fraser, and Michaela Blott. LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, page 291–297, Aug 2020.

[116] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 65–74, Feb 2017. arXiv:1612.07119 [cs].

[117] Arizona State University. Predictive Technology Model. `http://ptm.asu.edu/`, 2012.

[118] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc.

[119] Dong Wang, Ke Xu, and Diankun Jiang. PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 279–282, 2017.

[120] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. LUTNet: Learning FPGA Configurations for Highly Efficient Neural Network Inference. *IEEE Transactions on Computers*, 69(12):1795–1808, Dec 2020.

[121] Jingcheng Wang, Xiaowei Wang, Charles Eckert, Arun Subramaniyan, Reetuparna Das, David Blaauw, and Dennis Sylvester. A 28-nm Compute SRAM With Bit-Serial Logic/Arithmetic Operations for Programmable In-Memory Vector Computing. *IEEE Journal of Solid-State Circuits*, 55(1):76–86, 2020.

[122] Shibo Wang and Pankaj Kanwar. BFloat16: The Secret to High Performance on Cloud TPUs. `https://cloud.google.com/blog/products/`

ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

[123] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. C-LSTM: Enabling Efficient LSTM Using Structured Compression Techniques on FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 11–20, New York, NY, USA, 2018. Association for Computing Machinery.

[124] Xiaowei Wang, Vidushi Goyal, Jiecao Yu, Valeria Bertacco, Andrew Boutros, Eriko Nurvitadhi, Charles Augustine, Ravi Iyer, and Reetuparna Das. Compute-Capable Block RAMs for Efficient Deep Learning Acceleration on FPGAs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 88–96, 2021.

[125] Zhigang Wei, Aman Arora, Pragenesh Patel, and Lizy John. Design Space Exploration for Softmax Implementations. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 45–52, 2020.

[126] Paul Whatmough, Chuteng Zhou, Patrick Hansen, Shreyas Venkataramanaiah, Jae-sun Seo, and Matthew Mattina. FixyNN: Energy-Efficient Real-Time Mobile Computer Vision Hardware Acceleration via Transfer Learning. In *Proceedings of Machine Learning and Systems*, volume 1, pages 107–119, 2019.

[127] Wikipedia. Block Floating Point. https://en.wikipedia.org/wiki/Block_floating_point, 2021.

[128] Wikipedia. Rounding. https://en.wikipedia.org/wiki/Rounding, 2021.

[129] Claire Wolf. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/.

[130] Xilinx. GEMX. https://github.com/Xilinx/gemx, 2017.

[131] Xilinx. Accelerating DNNs with Xilinx Alveo Accelerator Cards. https://www.xilinx.com/support/documentation/white_papers/wp504-accel-dnns.pdf, 2018.

[132] Xilinx. Xilinx AI Engines and Their Applications. https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf, 2018.

[133] Xilinx. UltraScale Architecture Memory Resources. https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-me pdf, 2021.

[134] Xilinx. Xilinx ACAP AI Engine Architecture Manual. https://www.xilinx.com/support/documentation/architecture-manuals/am009-versal-ai-engine.pdf, 2021.

[135] Saeyang Yang. Logic Synthesis and Optimization Benchmarks User Guide Version 3.0, 1991.

[136] Sadegh Yazdanshenas and Vaughn Betz. COFFE2: Automatic Modelling and Optimization of Complex and Heterogeneous FPGA Architectures. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(1):3:1–3:27, 1 2019.

[137] C. W. Yu, J. Lamoureux, S. J. E. Wilton, P. H. W. Leong, and W. Luk. The Coarse-Grained / Fine-Grained Logic Interface in FPGAs with Embedded Floating-Point Arithmetic Units. In *2008 4th Southern Conference on Programmable Logic*, pages 63–68, March 2008.

[138] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(11):2072–2085, Nov 2019.

[139] Jialiang Zhang and Jing Li. Improving the Performance of OpenCL-Based FPGA Accelerator for Convolutional Neural Network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 25–34, New York, NY, USA, 2017. Association for Computing Machinery.

[140] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. DNNBuilder: an automated tool for

building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*, page 56. ACM, 2018.

[141] Wenlai Zhao, Haohuan Fu, Wayne Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. F-CNN: An FPGA-based framework for training Convolutional Neural Networks. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 107–114, 2016.

Index

Vita

Aman Arora was born in Sonepat, Haryana, India. He received his Bachelor of Technology in Electronics and Communication Engineering from National Institute of Technology, Kurukshetra, India in 2007. He worked at Freescale Semiconductor and then moved to the United States to join the graduate program at the University of Texas at Austin. He received his Master of Science in Electrical and Computer Engineering from The University of Texas at Austin in 2012. After that, he worked at Nvidia and returned to graduate school at the University of Texas at Austin to pursue a Ph.D. degree in 2019. During his graduate study, he interned at Intel and Rapid Silicon. He is a student member of the IEEE and ACM.

Email address: aman.kbm@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.