Rethinking FPGA Architectures for Deep Neural Network applications

SEYEDRAMIN RASOULINEZHAD

THE UNIVERSITY OF SYDNEY

Supervisor: Prof. Philip H.W. Leong
Associate Supervisor: Dr. David Boland

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor of Philosophy (PhD)

School of Electrical and Information Engineering
Faculty of Engineering
The University of Sydney
Australia

7 July 2023

Abstract

The prominence of machine learning-powered solutions instituted an unprecedented trend of integration into virtually all applications with a broad range of deployment constraints from tiny embedded systems to large-scale warehouse computing machines. Unfortunately, the existing computer architectures are generally designed and optimized in the absence of demand for such algorithms. At the same time, these tasks are well known for their computation complexities and enormous memory requirements. Therefore, rethinking the current architectures and enhancing them, whether through minor alterations or dramatic re-designs, is crucial going forward.

Among computing platforms, field-programmable gate arrays (FPGAs) offer an excellent trade-off between customizability and performance, where the user is in charge of designing the datapath as well as the software. Because of that, FPGA technology stands on an advantageous spot in the entire spectrum of solutions, from high-performance application-specific integrated circuits (ASICs) to general-purpose processors like graphics processing units (GPUs), which respectively suffer from low adaptation and high energy consumption. Particularly, this is a key feature for deploying machine learning applications efficiently, as research points to practical deployments with promising results using optimization techniques incorporating both hardware and software.

While recent research confirms the advantages of using contemporary FPGAs to deploy or accelerate machine learning applications, especially where the latency and energy consumption are strictly limited, their pre-machine learning optimized architectures remain a barrier to the overall efficiency and performance. Realizing this shortcoming, this thesis demonstrates an architectural study aiming at solutions that enable hidden potentials in the FPGA technology, primarily for machine learning algorithms. Particularly, it shows how slight alterations to the state-of-the-art architectures could significantly enhance the FPGAs toward becoming more machine learning-friendly while maintaining the near-promised performance for the rest of the applications. Eventually, it presents a novel systematic approach to deriving new block architectures guided by implementation constraints and machine learning algorithm characteristics through benchmarking.

First, through three modifications to Xilinx DSP48E2 blocks, an enhanced digital signal processing (DSP) block for important computations in embedded deep neural network (DNN) accelerators, targeting the standard, depth-wise, and point-wise convolutional layers are described. This includes 1) a flexible *precision* and run-time decomposable multiplier architecture for convolutional neural network (CNN) implementations, 2) a significant upgrade to DSP-DSP *interconnect*, providing a semi-2D low precision chaining capability that supports our low-precision multiplier, and 3) an improved data *reuse* via a register file which can also be configured as first in first out (FIFO) buffer. Compared with the 27×18-bit mode in the Xilinx DSP48E2, this Precision, Interconnect, and Reuse-optimised DSP (PIR-DSP) offers a $6\times$ improvement in multiply-accumulate operations per DSP in the 9×9-bit case, $12\times$ for 4×4 bits, and $24\times$ for 2×2 bits. As estimated, PIR-DSP decreases the run time energy to 31/19/13% of the original value in a 9/4/2-bit MobileNet-v2 DNN implementation.

Then, two tiers of modifications to FPGA logic cell architecture are explained that deliver a variety of performance and utilization benefits with only minor area overheads. In the first tier, we augment existing commercial logic cell datapaths with a 6-input XOR gate in order to improve the expressiveness of each element while maintaining backward compatibility. This new architecture is vendor-agnostic, and we refer to it as LUXOR. We also consider a secondary tier of vendor-specific modifications to both Xilinx and Intel FPGAs, which we refer to as X-LUXOR+ and I-LUXOR+, respectively. As shown, compressor tree synthesis using generalised parallel counters (GPCs) is further improved with the proposed modifications. Using both the Intel adaptive logic module and the Xilinx slice at the 65nm technology node for a comparative study, it is shown that the silicon area overhead is less than 0.5% for LUXOR and 5–6% for LUXOR+, while the delay increments are 1–6% and 3–9% respectively. As demonstrated, LUXOR can deliver an average reduction of 13–19% in logic utilization on micro-benchmarks from a variety of domains. Binarised neural network (BNN) benchmarks benefit the most with an average reduction of 37–47% in logic utilization, which is due to the highly-efficient mapping of the XnorPopcount operation on our proposed LUXOR+ logic cells.

Eventually, with the goal of exploring this new design space in a methodical manner, a problem formulation involving computing nested loops over multiply-accumulate (MAC) operations is first proposed, which covers many basic linear algebra primitives and standard DNN kernels. A quantitative methodology for deriving efficient coarse-grained compute block architectures from benchmarks is then proposed together with a family of new embedded blocks, called MLBlocks. An MLBlock instance includes several multiply-accumulate units connected via flexible routings, where each configuration performs a few parallel dot-products in a systolic array fashion. This architecture is parameterized with support for different data movements, reuse and precisions, utilizing a columnar arrangement that is compatible with existing FPGA architectures. On synthetic benchmarks, MLBlock offers $6\times$ improved performance for 8-bit arithmetic over the commercial Xilinx DSP48E2 architecture with smaller area and delay; and for time-multiplexed 16-bit arithmetic, achieves $2\times$ higher performance per area with the same area and frequency.

In summary, this thesis raises the shortcomings of the contemporary FPGA architectures for the deployment of machine learning algorithms while keeping the focus on embedded system applications. As a result of three research studies, first, it suggests enhanced DSP blocks and logic cells using minor alterations to the commercial and available FPGA architecture. Then, it presents a methodological approach to automate the architectural search for an embedded block for a given benchmark; in our case, state-of-the-art embedded machine learning models. These solutions show promising speed-ups for machine learning algorithms while maintaining the performance for other applications. These design techniques and outcomes are an important step in moving toward efficient computer architectures for practical deployments of machine intelligence in embedded systems in various industries, e.g. transportation, communication, security and surveillance, aerospace, and health.

Declaration

I, *Mr. Seyedramin Rasoulinezhad*, declare that this thesis is submitted to fulfil the requirements for the conferral of the degree *Doctor of Philosophy (PhD)*, from the University of Sydney, is wholly my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualifications at any other academic institution.

Seyedramin Rasoulinezhad,

7 July 2023

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Prof. Philip Leong and Dr. David Boland, for their unwavering support and kindness throughout my journey. I do confess I had no clue how rewarding this experience would be. They generously encouraged me to explore architectures, which brought me such delight that I barely wanted to leave the lab early. Undoubtedly, the true gain was observing ingenuity and honesty and collecting awesome memories. I must also acknowledge that in the final days leading up to publication deadlines, I often turned to them for assistance, and despite any inconvenience, they always managed to come through and help me out. This thesis is the result of learning and progressing under their passionate guidance. I am truly fortunate to have been their student, and I thank them for trusting me.

This thesis would not have been possible or as enjoyable without the valuable input and good times shared with friends and colleagues from the Computer Engineering Lab (CEL), Sean Fox, Siddhartha, Xiangwei (Louis) Li, Stephen Tridgell, Julian Faraone, Wenjie Zhou. Our friendly conversations and shared adventures have created memories that I will cherish forever. I must also thank our collaborators, Hao Zhou and Lingli Wang from Fudan University, China, Esther Roorda and Steve Wilton from The University of British Columbia. I would also like to thank the examiners for helping us by reviewing this thesis.

Ultimately, I sincerely thank my beloved wife, who stood strongly by my side in all circumstances, made so many sacrifices, and kindly lifted up my spirit when I was at my lowest. I do appreciate her patience while I got focused on writing this thesis. And my precious parents, who undoubtedly devoted their life to me without any hesitation, hoped I could become a good man one day. They all never had an idea what I was researching about, but they always listened attentively and pretended to be interested. I feel truly blessed to have you all in my life, and I will strive to be a better husband and son.

Authorship Attribution Statement

This thesis includes several chapters with material that was previously published during my PhD candidature under the supervision of Prof. Philip Leong and Dr. David Boland. The ideas and preparation behind each publication are primarily my own work under the oversight of my supervisors, with all assistance that I received stated and acknowledged below:

- Chapter 3 of this thesis is published as [1]. The hardware synthesis of the generated Verilog models was conducted by our collaborators, Hao Zhou and Prof. Lingli Wang, the Fudan University, China.
- Chapter 4 of this thesis is published as [2]. LUXOR designs were initially mine, which were enhanced and materialized during a joint effort with Dr. Siddhartha under the supervision of my supervisors; similar to Chapter 3, the hardware synthesis of the generated Verilog models was conducted by our collaborators, Hao Zhou and Prof. Lingli Wang.
- Chapter 5 of this thesis is published as [3]. The performance analysis is conducted using a benchmark driven tool developed by our collaborators Esther Roorda and Prof. Steven Wilton from the University of British Columbia, Canada.
- The writing and presentation of each publication were improved through discussion and feedback from all co-authors.

In addition to the statements above, in cases where I am not the corresponding author of a published item, permission to include the published material has been granted by the corresponding author.

Seyedramin Rasoulinezhad, 7 July 2023

As the supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Prof. Philip H. W. Leong, 7 July 2023

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation and Aims

Machine learning algorithms are designed to leverage data to tackle complex problems by automatic pattern discovery. Although these techniques were introduced decades ago, the improvements in computing power and access to publicly available large datasets enabled them as practical solutions. This led to an enormous interest in taking advantage of their precise detections in a wide variety of applications. Examples include self-driving vehicles [4], healthcare [5, 6], finance [7], scientific discovery [8], sport [9], cyber-security [10], and many other use cases [11].

The effectiveness of machine learning algorithms in learning difficult tasks is primarily because of their flexible mathematical model and large number of parameters. Unfortunately, the same features that make these algorithms accurate also make them expensive to implement. The underlying computer hardware that implements the model plays a significant role in execution efficiency, and usually mandates software-hardware co-design to find a good solution. This is crucial in embedded systems, where the design space is strongly influenced by 1) strict memory, energy and compute resource constraints and 2) high execution expectations. To meet high performance goals, it is necessary to utilise hardware acceleration.

Among high-performance hardware implementation technologies, graphics processing unit (GPU) and application-specific integrated circuit (ASIC)-based accelerators are common but suffer respectively from high power consumption and flexibility. In contrast, field-programmable gate arrays (FPGAs) offer a middle ground with excellent power consumption and reconfigurability, where the user can co-design the data path and acceleration algorithm in an accelerator. Modifying the computing model opens the door for application-specific optimizations. Research has shown promising optimization techniques, such as quantization [11, 12, 13, 14, 15, 16, 17, 18, 19, 20], optimized-kernels [21, 22, 23, 24, 25, 26, 27], customized data-flow [28, 29, 30, 31, 32], data pruning and compression [33, 34, 35, 36, 37, 38], approximate computing [18, 39, 40], etc, enabling practical deployment of complex models. This benefit can only be realized if the hardware can be adapted to the alternative computing model. FPGA customizability aligns such characteristics.

The underlying goal of FPGA architecture research is to devise flexible substrates that implement a wide variety of circuits efficiently. FPGA architectural explorations begin with specifying a set of target applications. During the 2000s, when the FPGA architecture research was maturing, high-precision digital signal processing circuits in networking, signal and image processing applications dominated the demand. Consequently, the commercial FPGA architectures became optimized for high-precision arithmetic. They have evolved to comprise both fine and coarse-grained reconfigurable blocks arranged in a columnar fashion. The most basic units are logic elements (LEs) which are built from look-up tables (LUTs) and additional logic such as adders and flip-flops. For higher area efficiency and speed, commonly-used circuits such as memories, digital signal processing (DSP) blocks and microprocessors are implemented as coarse-grained embedded blocks (EBs). A flexible interconnection network is used to connect LEs and EBs to Input/Output (IO) blocks which include general IO, memory interfaces, and transceiver blocks. Together these form a programmable system on a chip that can implement arbitrary circuits with high performance.

During the last decade, the FPGA architectures have not dramatically changed. The improvements mostly focused on component enhancements while keeping backward compatibility rather than fundamental changes. For instance, DSP blocks improved to support even higher precision arithmetic and wider logic operations [41]. LE structures moved toward including more registers and higher interconnection flexibility [42]. Memory blocks got denser while offering various cascading and pipeline stages. More efficient routing fabric and embedding dedicated hardened circuits for performance-sensitive tasks, such as high throughput communication IP cores and central processing units (CPUs), were other architectural trends.

The demand for the deployment of embedded machine learning applications begs us to revisit the FPGA architectures by including such algorithms in benchmarks. Such applications are dominated by multiply-accumulate (MAC) operations which currently could be mapped to DSPs and LEs [43]. However, DSP blocks are heavily underutilized for low-precision deep neural networks (DNNs), while LE-based implementations are not the most efficient way to use resources. This thesis argues for different approaches to address these inefficiencies. In particular, firstly, a series of modifications to the DSP blocks to provide much better performance for deploying embedded deep learning models while maintaining backward compatibility with reasonable overheads is suggested. Next, it proposes two tiers of alteration to the LE blocks to improve the implementation cost and performance for compressor tree circuits. Finally, a new coarse-grained EB to (fully or partially) replace the DSP blocks, shifting the architecture to be more machine learning algorithm friendly is proposed.

1.1.1 High-precision DSP Blocks

Machine learning algorithms are predominantly based on MAC operations. FPGA architectures accommodate such circuits by utilizing two types of resources, DSP blocks and configurable logic elements (CLBs). However, the efficiency of such resource mappings highly depends on arithmetic precision. Due to the historical focus on accelerating high-precision computations, in current FPGA architectures, embedded DSP blocks that harden MAC operations are optimised for high precision. Although different FPGA vendors offer precision customization, due to the associated overheads, this flexibility is limited.

Meanwhile, optimising quantization techniques, the practical arithmetic precision, especially for embedded machine learning applications, has shifted to 8-bits and below [11, 12]. This precision range is not compatible with high-precision DSPs. For instance, while the Xilinx (now part of AMD) DSP48E2 is capable of executing a 27×18 multiply and 48-bit accumulate operation, for the low precision case, it only delivers two 8×8 multiplies (with shared multiplicand) with 24-bit accumulation. This is roughly a third of its potential since a 27×18-bit multiplier occupies the area of roughly six 9×9-bit ones [44]. As a result, even in a state-of-the-art accelerator, DSPs impose a performance limit [45]. The Intel FPGA architecture has similar limitations.

1.1.2 Inefficiencies in Logic Elements

CLBs may be a better match for implementing low-precision arithmetic compared with high-precision DSPs. This is especially true where multiple operations can be fused and implemented in a single merged circuit. This makes them a preferred FPGA resource for implementing low-precision MAC operations for compact dot-product blocks. The design of such circuits (also known as "parallel computer arithmetic circuits") is a well-established field of research dating back to the works of Wallace [46], Dadda [47], Swartzlander [48], Verma [49], and others. In the context of FPGAs, there has always been interest in specialized arithmetic primitives, particularly those which improve performance over a wide range of application domains. One such primitive, generalised parallel counters (GPCs), enables fast accumulation of compressor trees. However, modern FPGA LUT based architectures are not particularly efficient for the implementation of compressor trees [50]. Due to the significance of resource efficiency in embedded machine learning applications, such circuits could dramatically enhance the overall performance.

1.2 Contributions

This thesis argues the necessity of re-thinking FPGA architectures, based on the observation that current designs are heavily optimized for high-precision digital signal processing applications. Most existing architectures were developed during the time before machine learning algorithms were included in the design goal benchmarks. A particular focus is placed on the embedded applications where many promising optimization techniques, such as quantization, kernel customization, etc., offer a significant improvement in implementation costs and performance, but their implementations are not a good match to the underlying FPGA hardware. In particular, a focus is placed on the main logic implementation blocks, namely DSP blocks and LEs. A number of contributions in the form of architectural modifications and block design techniques are presented. The contributions are described in detail below.

PIR-DSP: An FPGA DSP Block Architecture for Multi-precision Deep Neural Networks: Three modifications to Xilinx DSP48E2 DSP blocks (as a case study) are suggested to boost the computations in embedded DNN accelerators while maintaining backward compatibility with reasonable overheads. First, a flexible *precision*, run-time decomposable multiplier architecture is presented. Second, a significant upgrade to DSP-DSP *interconnect*, providing a semi-2D low-precision chaining capability is suggested. Finally, we improve data *reuse* via a register file which can also be configured as first in first out (FIFO). Applying the aforementioned techniques, a Precision, Interconnect, and Reuse-optimised DSP architecture is presented, called PIR-DSP.

LUXOR: An FPGA Logic Cell Architecture for Efficient Compressor Tree Implementations: We propose two tiers of modifications to the logic cells to deliver a variety of performance and utilization benefits. In the first tier, augmenting existing commercial logic cell datapaths (vendor-agnostic) with a 6-input XOR gate in order to improve the expressiveness of each element, while maintaining backward compatibility is proposed. A secondary tier of vendor-specific modifications to two commercial FPGA logic cell architectures are also given which demonstrates that compressor tree synthesis using GPCs is further improved with the proposed modifications.

MLBlocks: Rethinking Embedded Blocks for Machine Learning Applications: explores the design space for an embedded block in a methodical manner. Thus, instead of altering the contemporary designs, it first proposes a problem formulation involving computing nested loops over MAC operations, which covers many basic linear algebra primitives and standard DNN kernels. A quantitative methodology for deriving efficient coarse-grained compute block architectures from benchmarks is then proposed together with a family of new embedded blocks, called MLBlocks.

1.3 Thesis Structure

The main contributions of this thesis have been previously published in the following references:

- Chapter 3: [1] SeyedRamin Rasoulinezhad, Hao Zhou, Lingli Wang, and Philip H. W. Leong, "PIR-DSP: An FPGA DSP Block Architecture for Multi-precision Deep Neural Networks", 27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019.
- Chapter 4: [2] SeyedRamin Rasoulinezhad, Siddhartha and, Hao Zhou, Lingli Wang, David Boland, and Philip H. W. Leong,"LUXOR: An FPGA Logic Cell Architecture for Efficient Compressor Tree Implementations", The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2020.
- Chapter 5: [3] Seyedramin Rasoulinezhad, Esther Roorda, Steve Wilton, Philip H. W. Leong, David Boland, "Rethinking Embedded Blocks for Machine Learning Applications", ACM Transaction Reconfigurable Technology Systems, 2022.

This thesis presents the contributions of the abovementioned publications, each as a chapter. In addition, it takes the opportunity to present the suggested architectures as a series of research outcomes, starting by enhancing the current architectures (DSP and LE blocks) while keeping the backward compatibility. Then, it puts aside the backward compatibility and proposes a new methodology to automate the FPGA EB designing process with regard to a given benchmark, covering machine learning algorithms.

An overview of the technical details of chapters is encapsulated by Figure 1.1, where the structure for each chapter is given below:

- Chapter 2 provides background on 1) machine learning algorithms: DNNs and hardware-software co-design optimizations, 2) hardware accelerator platforms, 3) FPGA technology and contemporary architectures, and 4) compressor trees, parallel counters, and compressor circuits.

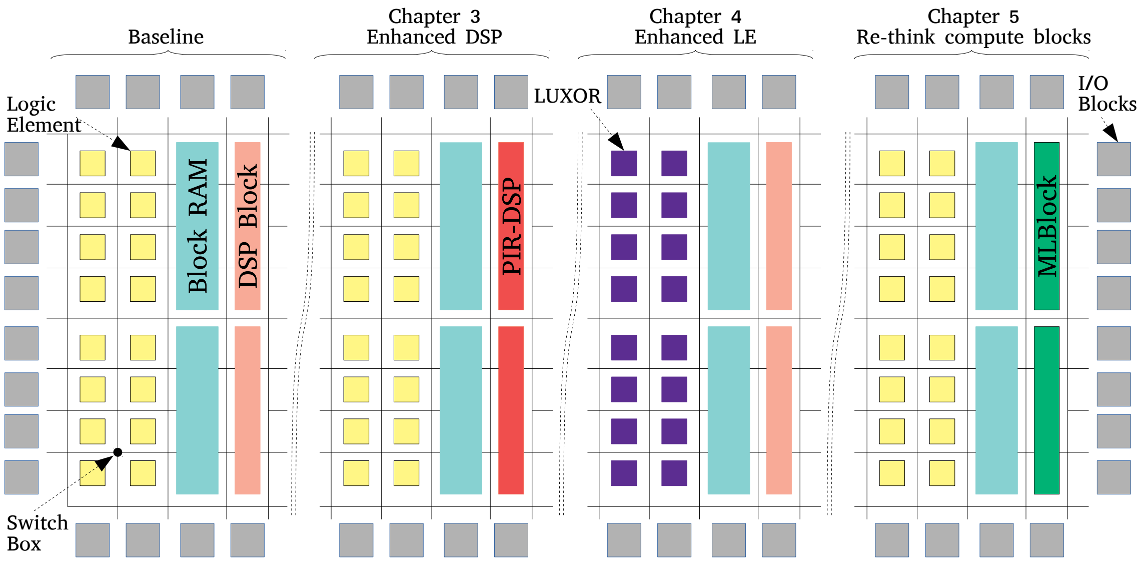


FIGURE 1.1. An overview of the presented chapters

- Chapter 3 suggests three modifications to FPGA DSP blocks (using Xilinx DSP48E2 DSP blocks as the case study), to boost the computations in embedded DNN accelerators while maintain backward compatibility with reasonable overheads. Applying the aforementioned techniques, a Precision, Interconnect, and Reuse-optimised DSP architecture is presented called PIR-DSP.
- Chapter 4 it proposes two tiers of modifications to the current FPGA logic cells to deliver a variety of performance and utilization benefits. It starts by augmenting existing commercial logic cell datapaths with a 6-input XOR gate (vendor-agnostic) in order to improve the expressiveness of each element, while maintaining backward compatibility. Then, a secondary tier of vendor-specific modifications to two commercial FPGA logic cell architectures is suggested.
- Chapter 5 explores the design space for an embedded block in a methodical manner. This is in contrast to altering the contemporary designs while guaranteeing backward compatibility and sacrificing the performance (like the presented works in Chapters 3 and 4). This Chapter first proposes a problem formulation involving computing nested loops over MAC operations, which covers many basic linear algebra primitives

and standard DNN kernels. A quantitative methodology for deriving efficient coarse-grained compute block architectures from benchmarks is then proposed together with a family of new embedded blocks, called MLBlocks.

- Lastly, Chapter 6 concludes the abovementioned research outcomes and discusses some of the understudied research questions as a suggestion for potential future works.

CHAPTER 2

Literature Review

Architectural research starts by defining target applications and the design exploration space. This work pursues new FPGA block architectures optimised for embedded deep learning applications and requires a comprehensive understanding of both DNN algorithms and FPGA technology. Background on these topics are covered in this chapter.

The evolution of machine learning algorithms has ignited an unprecedented enthusiasm for deploying such techniques in a multitude of applications. Among different machine learning techniques, developers are particularly interested in deep learning models as their generic structures make them universal solutions for many applications. This comes at the cost of computation resources and energy, which are always limited, irrespective of whether the target hardware is high-end servers or at the edge. This compels developers to trade traditional computation models, arithmetic precision, and sometimes accuracy for practical deployment. Efforts to find hardware-friendly models with minimal accuracy loss resulted in a collection of techniques. Integrating more native support for such optimisations is the goal of this thesis. Section 2.1 introduces deep learning, and comments on the role of embedded DNNs, and elaborates on performance-accuracy tradeoffs.

Although such optimisations benefit all platforms, customizability makes FPGAs advantageous among the wide spectrum of hardware accelerators from ASICs to general-purpose processing units. This advantage, especially for the inference task, is reached despite the fact that FPGA architectures are generally evolved in the absence of machine learning algorithms as highly in-demand applications. Section 2.2 advocates the gradual evolution of traditional FPGA architectures and elaborates on various FPGA blocks and their use cases in embedded DNN deployments. Through this introduction, previous studies, academic and commercial solutions, and potential architectural opportunities aiming for more efficient FPGA architectures support for embedded DNNs are discussed. Moreover, a three-class taxonomy to summarise the most recent architectural innovations in designing FPGA embedded blocks is suggested. This starts from minor modifications to the current micro-architectures and extends to integrating domain-specific engines and rethinking new EBs.

With a focus on arithmetic circuits, a common primitive is multi-operand addition and partial product reduction trees. These are used in parallel multipliers and utilise carry-save arithmetic. Compressor trees are a generalisation of this class of circuits, which historically received attention for digital signal processing applications. Effective hardware support for them is applicable to virtually all arithmetically intensive data flow graphs. In particular, DNNs are rich in compressor trees due to the predominance of MAC operations and in computationally fused version dot products. Section 2.3 provides a brief introduction to compressor trees and their implementation techniques that vary from one hardware platform to another.

In more detail, this chapter is organised as follows:

- Section 2.1, Overview of embedded deep neural networks: This section provides a background on the context of machine learning (ML) and DNN models, summarises basic computation kernels, and provides insights on various optimisation trends for both DNN models and deployment techniques leading to practical embedded DNNs.
- Section 2.2, FPGA architectures and opportunities for efficient deployments: This section presents why FPGA technology is uniquely suitable for high-performance accelerators. The gradual evolution to state-of-the-art FPGA architectures during the past two decades is discussed, explaining the reason as to why existing architectures are not highly optimised for embedded DNN applications.
- Section 2.3, Compressor trees: This section explains the primary concepts regarding compressor trees, including generalised parallel counter compressor circuits, and their applicability to the FPGA technology. Finally, a brief review of prior architectural studies to support compressor trees on FPGAs is given.

2.1 Embedded Deep Neural Networks Overview

This section starts by providing a brief background on artificial intelligence (AI), ML algorithms, and deep learning. Then, it defines DNN basics, including computation kernels and network architectures. Eventually, optimisation trends in DNN designs that seek more practical and hardware-friendly models are introduced. These techniques are often employed for embedded deployment, so we call such models embedded DNNs.

2.1.1 Artificial Intelligence, Machine Learning, and Deep Learning

According to John McCarthy, a founder of the AI discipline, the engineering of creating intelligent machines that can achieve goals like humans is called AI [51]. This very broad definition covers all methods that enable computers to reason based on rules between the inputs and outputs of a task. Traditional symbolic reasoning builds such intelligence through assembling human-designed rules; e.g., IBM DeepBlue [52] is a chess engine that relies on instructions and variables defined and fine-tuned by chess masters and computer scientists.

In contrast, ML algorithms are a subset of AI solutions in which the machine can learn rules by establishing the correlation between the inputs and outputs without being explicitly programmed. Such abstraction allows models to learn different tasks, at odds with the pre-machine learning era, when a custom algorithm was required for each problem. AlphaZero [53] by DeepMind is another Chess engine, which in contrast, learns by self-play. It leverages reinforcement learning, which is generalisable to other games such as Go and Shogi.

Inspired by the human neural system, artificial neural networks (ANNs) became a group of machine learning solutions that employ a brain-like network of artificial neurons (interchangeably called perceptrons) to mimic biological networks. Each artificial neuron mimics the signal transformation function of a biological neuron using the weighted sum of input signals and a bias value, followed by a non-linear function [54] (such as sigmoid, hyperbolic tangent, rectified linear unit (ReLU) [55] and its variations [56, 57]) to produce an output [51]. This computation is illustrated in Equation 2.1 where W_i, X_i and Y, and b are the weights, input signals (activations), output signals, and the bias value respectively, and $f(.)$ represents the non-linear function (Figure 2.1A). The functionality of each neuron depends on the weight and bias values (parameters) which are chosen using a training process [58].

$$Y_j = f(\sum_i W_{ji} X_i + b) \tag{2.1}$$

In a practical setting, a neuron receives a large number of input signals, and so this computation model became dominated by the dot-product between the inputs and the weight parameters. While a multitude of more complex and realistic neuron models have been proposed [59], the dot-product operation remains the computational bottleneck. This key observation establishes a foundation for hardware accelerators that leverage high-performance multiply and accumulate dot-product units for efficiently executing neural networks. It is worth noting there are other approaches to formulate a neural system, like spiking neurons [60] which use continuous mathematics. This topic is out of the scope of this thesis.

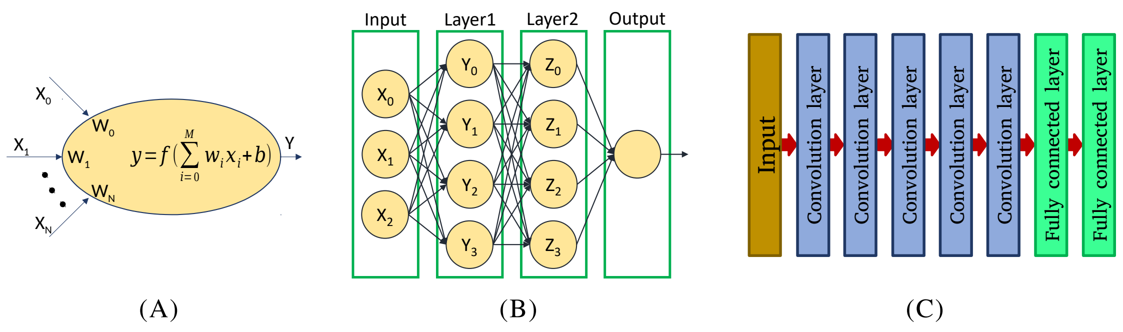


FIGURE 2.1. (A) The mathematical model of a neuron. (B) A neural network sample. (C) A deep neural network sample (convolution and fully connected layers are defined later in Section 2.1.2)

As illustrated in Figure 2.1B, by constructing a network of interconnected neurons, a neural network (NN) can be constructed. Although neurons can be bound in any arbitrary arrangement, in practice, neurons are aggregated into layers. Such structures receive the inputs via input nodes (or input layer) and process them layer by layer (hidden layers). Based on the neuron types and inter and intra-layer connections, a variety of neural network architectures are possible. More efficient and frequently used cases will be discussed in Section 2.1.2. Figure 2.1C pictures a practical network structure, called AlexNet [61], that is built by stacking seven layers of neurons. In the last layer (known as the output layer), the final outputs are generated. The computation of the outputs of the NN from the inputs is called the inference task.

In contrast to inference, training is the process of finding the best values for the model parameters, including the weights and the bias. This is done using an optimisation algorithm (such as gradient descent [62] or Adam [63]) to minimise a loss function. This process compares the NN's output with the desired output (if it is available in supervised learning) or the output's cost [64] (in unsupervised and reinforcement learning [65]). To optimise all neural layers based on the same loss function, each layer's gradients can be calculated either from the next layer's gradients using backpropagation [58] or directly from the final loss function using direct feedback alignment [66].

2.1.2 Deep Neural Networks: Basic Kernels and Architectures

The DNN refers to a class of artificial neural networks that are built by stacking multiple layers of neurons to extract higher-level features from the raw input progressively [67]. For such architectures, the number of hidden layers is called the network depth, typically greater than three. Generally, the deeper the model gets, the more complex features the model can extract. During training, the initial layer(s) learn to detect simple and basic attributes (features) of the input data, whereas the following layers augment such information to produce more complex understandings. Layer after layer, such cognition enhances and provides better insight for the final layers to utilise toward generating the desired outputs.

A DNN architecture can be described by its neural layers and their connections, and such flexibility in the layer types and interconnections enables an endless number of combinations. However, randomly picking multiple layers and arbitrarily linking them does not normally form an efficient architecture. Adding more layers, increasing the number of neurons in layers, or using more sophisticated neuron models and connection patterns can usually be translated to more architectural flexibility; consequently, higher accuracy could be expected. However, all the above will predominantly increase the implementation costs for training and inference tasks.

To tackle this issue, researchers initially explored the design space manually, which led to a set of primitive layers that benefit a wide range of applications [68]. These layers are generic and should be instantiated in a reasonable order and sized according to the target application. These days, network architecture search (NAS) techniques [69] automate such architectural exploration, including the best placement, sizing, and connections, aiming for the most efficient DNN architectures by involving both the performance and implementation costs.

A neural layer is a collection of neurons that receives a multi-dimensional tensor of input signals (called the input feature map or activation signals) to process according to a specified connection pattern between the input feature map elements and the neurons. This pattern and the neuron computation model mathematically translate to multi-dimensional kernel operations. Eventually, each neuron generates an output signal, where all those outputs together form another multi-dimensional tensor called the output feature map. As a simple example, assume a layer with only one neuron. Such a layer accepts a 1-D input feature map where all input elements are connected to that neuron. Using Equation 2.1, the layer computation is the dot-product operation between the 1D input feature map and a same size 1-D tensor of weight parameters, followed by adding a scalar value (the bias value) and then applying a non-linear function over that. The output is a scalar value that forms a 0-D output feature map.

2.1.2.1 Fully connected (FC) Layer

A FC layer can be realised as a layer of neurons where each neuron is connected to all elements of the input feature map. A generalisation for a FC layer is presented in Figure 2.2A, where a FC layer with N neurons takes an M-element feature map $\mathbf{F}$ as input, and produces an N-element feature map $\mathbf{G}$ as output. The computation is a vector-matrix multiplication of the input feature map with an $M \times N$-element kernel $\mathbf{K}$ followed by N-element vector addition for the bias values (B), and eventually passing the result through an element-wise non-linear function (assuming all neurons in a layer use the same non-linear function, f), i.e.:

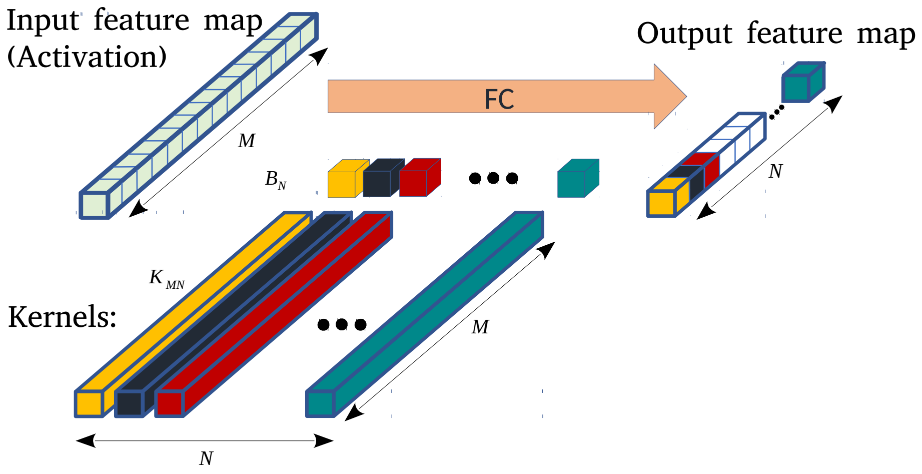
$$\mathbf{G}_j = f(\sum_{i=1}^{i<M} \mathbf{K}_{i,j}\mathbf{F}_i + \mathbf{B}_j). \tag{2.2}$$

Accordingly, the computation complexity and memory footprint for this layer are in $O(NM)$ and $O(NM)$, respectively, which pose challenges in practice, especially for large N and M values. The dominant computations are the vector-matrix multiplication (or, in a finer-grained manner, the neuron's dot-products that share the same input vector) and the element-wise adder and non-linear functions. Evaluation of these basic linear algebra operations famously benefits from parallelism and data reuse [70]. To minimise the memory requirement, besides efficiently sizing the layer parameters, N and M, lowering the data precision[71] and benefiting sparsity [72] are two practical solutions.

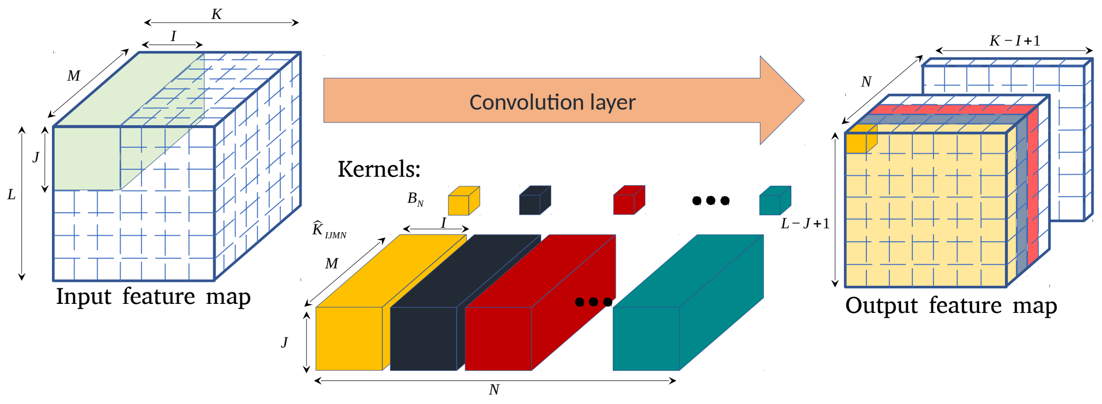
FC layers offer a trainable structure to execute regression analysis models [73]. To extend this feature, multiple FC layers can be stacked, one after another, to form a hierarchical regression model. This led to multi-layer perceptron (MLP), a group of ANNs, that wrap such structure (one or more FC layers) within an input layer and an output layer. This kind of neural network can solve various tasks stochastically, such as function approximation [74] and classification [75]. MLPs suffer from implementation costs due to their high degree of connectivity; consequently, they are not always the most efficient solutions. Yet, to benefit from the FC layer's characteristics, network designers add multiple FC layers in the decision-making stage (mostly after the feature extraction phase), where the FC layers match the desired task, e.g. convert high-level features to a class in image recognition. Google researchers have recently shown that FC layers and data reshaping layers alone can achieve close to the state-of-the-art results [76].

2.1.2.2 Convolutional (Conv) Layer

The convolution operation is simply a sequence of sliding dot products. The kernel tensor is shifted along the input tensor, and for each sliding position, the dot-product of the intersection



(A)



(B)

FIGURE 2.2. (A) a Fully connected layer. (B) a standard convolution layer.

points in both tensors becomes an entry in the output tensor. Using this operation, a group of neural layers called convolutional layers are derived. In these layers, a kernel slides over the input feature map (in defined directions) to generate an output per intersection window between them. Accordingly, the kernel is reused to generate multiple outputs utilising different windows of the input feature map. Also, in convolutional layers, the kernel only operates over a neighbourhood of the input feature map. This is in contrast to the FC layers, where each neuron connects to all input feature map elements. Convolutions allow the kernel to extract the same feature over the input tensor while mitigating implementation costs by reducing the number of connections and parameters.

As an example for the convolutional layer, Figure 2.2B shows a standard convolutional (SConv) layer that takes a $D_F \times D_F \times M$ feature map $\mathbf{F}$ as input, and produces a $D_G \times D_G \times N$ feature map $\hat{\mathbf{G}}$ as output. The output is generated via a convolution with a $D_K \times D_K \times M \times N$ kernel $\hat{\mathbf{K}}$ followed by adding the corresponding neuron's bias value and applying the non-linear function, i.e.:

$$\hat{\mathbf{G}}_{k,l,n} = f(\sum_{i=0,j=0,m=0}^{i<I,j<J,m<M} \hat{\mathbf{K}}_{i,j,m,n}\mathbf{F}_{k+i,l+j,m} + \mathbf{B}_n). \tag{2.3}$$

This layer consists of N neurons where each accesses a $I \times J \times M$ sliding window of the input feature map. Hence, each neuron produces a channel of the output feature map, which comprises $(K - I + 1) \times (L - J + 1)$ output entries (corresponding to each sliding position). The kernel shape may limit the sliding positions (particularly when a kernel size in a sliding dimension is more than 1), which causes neurons to skip processing the input feature map entries on the borders in some relative positions, as in such cases, the kernel window overpasses the input feature map borders. Padding is a technique that expands the input feature map with extra entries on outer borders (mostly with a default value). For instance, to maintain the input feature map dimensions in a SConv layer, the kernel slides alongside the K and L dimensions when the kernel sizes are I and J, respectively. Therefore, the input feature map should be padded by $(I - 1)/2$ and $(J - 1)/2$ on each side of the K and L dimensions.

Besides the SConv layer, there are many other convolutional layer choices that are defined by various kernel shapes, sliding directions and steps. For instance, the kernel shape indicates the neuron's spatial access pattern to the input feature map. Expanding (or shrinking) a kernel along a dimension enables (or disables) the activations to extract spatial features in that direction [21]. Kernel dilation [77], pursue the same goal by keeping the number of neuron's connections unchanged while exponentially expanding the convolution receptive field (Figure 2.3A). On the other hand, the sliding specifications control in which direction(s) and rate(s) a neuron should sample the input feature map to measure its configured feature. This is beneficial when a temporal or spatial change in the input feature map is expected (Figure 2.3B). As evident in the figure, changes to the sliding affect the number of times a neuron operates and hence the output feature map size.

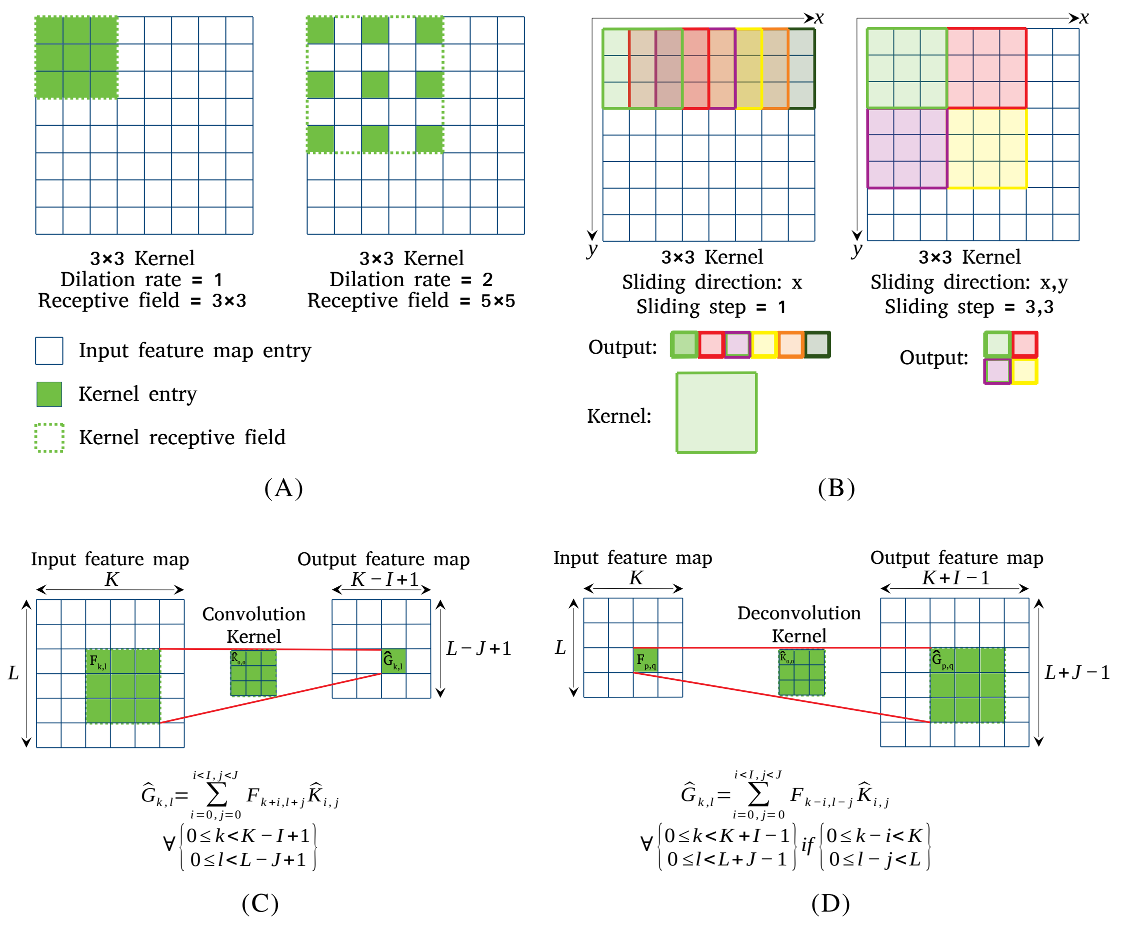


FIGURE 2.3. (A) Dilation in convolution layers. (B) Sliding direction and step. (C) A 2D convolution layer vs (D) A 2D deconvolution layer

Based on customizing the abovementioned specifications, variants with different performance and implementation costs can be derived. Some efficient variations will be covered later in Section 2.1.4.

The deconvolution layer (also known as transposed convolution) is another convolutional layer which is the opposite process of the convolution layer. A kernel slides over the output feature map, where each sliding position corresponds to an entry in the input feature map. The kernel projects the impact of this entry on a window of the output feature map entries. The window is the intersection of the output feature map and the kernel, and the corresponding kernel value scales the impact. Figures 2.3C and 2.3D compare a 2D convolution layer with a 3×3-kernel with a 2D deconvolution layer with the same size kernel. In contrast to a convolution layer which extracts features from a given input, the deconvolution layer tries to regenerate the corresponding convolution layer's input using the extracted features. The main application of deconvolution layers is when convolution and deconvolution layers are coupled together to build generative adversarial networks (GANs) [78] and auto-encoders [79]. These are particularly powerful tools for denoising [80], anomaly detection [81], and segmentation tasks [82].

The convolutional neural network, or convolutional neural network (CNN) for short, is a class of ANNs that are primarily based on stacking convolutional and FC layers. The convolutional layers mostly act as the feature extractor layers at the beginning of the network, while the FC layers implement the final stage decision making tasks such as classification. This lowers the computation and memory requirements compared to the MLPs, while it preserves the necessary structures (flexible kernels) to extract the spatial and temporal dependencies, e.g. multi-dimensional data (like works in computer vision [83]) and time series applications (like radio frequency signals [84]) applications.

Apart from FC and convolution layers, state-of-the-art DNNs include other layers such as Pooling layers [85]. Their purpose are 1) to add robustness against minor changes in the input (such as input perturbation or transformations like shift, rotate, zoom, etc.) and 2) to reduce the feature map size by discarding the least significant information. A pooling layer tiles the input feature map (using a window called polling kernel) and generates an output entry per each tile by combining (like using averaging) or pooling between (like selecting the maximum value) the neighbourhood of input feature map entries in each tile. For instance, after a SConv layer, the values in the same channel are the result of sliding the same kernel over different sliding positions. Therefore, the adjacent entries in a channel measure the same feature for two nearby positions. Consequently, replacing a neighbourhood of entries in a channel with the average of them 1) enhances the feature measurement precision and 2) reduces the feature map by discarding values that were very similar anyway.

Normalisation layers are also frequently used and aim to normalise the distribution of feature maps. This particularly speeds up the training and also the final accuracy in DNNs [86]. Batch normalisation (BN) [87] restricts the input value distributions for mini-batches of feature maps by removing the average bias and scaling down the values by the batch deviations. Weight Normalisation [88] and Layer Normalisation [89] are two newer techniques that sacrifice the training stability for improved computation speed by normalizing with a single vector instead of a mini-batch.

Concatenation [90], shift [23], shuffle [24], and element-wise [91] layers are other DNN layers that offer more connection patterns between the layers comparing to basic layer stacking. They generally do not consume a significant fraction of the total computation cost and, according to [92], convolution and FC layers together contribute more than 99% of the computations and required memory.

2.1.3 DNNs: The Rise, Evolvement, and Trends

As Charles C. Tappert [93] reasons, the history of DNNs goes back to 1962 when Frank Rosenblatt suggested a basic foundation for deep learning systems [94]. However, it took years of research until Yann LeCun et al. [95] finally proposed a practical training approach based on backpropagation, which enabled training deep artificial neural networks. Applying this method, they proposed the first deep CNN structure, LeNet [96], particularly designed to recognise hand-written digits. The most well-known variant of this network, LeNet-5, is built by stacking two 2D standard convolution layers with 5×5 kernel sizes, respectively, using 6 and 16 filters with no input feature map padding and three FC layers, respectively, with 120, 84, and 10 neurons [97]. Furthermore, to manage the feature map spatial expansion due to the number of filters in the convolution layers, they coupled each Conv layer with an average pooling layer with the 2D kernel window of 2×2 and stride of 2, which condenses the feature map resolution by $4\times$. Also, all neurons in this structure use the Sigmoid function as their non-linearity (Figure 2.4A).

The lack of high-performance computing systems for such workloads caused a huge burden for deeper models. Although LeNet-5 was a deep model at the time, compared to state-of-the-art CNNs, it is relatively tiny, with in total of 60k parameters administered for 341k MACs per given 28×28-pixel input image [51]. Even this 5-layer network required three days of

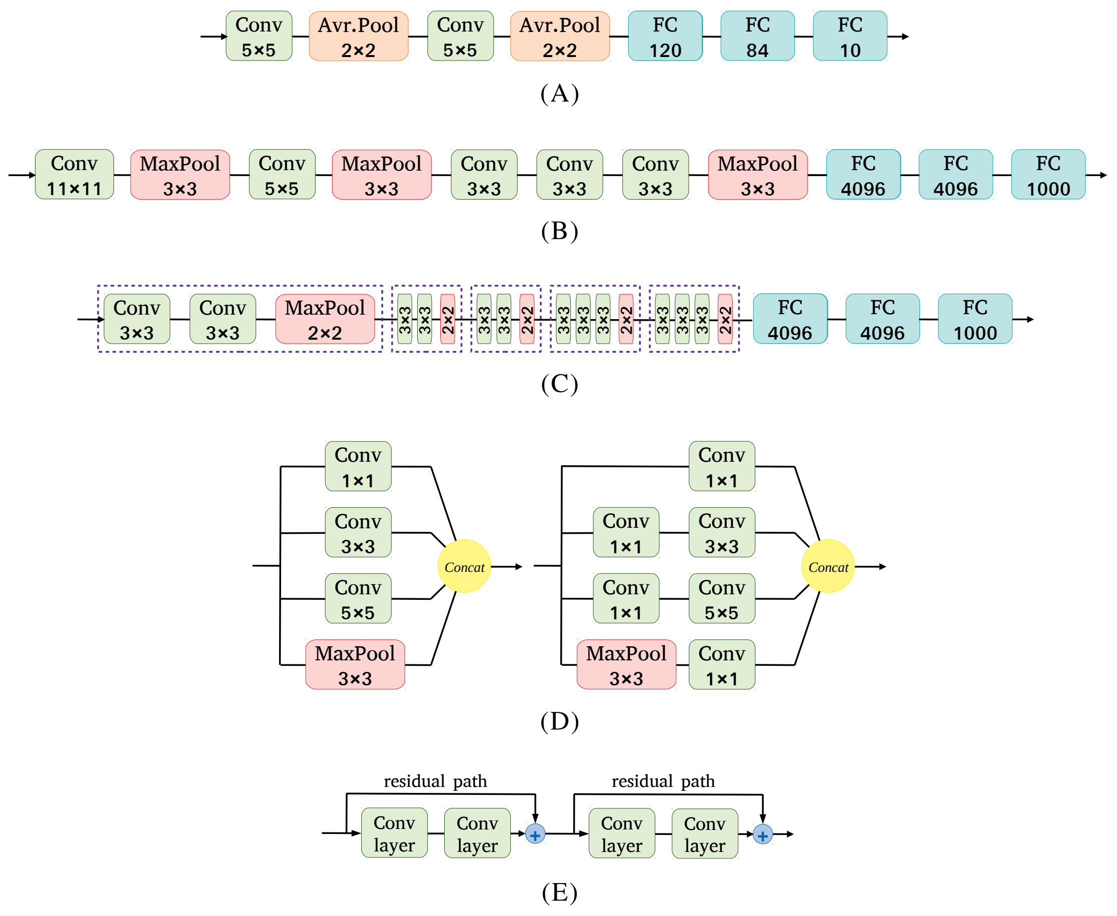


FIGURE 2.4. The schematic for (A) LeNet [96], (B) AlexNet [61], (C) VGG-16 [90], (D) GoogLeNet inception modules [98] (with and without down sampling respectively at left and right), and (E) vanilla residual path for ResNet module [91] (without downsampling)

training on a sun workstation, which was an enormous enhancement compared to the previous works.

The next substantial step for deep learning took place during the 2000s when faster computing systems with vector processors and GPUs were developed. By 2012, the tremendous advancement in parallel computing and general-purpose computing on graphics processing units (GPGPUs) triggered a turning point for deep learning when AlexNet [61] won the ImageNet Large Scale Visual Recognition Competition (ILSVRC) challenge [99], targeting the large ImageNet dataset, with 10,000 categories and 10 million images. This network achieved a Top-5 classification error rate of 15.3%, while the following best result was a 26.2% error rate from a non-neural network model (all reported performance metrics are from [51]). As it is shown in Figure 2.4B, This network includes five standard CONV layers and three FC layers that conclude to a storage requirement of 62M parameters and used 724M MAC operations per classification, which are roughly three orders of magnitude more than the corresponding metrics for the LeNet-5.

Meanwhile, the ILSVRC competition has become a global motivation for further significant progress on DNNs. This was the beginning of a new era for DNN, which was dominantly dedicated to accuracy improvement by adding more parameters and computational complexities. In the ILSVRC 2013 challenge, VGGNet was the runner-up with 7.4% Top-5 error and enhanced classification accuracy rates further using deeper CNN models with smaller convolution filters. VGGNet-16 was the variant used in the competition that required 138 M parameters and 15 G MAC operations [90], which are respectively are respectively about $2\times$ and $20\times$ more than AlexNet counterparts. The key idea was using modular architecture by only performing 3×3 convolution and 2×2 pooling layers, which enables deeper networks while maintaining the model size (Figure 2.4C).

In ILSVRC 2014, the GoogLeNet [98] took first place by achieving 6.7% Top-5 error by stacking the inception module, which incorporates multiple neural layers in parallel. In this module, multiple convolutional layers operate on the same input feature map, while the concatenation of their output becomes the module's output. This layer connectivity enables GoogLeNet architecture to slightly surpass the VGGNet accuracy performance while only using 7 M parameters and 1.43 GMACs over 22 layers. Although the reduction in parameters and the computations are significant, the layer variations in computation and connectivities pose challenges to the hardware accelerators that should switch between multiple computation kernels in run-time. This work also offers auxiliary classifiers that temporarily connect to the networks to boost the training process by compensating for gradient vanishing.

The next year, the ResNet [91] structure, based on adding residual path beside the neural layers (like Figure 2.4E), won the competition with a 5.3% Top-5 error. These connections allow the gradients to pass through the layers without vanishing, which concluded successful training for very deep networks, e.g., ResNet-50 with 25.5 M parameters for 3.9 G MAC operations. Another variant of this architecture, ResNet152, was the first NN that exceeded human accuracy (better than 5% Top-5 classification error rate for ImageNet data set) with over 11.3B FLOPs.

Since then, numerous DNNs with higher accuracy rates have been developed, and an in-depth analysis for substantial cases has been published by Simone Bianco et al. [100]. For instance, most recently, EfficientNet-L2 [101] accomplished the state-of-the-art 98.8% top-5 and 90.2% top-1 accuracy rates as a deep CNN for the ImageNet [99] dataset. However, this superior performance comes at the cost of massive memory and computation requirements, which are not always justifiable or sometimes not available, particularly for embedded system applications with constrained resources.

Performance-aware DNN designing [102] was an early solution to restrain the resource requirements, where the performance metrics (such as throughput, latency, classification rate, memory footprint, energy per classification, etc.) besides the accuracy guide the manual or automated NAS. Although this approach extracts the best-performing network for a fixed architecture with known resource budgets and a performance goal, the efficacy of the accelerator architecture itself remains questionable. The next step was relaxing the hardware architecture, which began the new era of network and hardware co-designing, where feedback from one to another enable more optimisation opportunities. This also enormously expands the design space that demands more efficient NAS techniques [103].

2.1.4 Embedded DNNs

There has been considerable interest in memory and computationally efficient DNNs for mobile and embedded applications where the common most crucial concerns are low latency and low energy while complying with strict resource constraints. In order to manage the massive computation and storage complexities of DNNs, efforts at reducing hardware resource usage at all design levels have been undertaken, *e.g.* efficient computational kernels [23, 24, 22, 26, 37, 27], data pruning [33, 34], memory compression [35, 36] and quantisation [17, 38, 104, 20, 19]. Although these techniques benefit any sort of DNN deployments, they particularly enable a series of optimisations to fit DNN models on embedded devices. This section reviews some of the key works.

2.1.4.1 Efficient Kernels

The presented standard 2D convolutional layer in Equation 2.3 is, in fact, a single instance from the convolutional layer generalisation, where numerous other instances can be defined by varying the convolution window dimensions and dilations and sliding directions and strides. Each configuration introduces different feature extraction capabilities at a particular computation cost. For example, eliminating a dimension from the convolution filter window stops the neurons' spatial access to the input feature map in that dimension. Thus, the convolution filters lose the chance to extract features incorporating input entries from the cancelled connections. Meanwhile, the downsize in convolution windows directly translates to less memory footprint and computation.

The early research on this multifarious feature extraction-cost tradeoff resulted in many optimised convolution layers, such as the grouped convolution layer that was primarily introduced in AlexNet [61] to tackle the computation complexities by limiting the neurons' connections. In contrast to standard convolution, these layers splits the input feature map into disjoint groups and each is convolved with separate and smaller kernels. This enables computation distribution over multiple GPUs and consequently boosts performance, as also reported in recent works such as ResNeXt [105] and SqueezeBERT [106]. The main drawback of channel grouping is limiting the interaction between features from different groups with each other. To partially solve this issue, Zhang et al. [24] proposed using a shuffle unit on top of grouped convolution layers, where this unit shuffles the features map channels in a way that each group maintains a similar representation into the next grouped convolution layer.

The *point-wise convolution (PW)* layer [107] is another influential case, where in comparison to the standard 2D convolution layer, the filter windows shrink to 1×1. This allows feature extraction only from the input feature map entries from different channels, which can be seen as trainable pooling for channels. Thus, this layer is particularly suitable for reducing or increasing dimensionality (As shown at the left side of Figure 2.4D for the inception module with downsampling [98]).

Inspired by PW convolution layers, Jin et al. [25] proposed flattening the 3D convolutional filters of the standard 2D Conv layer into three consecutive one-dimensional filters across all directions in 3D. This technique considerably reduces the memory footprint and speeds up the inference task up to $2\times$ while maintaining the accuracy for simple datasets such as CIFAR 10 and CIFAR 100.

Figure 2.5 illustrates another highly efficient substitute to the standard 2D convolution layer from MobileNet [21, 108], *depth-wise separable convolutions* which first factorises Equation 2.3 into M *depth-wise convolutions (DWs)*

$$\hat{\mathbf{G}}_{k,l,m} = \sum_{i,j} \hat{\mathbf{K}}_{i,j,m} \mathbf{F}_{k+i-1,l+j-1,m} \tag{2.4}$$

where $\hat{\mathbf{K}}$ is the $D_K \times D_K \times M$ depth-wise kernel and the m^{th} filter of $\hat{\mathbf{K}}$ is applied to the m^{th} channel of F to produce the m^{th} channel of $\hat{\mathbf{G}}$. Linear combinations of the M depth-wise layer outputs are then used to form the N outputs using a PW layer with N layers. A speedup of $\frac{N+D_K^2}{ND_K^2}$ is achieved where the typical values is about $8-9\times$ (for $D_K = 3$), with a minor reduction in accuracy for large datasets such as ImageNet [99]. A study on the speed/accuracy trade-offs of convolutional object detectors compares the use of the Inception, MobileNet, ResNet and VGG networks as the feature extractor backbone for object detection tasks, where MobileNet architecture achieves an excellent accuracy if low execution time on a GPU is desired [109]. With a similar idea, Wu et al. [23] also proposed a shift-based channel movement as a zero parameter alternative to the DW convolution in depth-wise separable convolutions, which reduces model parameters drastically and also the number of MAC operations.

The convolution operations can also be accelerated by alternative computations. Zhang et al. [27] proposed accelerating convolution operations by using the discrete fourier transformation (DFT). However, their proposed technique cannot support CONV layers with

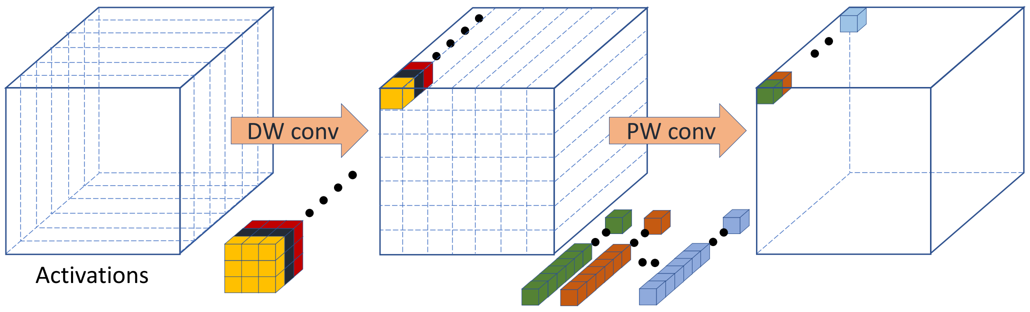


FIGURE 2.5. Depth-wise separable convolution layer

$stride > 1$ or PW and FC layers. Ding et al. [110] recommend a block-wise circular constraint approach to accelerate FC layers by converting the computation to the frequency domain. This technique can be applied to other CONV layers as well. The two mentioned approaches have drawbacks of involving complex number arithmetic. To address this issue, Winograd et al. [111] proposed a 2D convolution implementation using real number multiplication, which is optimised for 3×3 convolution implementations [26, 112]. As analysed in reference [92], these techniques can improve the overall deployment performance up to 2-3$\times$.

This brief review on the DNN kernel innovations highlights the necessity for having flexible hardware accelerators that can host different computation models with diverse characteristics. This is particularly crucial for embedded accelerators where slightly sacrificing the accuracy by simplifying the computation kernel is more considerable. Table 2.1 provides a summary of the architectures employed in a number of the recent state-of-the-art embedded DNNs. The last row underscores that standard, DW, PW, and FC layers account for almost all MACs operations.

2.1.4.2 Data Representation and Quantisation

Choosing an efficient data representation is a major design decision for deploying any algorithm, particularly in memory and compute-intensive cases, where boosting performance by reducing implementation costs is often achievable by utilising less costly alternative data types. Improving speed often requires sacrificing accuracy, and so enables a range of trade-offs to explore. When it comes to DNNs, the computation model offers many opportunities for optimisation. On the one hand, DNNs need an enormous memory footprint, complex data

TABLE 2.1. Summary of the architectures employed in a number of state-of-the-art embedded DNNs

Metrics	**NASNet-A (4@1056)[113]**	**MobileNetv2 1×[108]**	**ShuffleNetv2 1×[114]**	**SqueezeNet [22]**
Top-1/5 error	26% / 8.4%	28% / -	30.6% / -	42.5% / 19.7%
# of CONV Stages	22	20	20	14
# of SConv / Filter sizes	800 / 3	32 / 3	24 / 3	1376 / 3,7
SConv MACs / Parameter (% Total)	3.5% / 16.8%	3.4% / ∼0%	5.7% / ∼0%	72.1% / 45.5%
# of DW Conv.s	15290	7136	2426	0
DW Conv. kernels / input / channel / Strides	3,5,7 / 7-57 11-176 / 1,2	3 / 3-112 32-960/ 1,2	3 / 7,14,28 24-232 / 1,2	- / - - / -
DW Conv. MACs / Parameter (% Total)	14.1% / 5.4%	6.5% / 1.9%	2.7% / 1.0%	- / -
# of PW Filters / Channel Depths	18465 / 11-1056	9920 / 16-960	5572 / 24-1024	2600 / 16-512
PW Conv. MACs / Parameters (% Total)	**79.6% / 63.3%**	**88.7% / 61.2%**	**89.1% / 53.7%**	**24.5% / 54.5%**
Global Pool Size	3×3	7×7	7×7	13×13
FC MACs / Parameters	∼0.8M / ∼0.8M	1.3M / 1.3M	1M / 1M	- / -
SConv+PW+DW+FC MACs (%)	**97.63**	**99.03**	**98.28**	**96.68**
Total MACs / Parameters	564M / 5.3M	300M / 3.5M	141M / 2.3M	833M / 1.25M

movements, and numerous computation resources when the costs for all these directly depend on the employed data types. At the same time, DNNs are robust against using low-precision data types and aggressive quantisation, especially during the inference task, primarily because of using very large and over-parameterised DNN architectures [115]. These two sentiments led to extensive research on optimised DNN deployments [116, 117, 112, 118], co-designing new DNN architectures [119], and efficient training [120, 121, 122] by applying standard and customised data types on all aspects of data flow.

First and foremost, each data representation comprises unique characteristics, such as quantisation accuracy, dynamic range, etc [115], that may fit some algorithms but not all. *Floating-point (FP)* arithmetic encodes real numbers approximately, using three components: 1) a fixed-point value, called Mantissa ($\hat{M}$) with M bits, 2) an E-bit integer exponent ($\hat{E}$) of a fixed base, and 3) a sign bit ($\hat{S}$), that all together form a bit vector $(s, e_{E-1}, ..., e_1, e_0, m_{M-1}, ..., m_1, m_0)$ (Figure 2.6A) that represents the real value of

$$(-1)^{\hat{S}} \times (1.\hat{M}) \times 2^{\hat{E}}. \tag{2.5}$$

TABLE 2.2. Various floating point configurations.

Data representations	**Exponent**	**Mantissa**	Dynamic range[†]	Precision[‡]
FP32 [123]	8	23	1668	2^{-24}
FP16 [123]	5	10	241	2^{-11}
BF16 [14]	8	7	157	2^{-8}
FP8 [124]	4	5	114	2^{-6}

[†] $20\log_{10}^{\frac{max^+}{min^+}}$, $max^+ = (2 - 2^{-M}) \times 2^{2^{E-1}-1}$, $min^+ = 2^{-M} \times 2^{-(2^{E-1}-2)}$
[‡] relative round-off error, i.e. 2^{-M-1}

This format provides precision and dynamic range, respectively, through mantissa and exponent parts, and so, by sizing them, different data type instances with different characteristics could be driven, such as configurations listed in Table 2.2. Although these standard floating-point formats offer an accurate way to represent a wide range of numbers, the hardware cost and memory overheads associated with standard floating-point formats are extensively high. In fact, floating-point arithmetic requires extra circuitry to handle signs, pre-shift, post-shift, rounding, and special numbers (eg. denormals, Infinity, NaN, zero).

Fixed point arithmetic is an alternative to floating point. This representation operates similarly to integers, while a portion of the representation is assigned to the fractional part of real numbers, called F fractional bits (Figure 2.6B). This is equivalent to scaling an integer by the factor of 2^{-F}. From another perspective, fixed point numbers are similar to floating point numbers without the fixed based exponent section, which causes fixed point arithmetic to be limited in dynamic range. Like integers, the numeric range is uniformly quantised, meaning gaps between successive numbers are the same throughout the representation. While larger F translates to smaller gaps, and so, higher precision, this reduces the numeric range. Putting it all together, a bit vector $(x_{B-1}x_{B-2}...x_F.x_{F-1}...x_1x_0)_2$ in fixed-point, represents the real value given by:

$$(x_{B-1}x_{B-2}...x_F.x_{F-1}...x_1x_0)_2 = \sum_{i=0}^{i=B-1} x_i \times 2^{i-F}. \tag{2.6}$$

The main advantage of using fixed point arithmetic is a simpler implementation than floating point. Fixed point implementations are smaller, faster, and less energy-demanding. To

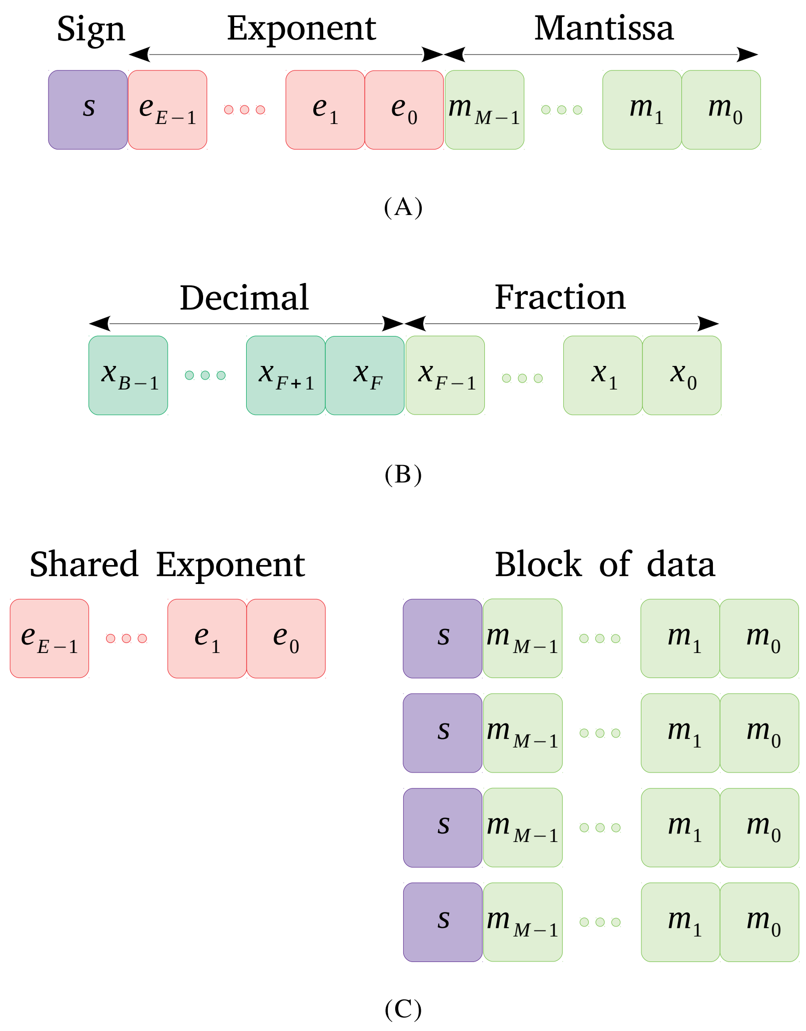


FIGURE 2.6. The generalised schematic for (A) fixed-point, (B) floating-point, (C) block floating-point representations.

elaborate, Table 2.3 compares the area and energy cost of different data type addition and multiplication as the dominant operations in DNNs. The post-synthesis area utilisation numbers are reported by William Dally [125] using Design Compiler under TSMC 45nm tech node, and the energy values are the rough energy costs for the studied 8-core superscalar processor by Horowitz [126]. The last column approximates the costs for the corresponding MAC unit, assuming no circuit fusion. As it shows, the energy consumption for a full precision floating point MAC unit is about $4.6pJ$, that is $1.4\times$ and $15.3\times$ the corresponding measurements for INT32 and INT8, while the same comparisons for the area utilisation are

$3.3\times$ and $28.4\times$. A similar trend is almost true for other platforms, such as FPGAs, which are the focus of this thesis. Reference [11] compared the implementation of MAC units with different word lengths on Xilinx and Intel FPGAs. They reported using fixed point 8×8-bit operations instead of single precision floating point, logic resources are reduced by $10 - 50\times$. Since FPGA resources are limited, smaller computation units means more can be used for parallelism for a given device.

The main advantage of using fixed point arithmetic is having simpler implementations than floating point numbers. This means fixed point computations are smaller, faster, and less energy-demanding. To elaborate on this, Table 2.3 compares the area and energy cost of different data type addition and multiplication as the dominant operations in DNNs. The post-synthesis area utilisation numbers are reported by William Dally [125] using Design Compiler under TSMC 45nm tech node, and the energy values are the rough energy costs for the studied 8-core superscalar processor by Horowitz [126]. the last column approximate the costs for the corresponding MAC unit assuming no circuit fusion. As it shows, the energy consumption for a full precision floating point MAC unit is about $4.6pJ$, that is $1.4\times$ and $15.3\times$ the corresponding measurements for INT32 and INT8, while the same comparisons for the area utilisation are $3.3\times$ and $28.4\times$. A similar trend is almost true for other platforms, such as FPGAs, which are the focus of this thesis. Reference [11] compared the implementation of MAC units with different word lengths on Xilinx and Intel FPGAs. They reported using fixed point 8×8-bit operations instead of single precision floating point, logic resources are reduced by $10 - 50\times$. Regarding the fact that FPGA resources are limited, the smaller computation unit means more computation units using the same area.

Besides the computation unit, the overall performance and implementation costs also depend on memory structure and the complementary data movements, where data representation affects both. This is crucial as the majority of the energy consumption happens accessing the data. Dally [125] has reported the energy cost for accessing data from different memory types. As measured, reading a word (32 bits) from a small (a few KB) local SRAM block, an On-chip SRAM (within a few MB), and an off-chip DRAM require $5pJ$, $50pJ$, and $640pJ$ amount of energy, respectively, that shows the significance of data access in comparison to

TABLE 2.3. The implementation cost comparison for some fixed point and floating point data representations in 45nm technology size. The numbers are picked from [126] and [125].

Data representations	**Add**		**Multiply**		**MAC**	
	Area (um^2)	**Energy** (pJ)	**Area** (um^2)	**Energy** (pJ)	**Area** (um^2)	**Energy** (pJ)
INT8	36	0.03	282	0.2	419†	0.3†
INT16	67	0.05	-	-	-	-
INT32	137	0.10	3495	3.1	3632†	3.2†
FP16	1360	0.40	1640	1.1	3000‡	1.5‡
FP32	4184	0.90	7700	3.7	11884^{II}	4.6^{II}

† accumulation in INT32

‡ accumulation in FP16

II accumulation in FP32

compute units. Thus, shrinking the data type size substantially reduces the overall energy costs.

To have the best of these two worlds, block floating point arithmetic [127, 128] is suggested that incorporates a shared exponent value for a block of (fixed point) data, where each entry is encoded by a single bit sign and a M bit mantissa value (Figure 2.6C). In comparison to fixed point numbers, the shared exponent allows a higher dynamic range for the whole block, while the difference between entries is still limited to the fixed point structure of the entries. However, the achieved dynamic range does not cost much, like floating point arithmetic, where each number has its own exponent. The main challenge for using block floating point is defining the block size, which depends on the application.

In the case of DNNs, both training and inference tasks rely on repeated calculations involving very large and very small numbers. However, the training task is more sensitive to quantisation error while requiring a much wider dynamic range to remain stable and keep converging to the optimal state. Because of that, most early commercial platforms, such as Nvidia GPUs, employ standard floating-point representations that offer a wide dynamic range and are known to perform well in mathematically intensive applications [129]. In recent years, the research focused on high-performance training platforms using TF32 [130], FP16 [121, 122], BF16 [14], and FP8 [128, 124, 131, 132] arithmetic. At the same time, the hardware companies pushed to natively support such data types [133, 134, 135]. There are also many signs of progress going further to use INT8 [136, 137], which these methods are often limited to simple NN models for relatively easy learning tasks while requiring hyperparameter tuning. Block floating point [138] and mini block floating point [15] are other two alternative data representation that has shown promising training results.

In contrast, the DNN inference task is much more resilient against low-precision arithmetic. This was a critical finding as inference mostly runs on embedded devices, where computation and memory resources are not abundant. In the early days of DNNs, Qiu et al. [36] studied the numerical distribution of DNN models and concluded the dynamic range for both weight parameters and activations in each layer is generally limited while it varies from layer to layer. Later on, Guo et al. [92] analysed different ImageNet [99] classifiers and showed applying linear quantisation on trained networks using 8 bits or higher causes insignificant accuracy drop while by choosing 6 bits or lower the accuracy degradation will become noticeable.

These promising results motivate researchers to explore novel techniques to improve the benefits from quantisation while preventing the accuracy drop. The next step was adding granularity to the quantisation process, as the distribution for the activations, weight parameters, and temporary results are also different. Qiu et al. [36] proposed an algorithm based on linear quantisation defining fraction bit-width for each layer, separately for weights and activations. Later, Guo et al. [139] suggest fine-tuning after post-training quantisation to recover the full-precision accuracy. Next, Jacob et al. [140] proposed involving the quantised inference during the training, called quantisation-aware training, that enhances the training results significantly. Many studies point to quantisation as a DNN architecture flexibility degree and tried to find better quantisation settings through NAS techniques [141, 142, 143, 144, 145].

Because of all the above, the efficient DNN deployment using fixed-point arithmetic became a common practice [20, 146, 112, 118], which substantially reduced the deployment costs, particularly as a wide range of hardware accelerators at the time natively supported various fixed point arithmetic, especially INT8, such as Nvidia Volta GPU [147], Google TPU [148], etc. This trend particularly benefited the FPGA-based accelerators, where the designs accomodate customised compute units with any arbitrary set of precisions [38, 149, 117]. Eventually, the linear quantisation idea has been taken to its conclusion by proposing ternary and binary quantisation which achieve extremely high speed and low energy implementations [29, 150, 17, 151, 152, 153, 154].

The main reason for pursuing linear quantisation was and is the compatibility with general-purpose hardware accelerators. However, uniform quantisation does not offer the best match for value distributions in DNNs. Nonlinear quantisation is another approach to map real values to a smaller set of values, while in contrast to linear quantisation, there is no mandate over the quantisation levels. This enables customised quantisation schemes to fit any arbitrary data distributions. Such mathematical advantage comes at the cost of more complex arithmetic, where sometimes the only way of implementation is value mapping through look-up tables. For instance, Chen et al. [155] proposed a trainable look-up table for each layer, in which the weight parameters are stored and accessed using a hash function. This limits the memory footprint for the weight parameters by only storing the hash code, rather than the actual high-precision values. On top of that, Han et al. [33] recommended applying weight value clustering that enhanced the compression up to 4-bit code for weight values without accuracy drop. Afterwards, Samragh et al. [35] observed how limited the nonlinear quantisation levels are and so proposed an alternative to MAC units, called factorised coefficient-based dot product memories, which include the MAC operation result for each possible weight and quantised activation input calculated at compile-time.

Due to the fact that nonlinear quantisation methods require customised data paths, customisable platforms such as FPGAs became a beneficiary of these innovations since state-of-the-art FPGA architectures integrate massive numbers of distributed look-up tables. Similar to uniform quantisation, there are some works on very low-precision nonlinear quantisation, such as trainable ternery-quantised networks proposed by Zhu et al. [19] where the positive and negative scales are flexible to be trained separately. This simple modification enables near floating point precision for the ImageNet [99] data set (by recovering the accuracy drop from just ternerising [156]) while achieving 16x model size reduction and 2% accuracy degradation on ImageNet. There are also some works on hardware-oriented nonlinear quantisation schemes where the available computation resources inspire particular quantisation patterns which are well matched to the hardware and the DNN value distribution also matches the data. One example is AddNet [18], based on reconfigurable constant coefficient multipliers inspired by FPGA high-performance logic element architectures.

2.1.4.3 Data Movement and Memory Management

Despite all progress toward efficient DNNs, the state-of-the-art network models still require extremely large numbers of parameters [157]. Modern DNN accelerators such as the Nvidia Turing GPU series [158] and Google TPUs [159] can not accommodate all parameters on-chip in many cases. One solution is folding the computation, introducing significant data movement between on-chip and off-chip memories to supply the compute units and store the results continuously. Such an approach requires high memory bandwidth and, more notably, contributes significantly to the overall energy consumption as the energy cost for data movement using the existing technologies exceeds the energy consumption for the actual computation [160, 28].

To elaborate on this, consider a fixed point MAC unit that multiplies in 8 bits and accumulates in 32 bits. This operation reads three operands (in total six bytes) and eventually writes back the result word. As Horowitz [126] reported for the $45nm$ technology size, the energy required for only reading the inputs from external DRAM will be in the range of $1.5 - 3nJ$. That is three orders of magnitudes higher than the actual operation energy cost as approximated in Table 2.3.

Although this generally happens for both server and end-point deployments, it becomes more crucial for embedded devices with much lower memory and compute budgets. Therefore, DNN accelerators must carefully orchestrate the data across compute units, on-chip memories, network-on-chip, and eventually external network links or memory blocks to minimise the overall energy costs, i.e., the DNN data flow should maximise data reuse for all involved parameters through an efficient memory hierarchy that prioritises local accesses [32].

There are two major classes of data flow in hardware accelerator architectures, 1) temporal and 2) spatial [161]. Both architectures utilise many processing units to handle the computations.

Orchestrating transfers between the memory hierarchies and processing units is the main differentiating characteristic that affects both control logic and data movement circuits.

In the temporal architecture, the processing elements are simply an arithmetic logic unit (ALU) without any local memory block, as a centralised control unit manages the flow between the memory hierarchy and PEs. Thus, PEs do not communicate with each other directly. In this method, flexibility for supporting various computations is on the shoulder of memory-PE connections with broadcasting, multi-casting, or reducing [32]. CPUs and GPUs are good examples of temporal architectures, where in both architectures, SIMD operations are centrally managed by control units.

In spatial architectures, some aspects of the control logic are distributed in each PE. Therefore, each PE has its own control unit and a memory to store required data. Also, PEs are interconnected in a spatial arrangement and can communicate directly and pass data locally to each other. Such an interconnection structure enables low energy-cost local data access from PE to PE, as opposed to temporal architectures, where all data access goes through centralised memory blocks. Another advantage of spatial structures is they can significantly reduce the required memory bandwidth by better reusing data.

Although both ASIC and FPGA platforms just provide compute, memory, and logic resources to implement any arbitrary circuits, most ASIC-and FPGA-based DNN accelerators use spatial architectures [162]. This is because DNN algorithms offer enormous data reuse opportunities. On the contrary, the main challenge to designing a spatial data flow is to find a cost-effective interconnection pattern that is flexible enough to efficiently support various computations. Referring again to Table 2.1, the state-of-the-art efficient DNN models exploit significantly different data flows, reuse opportunities, and memory and compute resource requirements.

Data reuse depends on the computation characteristics. DNN algorithms are primarily based on nested loop models having an inner MAC operation that enables four ways to harness reuse: 1) reusing a filter value for different input feature map entries, 2) reusing an input feature map element for different kernels, 3) reusing the partial MAC output values, and 4) reusing input entries by sliding, that is dedicated only to convolutional layers [32]. An accelerator can employ a set of these reuse techniques to keep the data in local memories and avoid going back to the DRAM memory. Based on these reuse opportunities, the state-of-the-art spatial DNN accelerators can be categorised into the following categories:

- Weight-stationary such as the TPU [148] and [163], in which the dataflow stores the filter weight values inside PEs using small register files, while inputs and the partial sum are streamed between PEs.
- Input-stationary like SCNN [164], where the input feature map entries stay on PE arrays, weights and partial sums are delivered to the PE, respectively from memory and PE-PE interconnections.
- Output-stationary like Origami [165] and [166] that keep and accumulate partial sums in each PE until the accumulation is finished. Meanwhile, the input feature map and weight values are passed by PE-PE interconnections for more data reused.
- No local reuse such as Nvidia Tensor Core architecture [135], where neither input feature map, weight, or partial sums remain on each PEs, but they flow from each PE in different directions to other PEs in order to be reused.
- Row-stationary that is proposed by Eyeriss [28] and Eyeriss-v2 [162], where the dataflow utilises weight reuse, input feature map sliding, and partial sums. By involving reuse in multiple data dimensions, this approach offers the highest energy efficiency.

2.2 FPGA Architectures for Efficient DNN Deployment

This section first demonstrates why FPGA technology has remarkable advantages over other hardware accelerator platforms for embedded DNNs. Background on FPGA architectures and their chronological architecture evolution are then described.

2.2.1 The FPGA Advantages for Embedded DNN

Choosing the right hardware to deploy a given algorithm is always challenging. This is because each application demands a set of considerations when sometimes one characteristic is only achievable at the cost of sacrificing other features. For instance, integrating domain-specific circuits enhances the performance of a particular set of applications, whereas it adds overheads to other workloads that do not benefit from it. Similarly, providing programmability and reconfigurability broadens the supporting algorithms and data paths (i.e. generality) while the added flexibility limits the performance and efficiency.

Table 2.4 present a comparison between the state-of-the-art accelerator platforms, where each technology offers a unique specification portfolio, while there is no solid supremacy among them. As shown, the differences are not limited to performance metrics and include production challenges and application-specific considerations. Ultimately, the most restrictive constraints dictate which hardware to use.

CPUs are the most generalised systems where users can execute any given algorithm through a pre-defined software instruction set. Although such generalisation is advantageous when dealing with various workloads, it comes at the cost of limited performance and substantial energy consumption. This is an issue for many compute-intensive applications, including DNNs, that offer abundant parallelism and data reuse, while CPU architectures can not benefit from it. To address this shortcoming, the state-of-the-art CPU systems have 10s of cores/threads, each incorporating vectorised units (such as AVX-512 units in Intel CPUs [172] that allow 32 pairwise INT8 multiplications and accumulate adjacent groups of 4 into 8 INT32 results.). However, the offered peak performance is still limited to 10s of TOPs. On the other hand, the main advantage of employing CPU systems is the ease of use, where the hardware is available off the shelf, they are easy to program, and are accompanied by a series of excellent development tools.

In contrast to CPUs, GPUs were initially designed as specialised ASIC chips to accelerate graphical rendering workloads [173]. Over time, due to the demand for parallel computing tensor-based computations (such as linear algebra subroutines [70]) for batched data, the GPU

TABLE 2.4. A comparison between the state-of-the-art accelerator platforms

	Specification	**CPUs ([167])**	**GPUs ([158, 168])**	**ASIC ([159, 169])**	**FPGA ([170])**
Computing architecture	Generality	Turing-complete		Specific domain	Any custom HW
	Data path parallelism	Vectorisation, few cores/threads	SIMD, many cores/threads	Design specific & fixed	Spatial Architecture
	HW specialisation	Fixed general datapath & memory subsystem		Full flexibility for specialisation up to fabrication	Reconfigurable
Performance†	Peak throughput‡	Low, 1-10s TOPs	High, 100-1000s TOPS	High, 100-1000s TFLOPS	High, 10-100s TOPS
	Latency	High		Low	
	Power consumption	High power		Most efficient	Moderate
	Energy efficiency	Very Low	Low	Most efficient	Moderate
Production challenges	NRE Cost	Off-the-shelf		Costly lithography	Off-the-shelf
	Unit price	Moderate	High	Low	High
	Ease of programming	Software (compilers & libraries)		Customised software stack	RTL / HLS
	Time to market	Low	Moderate	High	Moderate
In the lens of embedded DNNs	Arithmetic precision	High-precision FPs and fixed-points up to BF16 [134] and INT4 [158]		Design specific & fixed	Any arbitrary quantisation
	Data movement	Multi-level memory hierarchy		Design specific & fixed	Customisable
	Interface integration	Limited IOs, access through DMA, fixed memory hierarchy		Design specific & fixed	Customisable
	Inference	Poor performance, Great for debugging	High-throughput, not energy-efficient, better in large input batch sizes	High performance, energy-efficient (if the algorithm matches the data path)	High throughput, low latency, customisable data path Great for low batch sizes
	Training	Poor performance	Great production-ready platforms (like Nvidia GPUs [158] and TPU [159])		Not-efficient yet [171]

† For sufficiently parallel applications

‡ The reported ranges are for INT8 MAC accumulated in INT32

architectures became more general-purpose parallel processors (also known as GPGPUs) by offering programmability through a wide range of SIMD instructions that execute over thousands of optimised and specialised computing cores in parallel, while still delivering efficient rendering. Integrating these specialised but flexible circuits for parallel processing coupled with an energy-hungry multi-level hierarchical cache system limits the energy efficiency, particularly when the target algorithm does not perfectly match the data path or offers higher parallelism and data reuse patterns that GPU architecture can not benefit [174]. Despite this issue, GPUs are currently the best choice for DNN inference acceleration and training tasks due to their accessibility, ease of use, and high performance. Recent high-end Nvidia H100 series GPUs incorporate a specialised tensor core architecture [175] that enables an impressive maximum throughput of 989 TFLOPS and 1979 TOPS, respectively, for MAC operation in BF16 and INT8, at a power consumption of 700W [168].

Employing a customised ASIC chip is another acceleration option, where the designer can harden a specific data path targeting pre-defined targeted applications. The main advantage of this technology is the level of HW specification that leads to maximum performance and efficiency. However, designing an ASIC is extremely costly in time (usually years) due to the costly lithography fabrication process that is only economical for the highest volume production. Also, as post-fabrication alteration is not possible, designers end up continually fabricating new versions to maintain or enhance performance for the latest workloads. This is an important issue for algorithms that change frequently, e.g. DNNs.

Recently generalised domain-specific ASIC chips such as Google TPUs [159], GroqChip Processor [169], and Graphcore IPU [176] have been able to offer acceleration over a large class of DNNs while delivering high-performance and energy-efficiency. The primary disadvantage of this approach is the high level of specialisation that may not match the desired algorithm, i.e. the ASIC designer has to sacrifice generality to gain performance. Although some ASIC manufacturers provide open source APIs such as PyTorch [177], TensorFlow [178], etc., the developer still should deal with customised software stacks, especially for implementing customised kernels or enabling special hardware features. Eyeriss [28], FlexFlow [179], and Bit-Fusion [180] are some ASIC DNN accelerator examples from academia.

In a sense, GPU architectures and ASICs present two extreme sides of the wide spectrum of accelerator platforms. Respectively, they approach the problem with a philosophy of generalisation and specialisation, respectively. Correspondingly, these goals cause GPU architectures to sacrifice energy efficiency for generality, and in ASICs, to require constant upgrades to keep up with evolving DNN algorithms. To alleviate this issue, commercial platforms have pursued solutions such as application-optimised GPU architectures and more generalised ASIC datapaths [175, 176]. Despite the differences, both of these observations are because both technologies aim for a fixed data path without post-fabrication alteration. This raises the necessity for a new platform that offers ASIC-like data path specialisation with high performance and energy efficiency and GPU-like generalisation, where any arbitrary calculation could be effectively mapped to the hardware.

Reconfigurable architectures, such as FPGAs, address this gap by offering a unique combination of generalisation and specialisation. Their underlying approach is to utilise flexible substrates that implement myriad circuits efficiently. Like hardware specialisation in ASICs, FPGA platforms come with EDA tools to map any given datapath into the available functional-reconfigurable resources. In contrast, the user can reconfigure the FPGA fabric many times to upgrade the datapath or to fix bugs in a fraction of the ASIC design flow at virtually no cost. However, due to the flexibility overheads, the final circuit is not as performant and efficient as the ASIC equivalent. In comparison to GPUs, the reconfigurability makes FPGA advantageous. Instead of hardening a generalised data path, resources can be re-routed to form the desired data path aiming for more efficiency and higher performance.

Although all the abovementioned accelerator platforms have merits in DNN acceleration, currently, GPU and ASIC-powered platforms are the most common, production-ready, and high-performance training and inference engines. However, within the lens of embedded DNNs, FPGAs offer a unique set of features. As elaborated in Section 2.1.4, embedded DNN algorithms are well known for using special kernels with irregular parallelism and data reuse patterns, customised quantisation, etc., which change from time to time. However, due to the hardened datapath in CPUs, GPUs, and ASICs, the hardware design process is always guided by DNN model benchmarks and not the actual end-user algorithm. To maximise the gains, the hardware should support data path specialisation for any optimisation, including arithmetic precision, parallelism, data reuse schemes, circuit pipelining, and memory structures, etc. This should be combined with a reasonable design flow that enables new DNN models to be utilised, e.g. translators from DNN descriptions [181, 29] and parameterised domain specific processor overlays [30, 151]. Finally, the hardware flexibility in FPGAs offers the opportunity to co-design the algorithm and datapath enabling optimisations in computing architecture and algorithm dimensions to achieve higher performance and more efficient resource allocation [31].

Reconfigurable IOs is another advantage of FPGA architectures. Devices with any customised interface can directly connect to the FPGA IOs, which can reduce latency of the data transfer between the FPGA and external devices. This enables near-real-time processing, critical for some applications, including sensing and controlling systems in autonomous vehicles and robots. In contrast, GPU and CPU memory structures require accessing external devices through the DMA mechanism, which adds significantly to the overall latency.

As a final note, practical systems often require the execution of multiple tasks, each with different characteristics. The authors speculate that future systems will become more heterogeneous because they will be required to integrate general-purpose, domain-specific, and reconfigurable circuits, providing an improved global solution than could be provided by any single technology.

2.2.2 FPGA Architectures and Opportunities

Reconfigurable architecture technology emerged from the commercialisation of programmable logic devices (PLDs) [182] that incorporate memory cells to set the functionality of initially undefined-function logic units. The gradual advancement in the logic unit structures and their functional flexibility resulted in various PLD classes, starting from simple PLDs (SPLDs) that comprise programmable array logic (PAL) [1] and programmable logic array (PLA) [2] circuits [182], both providing flexible structures to implement sum-of-products. Later, complex PLDs (CPLDs) scale the integration of PLAs into arrays by providing a programmable interconnection scheme and distributed clocked registers [183].

FPGAs are the most functionality-rich architectures among reconfigurable architectures [184]. After decades of FPGA architecture research, commercial FPGA architectures have evolved to comprise both fine and coarse-grained reconfigurable blocks arranged in a columnar fashion. The most basic units are LEs which are built from LUTs and additional logic, such as adders and flip-flops. For higher area efficiency and speed, commonly-used circuits such as memories, DSP blocks and microprocessors are implemented as coarse-grained EBs. A flexible interconnection network is used to connect LEs and EBs to IO blocks which include general IO, memory interfaces, and transceiver blocks. Together these form a programmable system on a chip that can implement arbitrary circuits [185]. In addition to the described

[1]A programmable AND gate array linked to a programmable OR gate array
[2]A programmable AND gate array linked to a fixed OR gate

common theme, each FPGA company suggest diverses a series of FPGA architectures, where FPGA resources and interconnections are generalised or specialised according to particular benchmark sets.

In this thesis, we study the architectures of common FPGA compute resources, as known as logic element and DSP blocks, which are respectively categorised as fine and coarse-grained FPGA blocks. A brief background for these two blocks in different commercial architectures is presented below. A more detailed treatment is available in reference [184].

2.2.2.1 Logic Blocks:

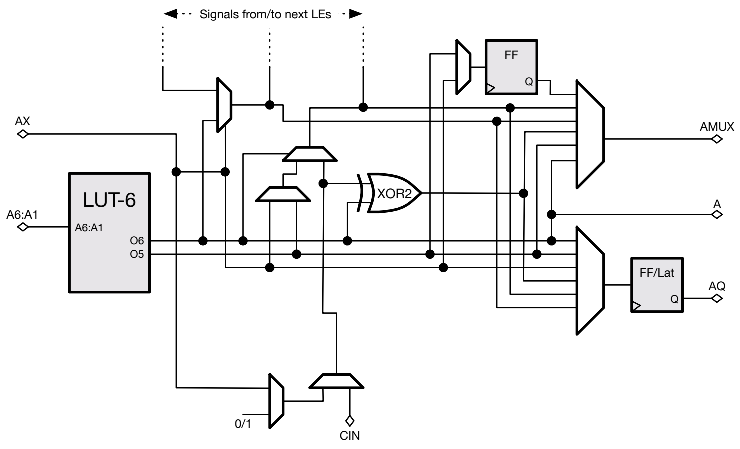
Reconfigurable architectures are fundamentally designed to provide generalised components that can implement a wide range of functions. After the success of CPLDs in scaling PLA units with a flexible interconnection fabric, Xilinx introduced a more area-efficient LUT-based logic unit that later became the primary component of modern FPGA architectures. A M-LUT can act as any M-input Boolean function by storing its truth table in reconfigurable SRAM cells if the M input signals connect to memory multiplexer select lines to output the corresponding value from the truth table. After connecting a bypassable flip-flop register to the LUT output for optional output registering, we form an FPGA basic LE [186]. To mitigate the interconnection costs in scale, LEs are implemented in clusters of N, called logic blocks (LBs), where within each group local interconnections are highly flexible with low latency [187], while LB-LB interconnections are simplified to mitigate the transmission delays.

Ahmed et al. [188] studied tradeoffs associated with this generic structure. Increasing M enhances the functional expression of LEs, at the cost of linear and quadratic growth for circuit latency and area costs, respectively. Also, the parameter N relies on another tradeoff between the richer localised connectivity vs a less expensive routing scheme. As shown empirically, 4-6 input LUTs clustered in groups of 3-10 LEs deliver the best-performing configurations, measuring the area-delay product. This outcome does match the architectural trends in commercial FPGAs. For instance, Xilinx FPGAs initially started with 2-member clusters of 3-LUTs (XC2000 series) and gradually upgraded to 6-LUT in the group of 4 Xilinx Virtex-5 architecture.

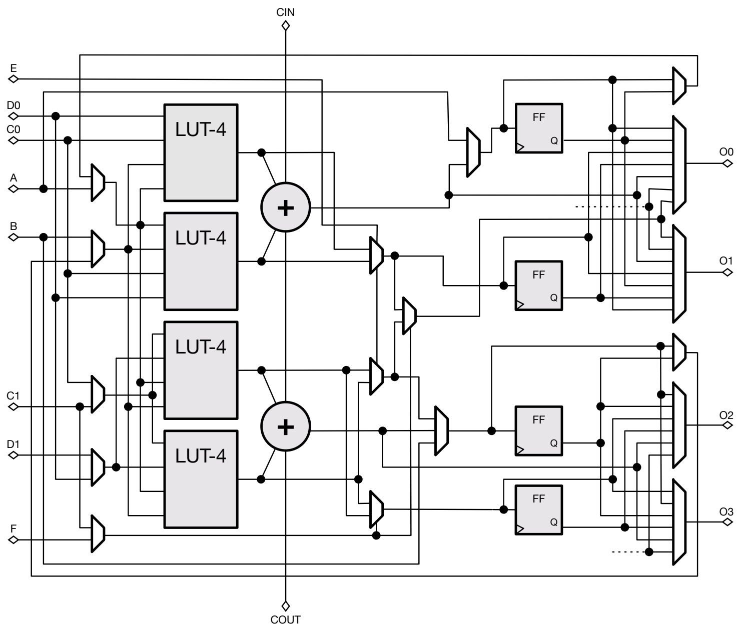
Fixing the size of LUTs over the entire architecture leads to inefficiencies, including underutilisation of large LUTs or gaining lower performance when M is small. Ahmed et al [188] observed in an experiment that using 6-LUTs instead of 4-LUTs delivers 14% higher performance in exchange for 17% more chip area. Another study [189] reports for a 6-LUT-based architecture that only about one-third of LEs were configured in 6-input mode. As a practical solution, fracturable LUT structures were proposed by Altera and initially commercialised in Stratix II architecture [190]). These enable decomposing large LUTs into multiple smaller ones where each can implement a function. Due to the interconnection limits, the decomposed LUTs may have shared inputs. To potentially register all outputs of decomposed LUTs, the number of flip-flops per LE also increased. In the case of not decomposing a LUT, the extra registers could be allocated for pipelining purposes, such as in deeply pipelined implementations. As an example, in Xilinx Virtex-7 architecture [191], a 6-LUT can decompose into two 5-LUTs while the inputs are shared, or two independent LUTs with the input size of 3 and 2 or less.

Another major evolution in LEs structures was integrating hardened adders and carry chain circuits for efficient native support for adders, compressor trees, and other multi-input circuits due to their substantial presence in almost all applications. Mapping a portion of logic to the relatively smaller hardened circuits and capturing potential LUT-LUT interconnections using the fast dedicated LE-LE interconnections will lessen the demand for LEs and routing resources and significantly enhance the overall performance.

CLB [192] and logic array block (LAB) are, respectively, state-of-the-art LB architectures from two major FPGA vendors, Xilinx and Intel. A CLB is composed of two slices, which are the basic unit of the FPGA's soft-fabric. Each slice is composed of four 6-input LEs, including a fracturable 6-input LUT and additional circuitry such as registers and multiplexers, which give the slice its expressiveness. Figure 2.7A shows a quarter of the slice architecture (a 6-input LE and the corresponding circuits) found in the modern Xilinx UltraScale+ FPGAs. Another notable feature of the slice is the presence of a fast carry-chain between the LEs, which is often used to implement arithmetic circuits such as ripple carry adders or compressor circuits.



(A) Xilinx UltraScale+ LE. A slice is composed of four LEs.



(B) Intel Stratix-10 ALM. Each ALM has 8-inputs and a fracturable 6-LUT [42].

FIGURE 2.7. Basic LE for Xilinx and Intel FPGA architectures.

In contrast, the basic LE in Intel's architectures is called an adaptive logic module (ALM) [42]. Figure 2.7B shows the ALM architecture of a modern Stratix-10 device. Each ALM is composed of a fracturable 6-input LUT, while primitives such as full-adders and multiplexers help to support higher-order boolean functions. Ten ALMs on Intel FPGAs are grouped to form a LAB, which augments the ALMs with more primitives such as HyperFlex registers, local interconnect, and configurable carry-chains [42].

In regard to embedded DNNs, the architecture of the LE plays a significant role, as low-precision and customised quantisation arithmetic units generally will be mapped into them. Therefore, some recent studies proposed various ways to upgrade commercial LE structures for MAC, XnorPopcount, and dot-product operations, which indicate the predominant operations in embedded DNNs. Boutros et al. [193] proposed a novel dual carry chain for ALMs that effectively reduces the overall chip area in embedded low-precision DNNs. Kim et al. [194] tackle the issue by integrating an extra sum chain, beside ALM carry chain, to efficiently implement pop-counting circuits for binarised DNNs achieving $2\times$ area reduction while adding 2% overhead on the operating frequency.

The three abovementioned operations are also members of a more generalised arithmetic class, called parallel digital arithmetic, that have been studied since the 1960s [46, 47, 48]. This was well before LUT-based compressor and parallel counter circuits became popularised in the past two decades primarily from work by Parandeh-Afshar et al. [195, 50] and Kumm et al. [196, 197]. In [50], the authors proposed architectural changes to the Intel ALM carry-chains such that large compressors like (6:2) and (7:2) can be efficiently mapped to single ALMs. Although their proposed compressor is very efficient, for modern applications such as XnorPopcount in binarised neural networks (BNNs) [198], these compressors would be significantly underutilised. In Chapter 4, a series of modifications to the LEs of both Intel and Xilinx architectures have been proposed that substantially optimise the implementation of multi-input circuits.

Recent work on LE structures by Rasoulinezhad et al. [2] enhanced the support for low-precision multi-operand operations such as dot-product operations that is described in detail in Section 4. Similar work by Boutros et al. [193] also suggests LE-level modifications that offer better support for low-precision adders and multipliers. More comprehensive details of this research work is presented in [199].

2.2.2.2 DSP Blocks

Adding specialised circuits to FPGA architecture is always challenging as it is contradictory to the philosophy in FPGAs of keeping the architecture as general as possible. Moreover, they result in wasted area if unused. However, ubiquitous operations such as additions, multiplications, and fused together as MACs provoke designers to harden such circuitries in configurable EBs.

In the early days, FPGA platforms were dominantly employed in communication systems, signal processing devices, and data acquisition tasks that heavily relied on high-precision arithmetic. This led to DSPs that offer a set of costly high-precision logic and arithmetic with high performance while it captures all required LUT, registers, and routing resources for the equivalent implementation by LEs, in a relatively smaller area.

The first DSP generations were commercialised by Xilinx in Virtex-II [200] architecture that simply incorporates an 18×18 multiplier with registered input and output ports. This precision was particularly chosen to match the optimised FIR filter implementations [201] and the distributed memory block (called Select RAM) structures [200]. In contrast to Xilinx dedicated multipliers, the first DSP block from Altera was a decomposable multiplier architecture followed by a flexible multi-input accumulation unit. This architecture is based on four 18×18 multipliers, where each can be decomposed into two 9×9 separate multipliers. All four together with an extra carry chain adder can be used to form a 36×36 multiplier. Thus, this block can handle one 36×36, four 18×18, or eight 9×9 multiplications or MACs. Another feature in this architecture was providing dedicated streaming connections for input registers that offer efficient data movement for the implementation of 1D filters, including FIR filters [201].

Later, Xilinx generalised the idea of including the accumulation units after the multiplier circuit by offering a flexible ALU circuit that implements multi-input logic functions as well as addition/subtraction/accumulation operations [202]. They also added dedicated and low latency cascade interconnections between two consecutive DSPs for expansion to higher precision and complex arithmetic and efficient implementation of 1D and 2D digital filters.

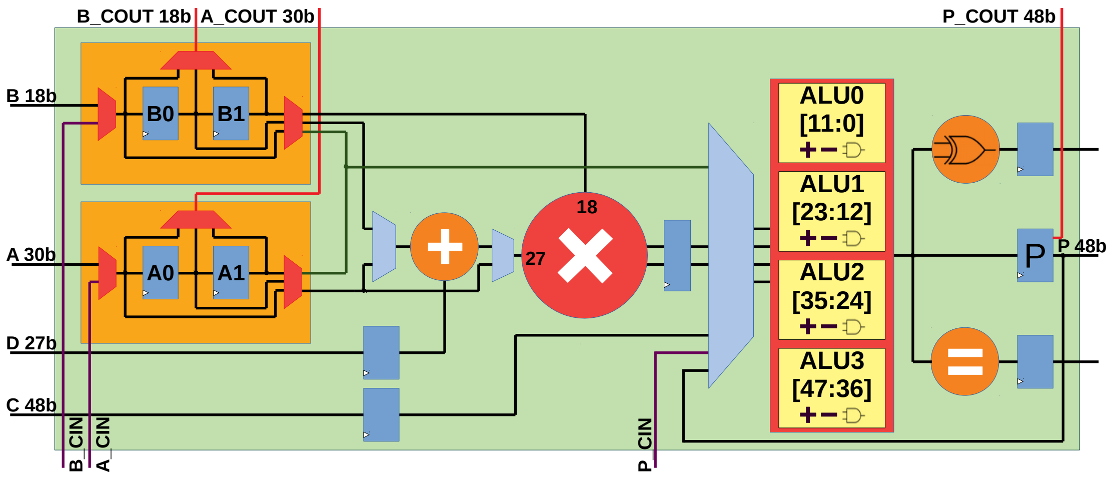


FIGURE 2.8. Xilinx DSP48E2 schematic.

Until recently, the focus of FPGA vendors was to support applications where optimised DSP blocks for high-precision arithmetic with multiple pipeline stages were desired. For instance, Xilinx proposed DSP48E1 blocks with a new 25×18 multiplier that operates in a high-speed multi-stage pipeline mode coupled with a pre-adder, comparator and wide XNOR gates. The multiplier size again increased to 27×18 in the most recent versions, the DSP48E2 [203] illustrated in Figure 2.8. It also included a 27-bit pre-adder, 48-bit accumulator, 48-bit ALU, and a 48-bit pattern detection unit. In SIMD mode, dual 24-bit or quad 12-bit addition and subtraction operations are supported. Intel sacrificed previous flexibilities to optimise the Intel Stratix V DSP block [204] with only two multiplier configurations: one 27×27 or two separate 18×18 multiplications (or MACs). Note 27×27 multiplication is also required for FP32 operations that later in Intel Stratix 10 DSP architectures become natively supported [41].

Rapid progress in machine learning algorithms, with models changing frequently, established an interest in leveraging the FPGA technology for both prototyping and efficient inference deployment. Besides reconfigurability, customised interfaces, and high-energy efficiency, the presence of high-performance DSP blocks was an opportunity. However, early models were relatively simple in terms of the computation datapath and memory structure that mapped straightforwardly to high-performance GPU systems that comprise a large number of hardened high-precision arithmetic units and efficient memory structure. Over time, the optimisation techniques previously discussed in Section 2.1, changed the circumstances for FPGA, but the DSP architectures were still inefficient due to the previous focus on high-precision arithmetic.

Some works suggested innovative ways to reuse available architectures for a pack of low-precision operations. For instance, Xilinx has proposed a method to use 8 DSP blocks to perform 7×2 8-bit multiply-add operations, achieving a $1.75\times$ performance improvement over a naive implementation [44]. Similarly, Colangelo et al. [205] proposed to use an 18×18 multiplier as four different 2×2 multipliers in the Intel architecture. A more generalised study on packing arbitrary precision MAC operations on high-precision DSPs was recently presented by Sommer et al. [206]. This work also advises overpacking that leverages approximate computing to map more multipliers with reasonable overlaps, which enables six 4-bit multiplications using a single Xilinx DSP48E2 multiplier rather than just four.

In order to address this shortcoming through architecture, there are three main tiers of solutions that include dense compute support for low-precision arithmetic while meeting the IOs and area budget limits, described in the following subsections:

1) Enhancing Existing DSP Blocks:

Some studies have been conducted to support larger numbers of low-precision operations using existing DSP blocks. It is interesting that predecessor DSP architectures, such as Intel Stratix-IV architecture that supports up to eight 9×9 multiplications [207], were more flexible and better matched to the new computation demands. Fracturable multi-precision FPGA hard blocks have been first proposed by Parandeh-Afshar and Ienne [208]. This DSP variant, based on a radix-4 Booth architecture, supports 9/12/18/24/36 multiplier word lengths and multi-input addition. Boutros et al. [209] proposed a modification of the Arria-10 DSP that can support $4\times$ 9-bit or $8\times$ 4-bit MACs. For the AlexNet, VGG-16, and ResNet-50 DNNs, this architecture improved speed by up to $1.6\times$ while reducing the utilised area by up to 30%. Later in Chapter 3, a new DSP architecture called PIR-DSP is proposed that demonstrates novel flexibilities in precision, interconnection, and data reuse to enhance the current DSP architectures.

Low precision is also supported on the latest commercial FPGAs as Intel Agilex, which offers native support for INT9 and FP16 using DSP blocks [210]. A recent Xilinx DSP architecture, the DSP58, offers a broader range of precisions compared to its predecessor (the DSP48E2), while providing complete backward compatibility [211]. A single instance of DSP58 is capable of MAC operations up to 27×24 bits, wider pre-adder and logic circuits, as well as support for a run-time configurable 3-element dot-product of 9×8-bit signed values. The main issue with this approach is it increases the critical path for extremely low-precision applications since flexibility requires more multiplexers in the implementation.

2) Integrating Domain-specific Engines:

Adding hardened domain-specific processing units is a new commercial trend. The Xilinx Versal architecture uses a coarse-grained gate array (CGRA) of AI-engines, on the same die as the FPGA [212], interconnected via a network-on-chip. An AI-engine is a simple RISC processor enriched by fixed and floating-point SIMD units accessing dedicated register files, data and program memory, and streaming interconnections. This introduces an additional heterogeneous compute platform and network on the chip to manage, and the RISC processor is large in area compared with a DSP block.

Intel Agilex FPGAs utilise chiplet technology for connecting custom circuits from separate dies as a system in package [210]. This enables integration of an embedded ASIC (eASIC) Intel tensor tile architecture with promising results for deep learning benchmarks [213]. This technology also opens the door for emerging embedded FPGA (eFPGA) designs, such as EFLX tiles from Flex-Logix [214], which natively supports low-precision convolution and matrix operations.

A fundamental issue with domain-specific approaches reviewed is they advocate heterogeneous FPGA and CGRA resources as a solution for ML. This does not directly address the shortcomings of current FPGA architectures, instead it extends the fabric with CGRA resources off to the side. The other issue is resource duplication, where the separation of CGRAs from the FPGA resources necessitates duplicating memory, buffer and routing resources on the CGRA side.

3) Designing New Embedded Block:

This group of solutions advise rethinking new EB architectures rather than upgrading DSP blocks, so no backward compatibility is guaranteed. Intel Stratix 10 NX series replace DSPs with the same size AI-Tensor blocks [215] that are optimised for INT8 and INT4 operations, respectively, up to 30 and 60 MACs in the form of one-input-shared 10-element dot-products. To maintain a similar IO budget, this structure employs double buffering and input sharing among the multipliers [216]. Likewise, Achronix MLP72 block [217] is capable of computing dot-products of 32, 16, 4 and 2 pairs of inputs, respectively, in INT3/4, 8INT, INT16, and BF16 precisions. This block also natively supports block floating point arithmetic through dedicated circuitry for exponent addition and subtraction. As the name of these two extremely specialised architectures also highlights, they are no longer considered typical DSP blocks, rather they form a new class of AI/ML-optimised EBs.

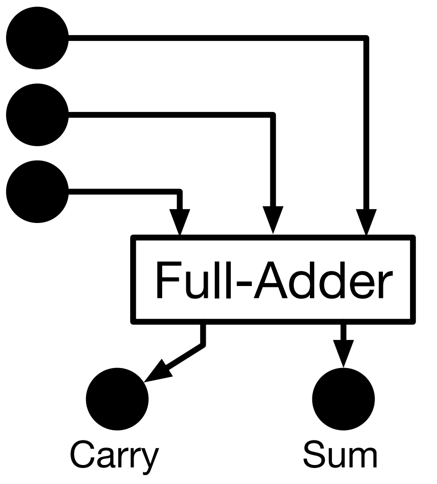
Recently, Arora et al. [218] proposed an output stationary 2D systolic array block to augment an FPGA architecture. The blocks can be composed through a 2D mesh arrangement to create larger systolic arrays. This architecture focuses on only one computation, namely matrix multiplication. Unfortunately, many applications do not map well to matrix multiplication units, e.g. 1D data processing and Microsoft Brainwave-like accelerators that use matrix-vector unit (MVU) as the primitive. As a follow-up, Arora et al. [219] proposed Tensor Slice. This EB is a flexible array of processing elements supporting multiple hand-picked tensor operations with various precisions.

2.3 Compressor Trees

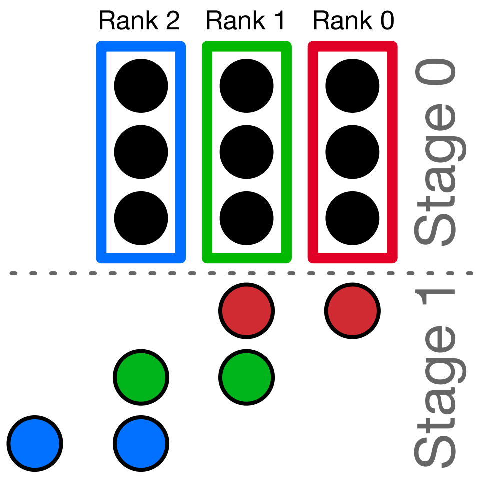
This Section provides background on parallel counters, GPCs, compressors, and compressor trees.

2.3.1 Parallel Counters

Parallel counters are digital circuits that simply count the number of asserted bits in the input, returning this value as a binary output. They can be specified in $(p{:}q)$ notation, where p is the number of input bits, and q is the number of output bits used to express the result in binary notation. Half-adders (HAs) and full-adders (FAs) are commonly used parallel counters, denoted as (2:2) and (3:2), respectively. Parallel counters can also be expressed in dot notation [220] as shown for the full-adder in Figure 2.9A. We use this notation frequently in this paper to visualize various designs, and use the terms bits and dots interchangeably. Figure 2.9B shows how FAs can be used in parallel to implement a single stage of carry-save addition for a 3-bit (3b) 3-operand addition. Note that each FA takes inputs from a single column, and hence, all input bits to a parallel counter have the same $rank$, *i.e.* they all have the same weight.



(A) FA takes in three bits (dots) and produces two outputs: sum and carry



(B) One stage of 3b carry-save addition of three operands using three FAs in parallel

FIGURE 2.9. (3:2) parallel counter, also known as a full-adder.

2.3.2 Generalised Parallel Counters

Generalised Parallel Counters, or GPCs, were first proposed by Meo [221] and subsequently shown by Parandeh-Afshar et al. [195] to map efficiently to FPGAs. Unlike parallel counters, GPCs allow input bits to have different weights, which, in the dot notation, make the GPCs

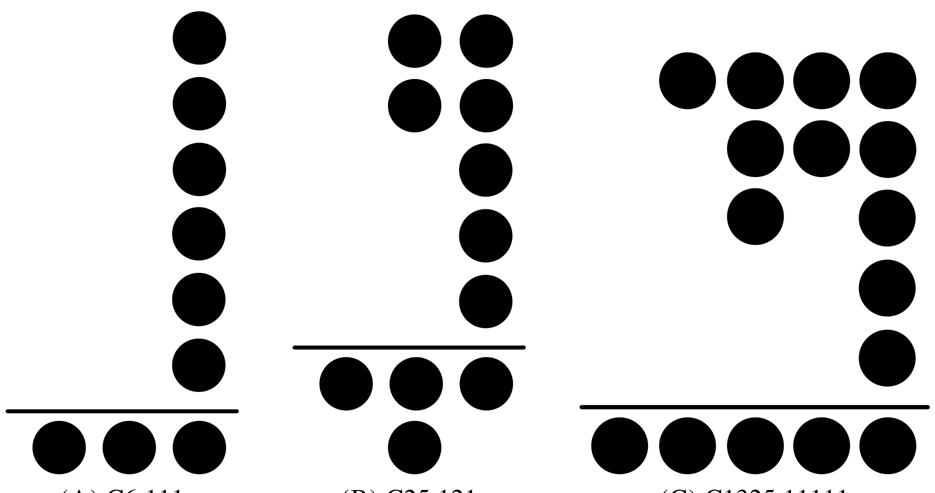

(A) C6:111 (B) C25:121 (C) C1325:11111

FIGURE 2.10. Three popular GPCs found in the literature

appear as multi-column counters. Figure 2.10 shows the dot notation of some previously published GPCs [222]. Mathematically, GPCs are written as a tuple:

$$(p_{n-1}, ..., p_1, p_0{:}q_{m-1}, ..., q_1, q_0) \tag{2.7}$$

, where p_i is the number of input bits in the i^{th} column, and q_j is the number of output bits in the j^{th} column. FPGA implementations can be classified as lookup table-based GPCs [223], or carry-chain-based GPCs [224]. As their names suggest, the shape of a GPC can have a profound impact on its hardware implementation on FPGAs, and subsequently its performance and efficiency in a compressor tree. Popular metrics to quantify the efficiency of a GPC include [225, 222]:

$$\text{GPC efficiency, E} = \frac{p-q}{k} \tag{2.8}$$

$$\text{Strength, S} = \frac{p}{q} \tag{2.9}$$

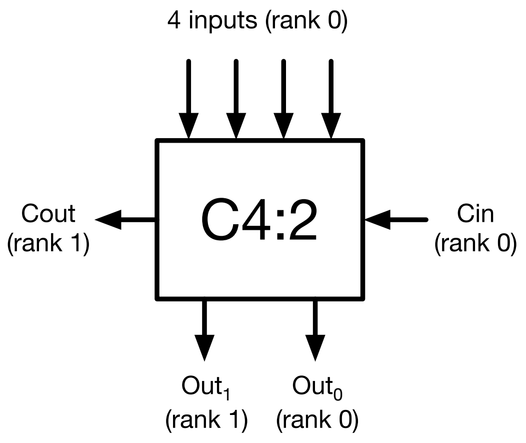
$$\text{Area-Performance Degree (APD)} = \frac{(p-q)^2}{k * d} \tag{2.10}$$

$$\text{Arithmetic slack, A} = 1 - \frac{1 + \sum_{i=0}^{m-1} 2^i p_i}{1 + \sum_{i=0}^{n-1} 2^i q_i} \tag{2.11}$$

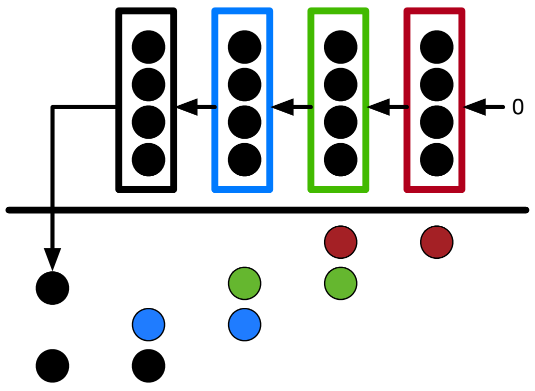
where p and q are the number of input and output bits to/from the GPC respectively, k is the area utilisation (in LEs) of the GPC, and d is the critical path delay (in nanoseconds) of the

GPC implementation. We tabulate the efficiency of each GPC studied in this work using these metrics later in this paper (Table 4.1 and Table 4.3).

2.3.3 Compressors



(A) Block diagram



(B) One stage of a four-operand 4b-addition using four (4:2) compressors

FIGURE 2.11. Simple (4:2) compressor example.

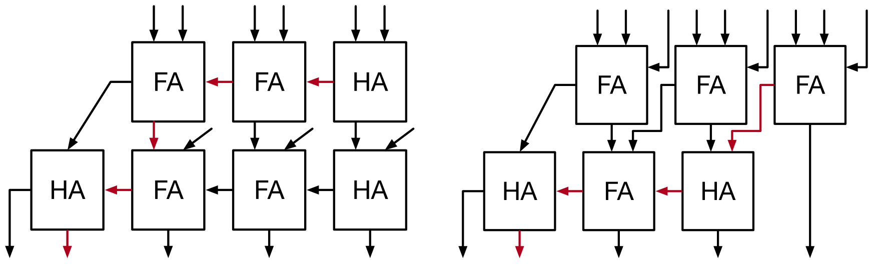
Compressors can be considered parallel counters, with one main difference: they have explicit carry-in (Cin) and carry-out (Cout) bits that can be connected to adjacent compressors in the same stage, as shown in Figure 2.11B. In contrast to carry-propagate adders, the carry-chains between compressors are not cascaded and hence reduce the critical path. Instead, they are connected in a carry-save manner. So the overall delay of the circuit scales much better (Figure 2.12). To the best of our knowledge, the (4:2) compressor (see Figure 2.11A) is the only FPGA-friendly [196]) design that targets Xilinx FPGAs, while no efficient compressors exist for Intel devices. Parandeh-Afshar et al. [50] addressed this issue by proposing configurable carry-chains as modifications to the Intel ALM, supporting 6:2 and/or 7:2 compressors.

For brevity, we describe adders/compressors/GPCs uniformly with a tuple like Equation 2.7 which is simplified by omitting commas. For example, we describe the GPC (6:1,1,1) as C6:111, the (4:2) compressor as C5:21, or the full adder as C3:11.

2.3.4 Adder and Compressor Trees

For multi-operand addition, we can build adder trees by chaining multiple ripple-carry adders (RCAs). Figure 2.12A shows the addition of 3×3b operands. The carry-out from each FA/HA propagates to the next FA, which results in a long critical path along the carry-chain (shown in red). While the RCA has a small area footprint, this long delay is undesirable and can limit performance, especially for operands with large bitwidth.

The carry-save adder (CSA) [226] addresses this issue by treating the full-adder as a C3:11 compressor and breaking the carry-chain as shown in Figure 2.12B. By avoiding the carry chain, the delay is largely determined by the depth of the tree. However, the final stage must be reduced to the final answer using an RCA. Nevertheless, the CSA adder reduces the overall delay of the addition. For the example in Figure 2.12, the critical path delay has one less full-adder delay.



(A) Ripple-carry adder. (B) Carry-save adder.

FIGURE 2.12. Examples of two types of adders.

This idea of breaking the carry-chain dependency up till the final RCA stage is the basis behind compressor trees. A compressor tree is simply a circuit that takes in a set of binary values (or dots) that represent multiple operands, and outputs the result as a sum and carry. Stage 0 in Figure 2.9B is a compressor tree that produces sum and carry bits as inputs into Stage 1, which are then evaluated by an RCA to produce the final result (see HA→FA→HA row in Figure 2.12B, which is the RCA stage). Compressor trees can be built using GPCs, compressors, or both, and efficient compressor tree design is an active area of research with large bodies of existing literature [196, 225, 227, 224, 228, 50, 222, 197].

The reader is encouraged to read [50] for a more detailed background on parallel counters, GPCs, compressors, and different methods of compressor tree implementations.

CHAPTER 3

PIR-DSP: An FPGA DSP block Architecture for Multi-Precision Deep Neural Networks

This chapter describes a novel DSP block architecture called PIR-DSP, which incorporates precision, interconnect and reuse optimisations to enhance FPGA DSP blocks for adopting embedded DNN applications, where standard, PW, and DW convolutions form the majority of the computations. It was shown that the PIR-DSP block could significantly reduce the energy consumption for low-precision deployments at a reasonable cost of area overhead.

The presentation here expands on work previously published in Reference [1].

3.1 Introduction

Utilising massively parallel architectures, DNNs are much more memory and computationally expensive than previous approaches, and efficient implementations continue to pose a challenge. As described in Chapter 2, modern CNNs shifted from using standard convolution operations to customised kernels [21, 25, 107] aiming for more compute-efficient feature extraction. Even though all these kernels are alike concerning inner MAC operations, their parallelism and data reuse patterns are distinct. Besides the model optimisations, DNN inference accelerators employ low precision arithmetic operations to decrease memory footprint and computation requirements [116, 117, 112, 118].

FPGA technology well suits such levels of customisation by providing configurable resources and interconnections for implementing arbitrary precision arithmetic and specialised data path configurations. Reference [11] compared the implementation of MAC units with different word lengths using LEs on Xilinx and Intel FPGAs. They reported that using fixed point 8×8-bit operations instead of single precision floating point, logic resource utilisations are reduced by $10 - 50\times$. This idea has been taken to its conclusion with ternary and binary operations which achieve extremely high speed and low energy implementations on FPGA platforms [29, 150]. Although LEs are flexible-enough to implement any given circuit, their poor compute density is a barrier to efficient and high-performance deployments.

Current FPGAs also include DSP blocks to allow efficient implementation of MAC operations. Unfortunately, as for CPU, GPU and ASIC architectures, they are optimised for higher precision (8-18 bits) and do not efficiently support low precision MAC operations, leading to inefficiencies in resource usage and energy consumption. Using high precision DSPs for low precision calculations is a waste of area and requires additional LUT resources to implement the remaining operations if the DSPs are all utilised. In addition, researchers have proposed strategies involving the run-time selection of word lengths, which can not efficiently be implemented in current FPGA architectures [229].

In addition, research on computer architectures for DNN accelerators has extensively utilised 2D systolic architectures [28, 162]. However, current FPGA DSP block layouts are based on 1D-DSP columns. This mismatch to 2D systolic architectures leads to inefficiencies and requires that general purpose rather than dedicated routing resources be used. It's worth to note each data path pattern mandates a complementary buffering setting that adds to costs by utilising the LEs as pipeline registers and FIFOs.

Through three modifications to Xilinx DSP48E2 DSP blocks, while guaranteeing complete backward compatibility, we address the raised issue for important computations in embedded DNN accelerators, namely the standard, depth-wise, and point-wise convolutional layers. First, a flexible *precision*, run-time decomposable multiplier architecture for CNN implementations is proposed. Second, a significant upgrade to DSP-DSP *interconnect* is suggested that provides a semi-2D low precision chaining capability to support the low-precision multiplier architecture. Finally, data *reuse* is improved via a register file which can also be configured as FIFO. Compared with the 27×18-bit mode in the Xilinx DSP48E2, the proposed Precision, Interconnect, and Reuse-optimised DSP (PIR-DSP) offers a $6\times$ improvement in multiply-accumulate operations per DSP in the 9×9-bit case, $12\times$ for 4×4 bits, and $24\times$ for 2×2 bits. As estimated, PIR-DSP decreases the energy consumption to 31/19/13% of the original value in a 9/4/2-bit MobileNet-v2 DNN implementation.

3.1.1 Previous Work in Multi-precision DSPs

Some previous research has been conducted in supporting larger numbers of low-precision operations using existing DSP blocks. Section 2.2.2.2 presented examples using both Intel and Xilinx DSP blocks. However, to address this shortcoming with DSP architecture, some other studies tried to enhance current DSPs with minor modifications by adding native support for low-precision MAC operations while keeping the promise of backward compatibility. Multi-precision FPGA hard blocks have been proposed by Parandeh-Afshar and Ienne [208]. This DSP variant, based on a radix-4 Booth architecture, supports 9/12/18/24/36 multiplier wordlengths and multi-input addition. later, Boutros et al. [209] proposed a modification of the Arria-10 DSP that can support $4\times$ 9-bit or $8\times$ 4-bit MACs, respectively in the form of 2 and 4-element dot products. For the AlexNet, VGG-16, and ResNet-50 DNNs, this architecture improved speed by up to $1.6\times$ while reducing utilised area by up to 30%.

The proposed PIR-DSP differs from previous designs in that it is a parameterised DSP block architecture with improved flexibility, considers buffering within the DSPs, and also considers inter-DSP interconnect. This serves to improve the speed and energy consumption of the standard, DW and PW convolutions of Table 2.1, with FC layer computations unaffected by our changes. Recently, Dai et al. [230] incorporates approximate multipliers into PIR-DSP architecture to reduce the overheads and gain higher performance for machine learning applications.

In addition to upgrading DSP blocks, integrating domain-specific engines and designing new EBs are two other approaches to enhance FPGA architectures in general for AI workloads, where both methods commonly suggest moving on by integrating new specialized blocks without addressing the DSP architecture in-efficiencies for the new demand. Although these techniques are out of the scope of this chapter, the reader is highly encouraged to read Chapter 2 for a more in-depth background.

3.1.2 Contributions:

A novel *precision, interconnect* and *reuse* optimised DSP block (PIR-DSP), which is optimised for implementing area and energy-efficient DNNs, is proposed. In particular, this chapter presents the following contributions:

- Precision: A parameterised MAC (MAC-IP) with run-time precision control, utilising a novel combination of chopping and recursive decomposition.
- Interconnect: A DSP interconnection scheme that provides support for semi-2D connections and low-precision streaming.
- Reuse: Inclusion of register files within the DSP to improve data reuse and reduce energy.
- Performance evaluation of the PIR-DSP, which incorporates the MAC-IP, interconnect and reuse optimisations, for implementing machine learning primitives including standard, DW and PW convolution layers in recent embedded DNNs.

PIR-DSP is implemented as an open-source parameterised module generator that can target FPGAs or ASICs. All source code and data, along with a spreadsheet to reproduce all the results in this chapter, are available from `www.github.com/raminrasoulinezhad/PIR-DSP`.

3.2 PIR-DSP: Architectural Modifications to the Xilinx DSP48E2 DSP Block

This section presents PIR-DSP architecture (Figure 3.1) through three modifications to the Xilinx DSP48E2 block (that was previously introduced in Chapter 2 and illustrated in Figure 2.8).

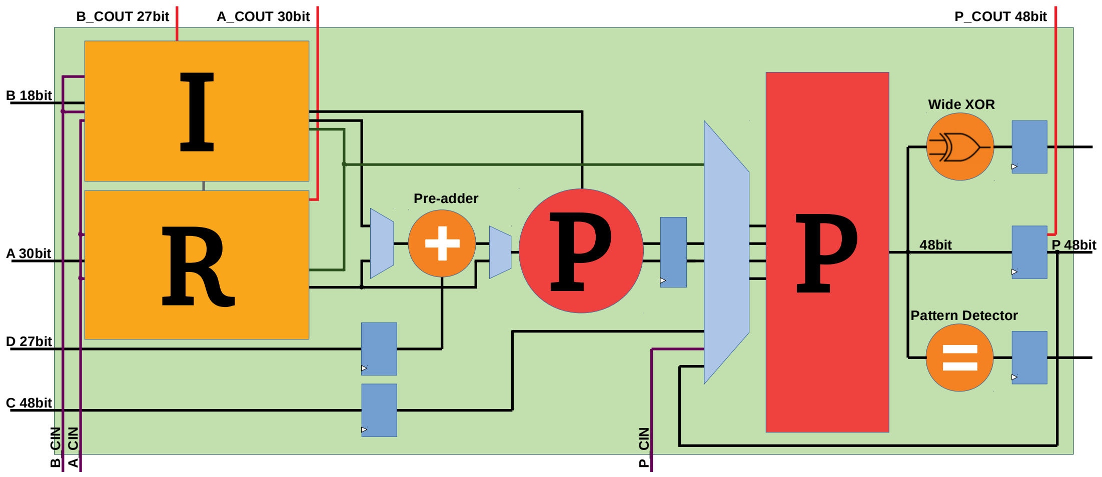


FIGURE 3.1. The overall PIR-DSP schematic (P, I, R blocks encapsulate *precision*, *interconnect*, and *reuse* circuits).

3.2.1 Precision: Decomposable Multiplier

Our multiplier decomposition strategy is based on two approaches: chopping and recursive decomposition.

3.2.1.1 Chopping

A signed 2's complement number can be represented as the sum of one signed (the most significant part) and an unsigned term

$$\begin{aligned} A^s &= [a_{n-1}a_{n-2}...a_{k+1}]_2^s \times 2^k + [a_k a_{k-1}...a_0]_2^{un} \\ &= A_H^s + A_L^{un} \end{aligned} \tag{3.1}$$

where the k^{th} bit is the dividing point and the A_H^s and A_L^{un} are the signed and unsigned portions.

When applied to signed multiplication, this enables the separation of lower-precision product terms

$$A^s B^s = A_H^s B_H^s 2^{2k} + A_H^s B_L^{un} 2^k + A_L^{un} B_H^s 2^k + A_L^{un} B_L^{un} \tag{3.2}$$

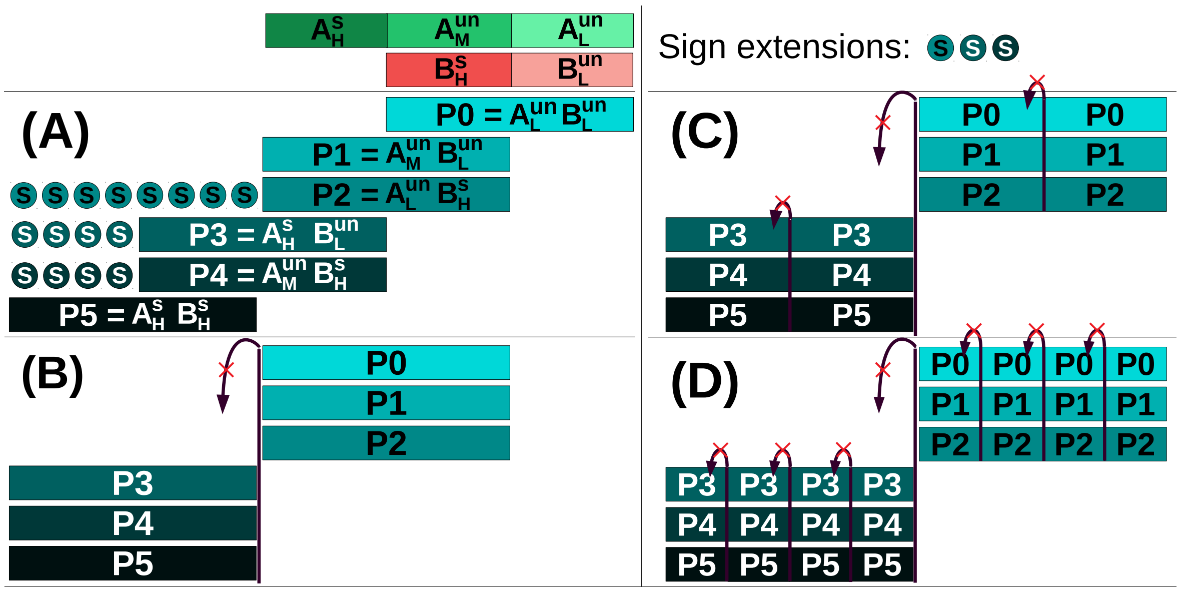


FIGURE 3.2. High-level presentation of chopping (A and B), and divide and conquer techniques (C and D).

with each input being chopped at the k^{th} bit. This can be generalised for any N$\times$M-bit multiplier with chopping size C, if N and M are dividable by C, as follows:

$$A^s B^s = \sum_{i=j=0}^{i=\frac{N}{C}-1, j=\frac{M}{C}-1} A_i^{\{s \text{ or } us\}} B_j^{\{s \text{ or } us\}} 2^{(i+j)C} \tag{3.3}$$

where A_i and B_j are the i-th and j-th C-bit chopped sections of variable A and B, which are respectively signed only when $i = \frac{N}{C} - 1$ and $j = \frac{M}{C} - 1$.

Consider Equation 3.3 applied to a 27$\times$18-bit multiplier with chopping size 9. As shown in Figure 3.2A, standard multiplication is done by summing the six partial results with appropriate shifts. Figure 3.2B shows that by controlling the shift steps for the first, fourth and fifth partial results, the summation can be arranged into two separate columns, where each column calculates a 3-C$\times$C-bit-MAC operation with separated carry-in signals

$$\begin{aligned} \text{Out}_{\text{LSB}} &= P_0 + P_1 + P_2 + C_{in0} \\ \text{Out}_{\text{MSB}} &= P_3 + P_4 + P_5 + C_{in1}. \end{aligned} \tag{3.4}$$

3.2.1.2 Recursive Decomposition

We employ the twin-precision technique [231] in a signed/unsigned $N \times N$ multiplier. Inputs are 1-bit extended according to the individual sign control signals and their most significant bits (MSBs). The extended inputs are then multiplied using a $(N+1) \times (N+1)$ signed multiplier based on the Baugh-Wooley structure [232]. Figure 3.3A shows the baseline multiplier where A and B are 9-bit numbers and each colored circle represents a logical function. By modifying the logic circuits of the PPs and preventing carry propagation using mode control signals, the multiplier can also operate as two half-precision multipliers. The required modifications are depicted in Figure 3.3B. Figure 3.3C shows a recursive application of the technique to compute four quarter-precision values in parallel, only small changes to the PP logic and carry propagation paths being required.

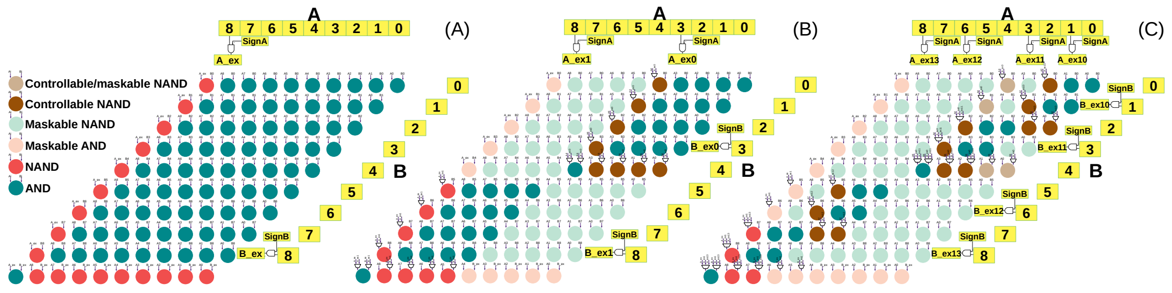


FIGURE 3.3. Recursive decomposition of a signed/unsigned 9×9 multiplier for depth factors (from left to right): (A) 0, (B) 1, and (C) 2. Maskable AND and NAND gates are three-input AND and NAND gates respectively where the output can be masked using the 3rd input. Controllable NAND gates use an XOR gate to pass or flip the AND gate output. Controllable and maskable NAND use both schemes.

Our multiplier is parameterised by chopping factors (separately for each of the two inputs) and the depth. For an M×N multiplier, we use the notation M×NCijDk where i and j are chopping factors (the numbers of times we chop M and N), and k is the recursive decomposition depth factor.

We applied our idea to the Xilinx DSP48E2 27×18 multiplier which produces two partial results (the following ALU is responsible for adding these two outputs). To create a 27×18C32D2 configuration, we chop A and B into $i = 3$ and $j = 2$ 9-bit parts. As each smaller multiplication is a signed/unsigned 9-bit multiplication, we then used recursive decomposition with depth $k = 2$ to change the 9×9 signed/unsigned multiplier to additionally support two 4×4 or four 2×2 multiplications (Figure 3.3C). Extra bits are included so that this is done without precision loss. Figures 3.2C and D show how the bit-level carry propagation from each column to the next is arranged. Combining the six 9×9 multipliers, we can compute the following multi-precision MAC operations without precision loss:

- One signed/unsigned 27×18
- Two sets of signed/unsigned $(9 \times 9 + 9 \times 9 + 9 \times 9)$
- Four sets of signed/unsigned $(4 \times 4 + 4 \times 4 + 4 \times 4)$
- Eight sets of signed/unsigned $(2 \times 2 + 2 \times 2 + 2 \times 2)$

We have developed an IP generator which uses these techniques to convert any size multiplier to a MAC-IP. Each operand can be signed or unsigned, controllable using a selector signal at run-time.

3.2.2 Interconnect: Low-precision, Semi-2D DSP-DSP Communication

Low energy and high performance DNN accelerators have been demonstrated using systolic array architectures [28, 162]. In this section, we focus on data movement among processing elements (PEs), which are DSP blocks in this content. In particular, 3×3 convolutions are of most interest as these dominate the embedded DNNs reviewed in Chapter 2.

Whereas in ASIC designs the PEs can be arranged in a 2D pattern, FPGA DSP blocks are arranged in columns. In each column, DSP inputs and outputs can be passed via dedicated chain connections. This single-direction chaining is highly efficient for their intended signal processing applications. Although general routing resources make it possible to configure a 2D mesh network of PEs, this approach introduces significant amounts of additional circuitry and latency compared with direct connections.

In 2D systolic architectures, PE interconnections must forward input and result data to two different destination PEs, usually in different dimensions. Figure 3.4A shows a 2D PE

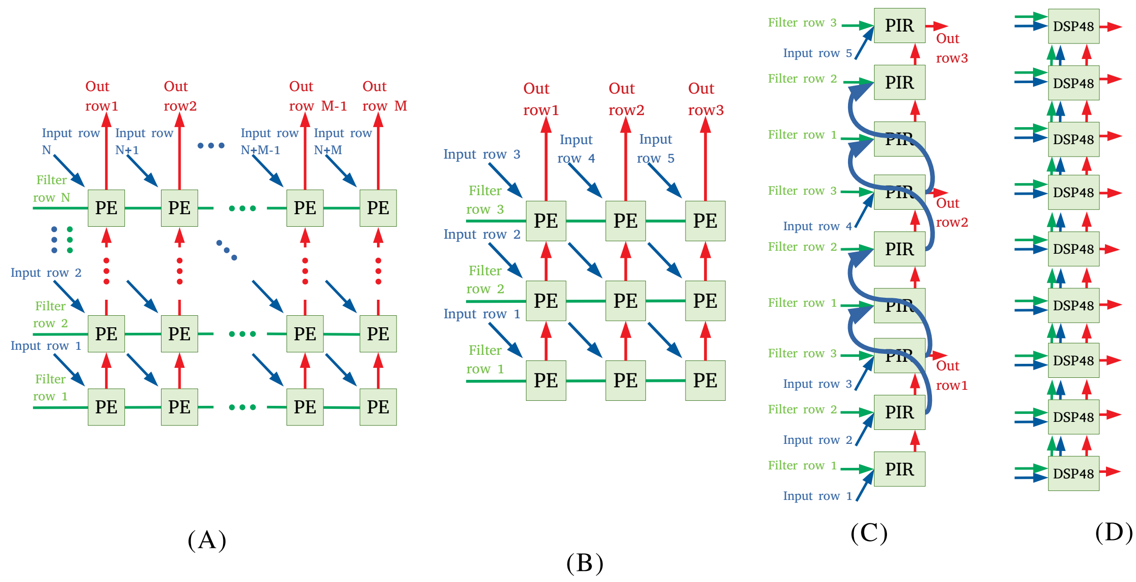


FIGURE 3.4. (A) Conventional 2D processing element architecture in [28] (B) 3×3 convolution layer implementation on 2D architecture (C) Our Semi-2D DSP arrangement (D) conventional FPGA column-based arrangement.

architecture, proposed in [28], which is a N×M mesh network of PEs with unidirectional communications occurring in horizontal, vertical and diagonal directions. In Figure 3.4B a 3×3 convolutional layer is assigned to three rows of the PEs. By rearranging this three-row architecture as shown in Figure 3.4C, we organise them as a column. When implementing 2D systolic arrays solutions on conventional FPGA column-based chains, it is impossible to use both the input and output dedicated chain connections as they have same source and destination. Figure 3.4D shows a column-based connection which is capable of forwarding the data/result to the next DSP block. This addresses the difficulty of implementing a 2D interconnection on a 1D array, by supporting data forwarding to *two* DSPs instead of a single one. This is particularly effective for the case where one dimension is small (*e.g.* 3 elements for 3 × 3 convolutional layers).

Current DSP columns are capable of streaming high-precision data over the chains. To stream low precision inputs, we make some minor modifications to the input B register and chaining connections to support both high and low precision data streaming. Also, we modified both DSP input chains (A and B) to support run-time configurable input data forwarding up to next two DSPs. This is done by bypassing the next DSP to enhance the implementation capabilities for improving data reuse via a small modification to current FPGAs. With our changes, the 18-bit input B can feed both B 27-bit shift registers and their 9-bit LSB portions via both A and B chains. Furthermore, the design supports run-time configuration (Figure 3.5) of stream precision. When used to implement convolutional layers, these modifications support one high-precision or two low-precision streams for the Stride = 1 and 2 cases.

3.2.3 Reuse: Flexible FIFO and Register File

In DNN implementations each input/parameter takes part in many MAC operations, so it is important to cache fetched data. Since data movement contributes more to energy consumption than computation, this leads improved speed and energy [28, 162]. Unfortunately, Xilinx DSP blocks do not support caching of data (this is done using fine-grained resources or hard memory blocks). Intel DSPs do include a small embedded memory for each 18-bit multiplier, but they cannot be configured at run-time and hence can only be used efficiently for fixed coefficients, making them unsuitable for buffering of data for practical sized DNNs.

We propose a small and flexible FIFO/register file (RF) to enhance data reuse. This is a wide shift register can be loaded sequentially and can be read by two standard read ports. The two read port address signals can be provided from outside the DSP block. The first is used inside the DSP and brings the requested and the next data for multiplier and multiplexer units (two 27-bit read ports are needed to feed our multiplier). As RFs are mostly used to buffer a chunk of data inside the DSP, writes always occur as a burst. The other read port is used to select the data for DSP-DSP chaining connections. Using this approach, we arrange the RF as a flexible FIFO. By adjusting the FIFO length, systolic array implementations with different buffering patterns can be implemented. The schematic of our implemented FIFO/RF is given in Figure 3.5, and operates on input A.

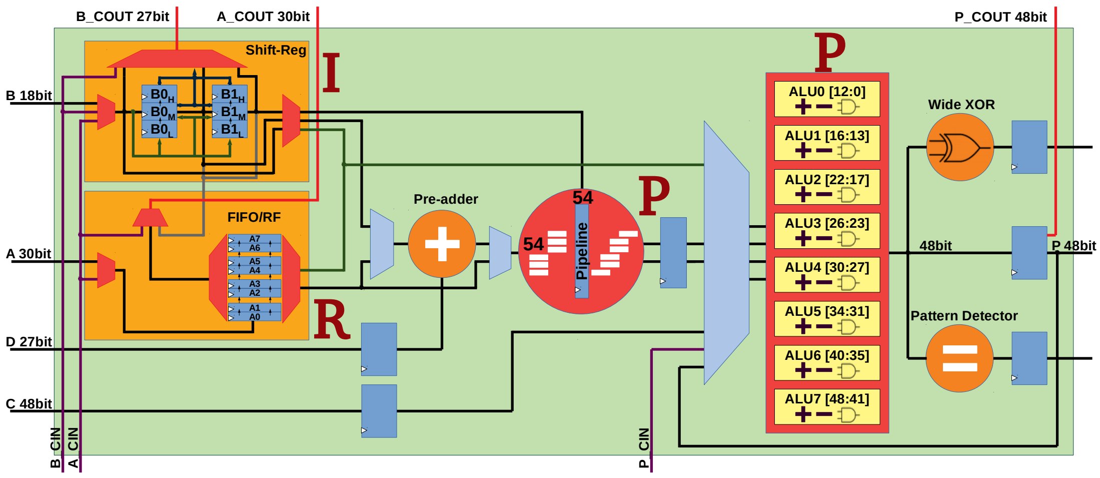


FIGURE 3.5. The detailed PIR-DSP schematic using a 27×18C32D2 MAC-IP (all registers are bypassable).

3.3 Experimental Study

3.3.1 Baseline DSP48

As a baseline, we modelled the Xilinx DSP48E2 DSP block using Verilog and synthesized it using SMIC 65-nm technology standard cell with Synopsis Design Compiler 2013.12. Post-synthesis reports show that DSP48E2 timing is consistent with reported speeds for DSP48E1 in Virtex 5 speed grade -1, especially the critical path, which is 3.85 and 3.94 ns, respectively for our DSP48E2 and Virtex-5 DSP48E1. A comparison with DSP48E1 rather than DSP48E2 was made as the former generally has the same DSP architecture and 65 nm process technology [233]. DSP48E2 is the most recent version, including three major architectural upgrades; wider multiplier unit (27×18 instead of 25×18), pre-adder module, and wide XOR circuit. We were not able to compare area since no information is available for the DSP48E1/2 [234].

The baseline DSP48E2 multiplier produces two temporary results, and these are added using the ALU to produce the final MAC output. As a longer critical path is created by the PIR-DSP partial product summation circuits, we applied parallel computing and carry-lookahead techniques for both multiplier and ALU. It was also necessary to add a new pipeline-register

TABLE 3.1. MAC-IP post synthesis results (area ratio 1 = 9224 um^2).

MAC Model	Area (ratio)	Fmax (MHz)	# of MAC / Energy per MAC (pJ)			
			27×18	9-bit	4-bit	2-bit
27×18-MAC	1	763	1/28.4	1/28.4	1/28.4	1/28.4
27×18C32D0	1.46	730	1/37.6	**6/6.3**	6/6.3	6/6.3
27×18C32D1	1.86	671	1/43.9	6/7.3	**12/3.7**	12/3.7
27×18C32D2	1.70	538	1/47.9	6/8.0	12/4	**24/2.0**
27×27C33D0	2.12	714	1/54.1	9/6.0	9/6.0	9/6.0
27×27C33D1	2.21	581	1/59.5	9/6.6	18/3.3	18/3.3
27×27C33D2	2.36	380	1/90.8	9/10.1	18/5.0	36/2.5

layer to the multiplier unit to reduce the critical path our more complex circuit. Modifications to the ALU also required replacing the DSP48E2 12/24/48-bit SIMD add/sub operations with a 4/8/18/48-bit SIMD which leads to smaller and width-variant ALUs since they must be aligned with the carry propagation blocking points, as shown in Figure 3.5. We note that the multiplier in the DSP48 is not on the critical path so adding a similar pipeline register does not affect its critical path.

3.3.2 Precision (MAC-IP)

Figure 3.6 and Table 3.1 show post-synthesis Area, Maximum frequency, and energy per MAC operation results for different configurations of the MAC-IP using the performance optimisation synthesis strategy and statistical signal toggle rates. Note, some configurations may not natively support all computation precisions. In those cases, the energy consumption values are estimated using the closest precision mode.

Table 3.2 Configuration #0 shows our synthesised DSP48E2 area and maximum frequency. Configurations #1 to #3 results are obtained by simply replacing the multiplier and ALU units in the DSP48E2 with the MAC-IP. These configurations are different in their 9×9 multiplier structures, respectively generated by recursive decomposition factors of 0, 1, and 2, and depicted in Figure 3.3A, B, and C. Upgrading the multiplier to a 27×18C32D2 MAC-IP,

TABLE 3.2. PIR-DSP synthesis results for different MAC-IP configurations. The PIR-DSP case includes all our optimisations.

#	DSP Version	Area		F_{Max}
		Post Synth.	ratio	(MHz)
0	DSP48E2	25419	1.00	463
1	+ M27×18C32D0 MAC-IP	28638	1.13	520
2	+ M27×18C32D1 MAC-IP	28838	1.13	463
3	+ M27×18C32D2 MAC-IP	29097	1.14	358
4	+ M27×18C32D2 MAC-IP + interconnect	29972	1.18	362
5	PIR-DSP=MAC-IP+ + interconnect + reuse	32505	1.28	357

leads to improvements in MAC capabilities of $\times 6$, $\times 12$, $\times 24$ times for 9, 4, 2-bit MAC operations respectively, at the cost of a 14% increase in area.

3.3.3 Interconnect and Reuse (PIR-DSP)

Table 3.2 Configuration #4 is produced by adding the interconnect optimisation to Configuration #3. Configuration #5 is the final implementation of PIR-DSP which adds the reuse optimisation. As in the DSP48 data sheet, the reported F_{max} to the P output omits the wide XOR and pattern detector circuits of Figure 3.5.

To evaluate the effectiveness of our proposed data movement modifications for low-precision computations, we focused on the total run-time energy required by implementing low-precision versions of some well-cited embedded CNNs.

We extracted the read and write energy using Xilinx Power Estimator (XPE) for BRAM and LUT blocks on the Virtex-5 FPGA. $E_{\text{BRAM, Read}}$ and $E_{\text{BRAM, Write}}$ per byte were estimated for an 18-bit wide memory configuration (the most efficient way to use BRAMs). To estimate the energy associated with moving data from an off-DSP RF and shift-register (SR), we configured the LUTs respectively as RAM with Fanout = 4 (for broadcasting), and shift register with Fanout = 1 (streaming) (Table 3.3). Using results for small register files in [235,

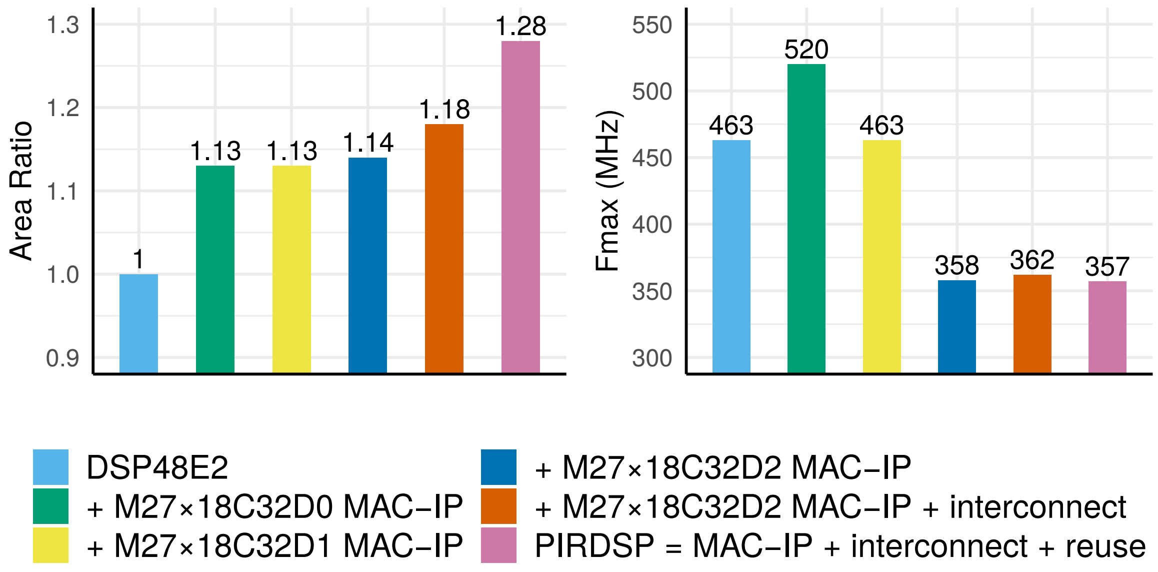


FIGURE 3.6. PIR-DSP synthesis results for different MAC-IP configurations. The PIR-DSP case includes all our optimisations.

236, 126], we estimated our embedded 4×2 30-bit RF read & write energy to be 1.1 pJ/byte. RF width and size are selected respectively, to fully feed the multiplier/pre-adder in high/low-precision and to be similar to Intel DSP block read-only RFs which are configured in two 8×18-bit memories per DSP. To estimate input B energy which operates as a SR and a normal register we used results for high-performance [237] and low energy flip-flops (FFs) [238] to obtain estimates of 180 fJ and 90 fJ respectively. Energy required to transfer data from DSP-DSP was obtained from [239], and scaled to 65nm technology (Using scaling factors in [240]), to obtain 2 pJ per byte. Using the energy ratios from Table 3.1, energy consumption for 9/4/2-bit MAC operations are 89/44/22× that of a 9-bit register. Table 3.4 summarises the estimated energy ratios for data movement. We further assume that all elements (except the MAC) scale linearly with wordlength.

3.3.4 Implementation of Convolutions

We now describe implementations of standard and DW convolutional layers, using a 3×3 DW convolution layer as a case study. According to Equation 2.4, output channels can be computed in parallel. We assumed input and weight parameters are located in BRAMs and

TABLE 3.3. Estimation of BRAM, Off-DSP RF and RS Read/Write access energy 9-bit word on a Xilinx XC5VLX155T extracted from XPE tool (pJ)

BRAM Metrics	Method	BRAM Output width			
		18	9	4	1
E_{Read}	100% Read usage	8.45	15.8	32.3	116
E_{Write}	100% Write usage	9.98	17.9	35.6	128
E_{BRAM} (E_B)	E_{Read}+E_{Write}	**18.43**	33.7	67.9	244
LUT Metrics	Method	LUT FanOut			
		4	3	2	1
E_{RF} (Off-DSP)	LUT as RAM	**3.60**	3.28	2.96	2.64
E_{SR} (Off-DSP)	LUT as SR	4.92	4.59	4.27	**3.95**

TABLE 3.4. Data movement energy ratios in 65 nm Technology (1× = 90fJ).

Energy	FF	SR_e	RF_e	Chain	RF	SR	BRAM(B)	MAC
Ratio	1	2	12.5	23	40	44	205	89-22

results will be written back to BRAMs. In an implementation on conventional DSPs [241], weight stationary data flow was used, with each input feature map element fetched once from BRAMs and then streamed over off-DSP SRs. Meanwhile, weight parameters should be accessed by multiple DSPs, where each DSP reuses a parameter for processing a row of input feature map. Thus, a weight parameter is fetched from BRAMs F_h times to be stored in DSP registers and locally accessed for $F_h \times F_w$ times from the DSP registers in total. Mathematically, each filter and input element are respectively used $F_h \times F_w$ and $K_h \times K_w$ times. The average energy for the described data flow where E_{MAC} is the energy consumption of the MAC computation is

$$\begin{aligned} E_{conv.} &= E_{Input} + E_{Weight} + E_{MAC} \\ &= (\frac{E_B}{K_w K_h} + E_{SR} + E_{FF}) + (\frac{E_B}{F_w} + E_{FF}) + E_{MAC} \end{aligned} \tag{3.5}$$

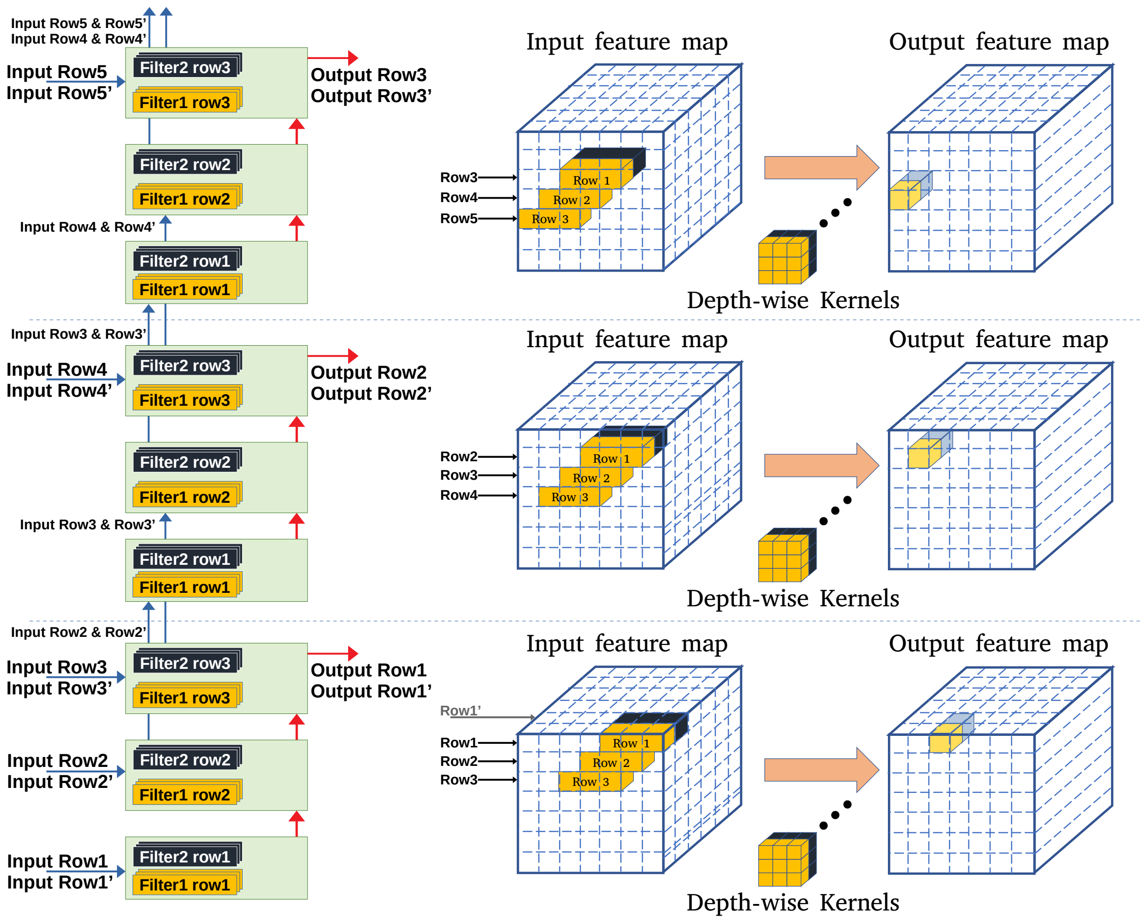


FIGURE 3.7. Proposed implementation for standard and DW convolution layers (the computation assigned to each three PIR-DSPs is shown at right).

3.3.4.1 Depth-wise Convolution

For a PIR-DSP implementation, inspired by the Eyeriss architecture [28], we mapped computation of multiple rows of output channels to a three-cascaded PIR-DSP. Each PIR-DSP can compute 2/4/8 sets of three-MAC operations for 9/4/2-bit precision. Each three-MAC operation can be used for a row of a 3×3 DW kernel. Cascading three PIR-DSPs, we can sum the partial outputs to produce the final output feature map elements. As illustrated in Figure 3.7 for 9-bit precision, each PIR-DSP receives two streams of 9-bit data (as each PIR-DSP can compute two parallel three-MAC operations). The three-cascaded PIR-DSPs can forward two of their streams to the next three-cascaded PIR-DSP over the DSP-DSP chains, and we can implement K rows of 2/4/8 channels of the output matrix for 9/4/2-bit precision using a column of 3K PIR-DSPs. For this case, E_{Input} becomes

$$E_{Stream,Input} = \frac{E_B + (NoF)E_{Chain}}{K_h K_w} + E_{SR_e} \tag{3.6}$$

where NoF is the number of forwarding over chains for each input stream (2 in our case as each row of the input stream is involved in three rows of output feature map). To implement other kernel sizes, we use a kernel tiling approach with tile sizes of $3{\times}3$, $2{\times}3$, and $1{\times}3$ which are respectively the computation capabilities of a three-cascaded, two-cascaded, and a PIR-DSP. Thus a $5{\times}5$ kernel can be implemented using $2\times$ three-cascaded DSPs and $2\times$ two-cascaded DSP groups where NoF is 6.

3.3.4.2 Standard Convolution

For the case of standard convolution, our RF reuse reduces E_{Input} by a factor of RF_{size} (last line of Table 3.5) when an input value is reused for multiplication with various weight parameters. The calculated access energy ratio in the last column indicates that PIR-DSP uses 31% of the data access energy for a middle bottleneck layer of MobileNetv2 [108] which applies 192 depth-wise $3{\times}3$ filters on an input feature map of shape $56^2 \times 192$.

3.3.4.3 Point-wise Convolution

For a PW convolution, each input channel can be streamed into a DSP to be multiplied by corresponding weight parameter, producing a partial result which is cascaded and summed to produce an entry of the output feature map. In a PIR-DSP implementation, we assign three channels of input and three corresponding channels of 2/4/8 PW kernels to a PIR-DSP, depending on precision. PIR-DSP using 2, 4, or 8 three-MAC operations computes partial results of each filter on the same input stream in parallel (the stream includes an element of three channels of input feature map in each cycle). By cascading we can compute 2, 4, or 8 six-MAC operations (computing six elements of the PW kernels). Also, as illustrated in Figure 3.8 for 9-bit precision, each two-cascaded PIR-DSP can forward their streams to the next two-cascaded DSP, which leads to energy reduction as summarised in Table 3.5. Thus, PIR-DSP uses saved weights and performs a MAC with the 2/4/8 3-channel weight

TABLE 3.5. Read Access Energy for Standard/DW/PW Conv Layer per MAC (baseline implementation uses off-DSP resources to stream input over saved weights in DSP registers).

	Method	E_{Input}	E_{Weight}	Ratio (%)
Standard/DW	Baseline	$\frac{E_B}{K_h K_w} + E_{SR} + E_{FF}$	$\frac{E_B}{F_h F_w} + E_{FF}$	100
	Stream	$\frac{E_B + (NoF)E_{Chain}}{K_h K_w} + E_{SR_e}$	$\frac{E_B}{F_h F_w} + E_{FF}$	45
	Stream&RF	$\frac{E_B + (NoF)E_{Chain}}{K_h K_w RF_{s=4\times2}} + E_{SR_e}$	$\frac{E_B}{F_h F_w} + P_{RF_e}$	31
PW	Baseline	$\frac{E_B}{N} + E_{SR} + E_{FF}$	$\frac{E_B}{F_h F_w} + E_{FF}$	100
	Stream	$\frac{E_B}{N} + E_{Chain} + E_{SR_e}$	$\frac{E_B}{F_h F_w} + E_{FF}$	58
	Stream&RF	$\frac{\frac{E_B}{N} + E_{Chain}}{RF_{s=4\times2}} + E_{SR_e}$	$\frac{E_B}{F_h F_w} + E_{RF_e}$	44

parameters which are saved in two 27-bit registers. Furthermore, the RF improves input data reuse. By applying the equations to a middle bottleneck layer of MobileNet-v2 (which includes 192 PW $1 \times 1 \times 32$ filters on $56^2 \times 32$ input feature map), our proposed optimisations can reduce the read access energy to 44% of the original value.

A similar analysis was applied to all layers of some common embedded DNN models, the results in Table 3.6 are obtained. For example, when applying all our optimisations to MobileNet-v2 [108], energy is reduced to 31/19/13% of the original value for 9/4/2-bit precision.

3.3.5 Comparison with Previous Work

BitFusion [180] is an ASIC DNN accelerator, supporting multi-precision MACs. The reported area is for a computation unit in 45-nm technology, comprising 16 BitBricks, each of which is a 2-bit plus sign multiplier. This is similar to our 27×18C32D2 MAC-IP (Table 3.1),

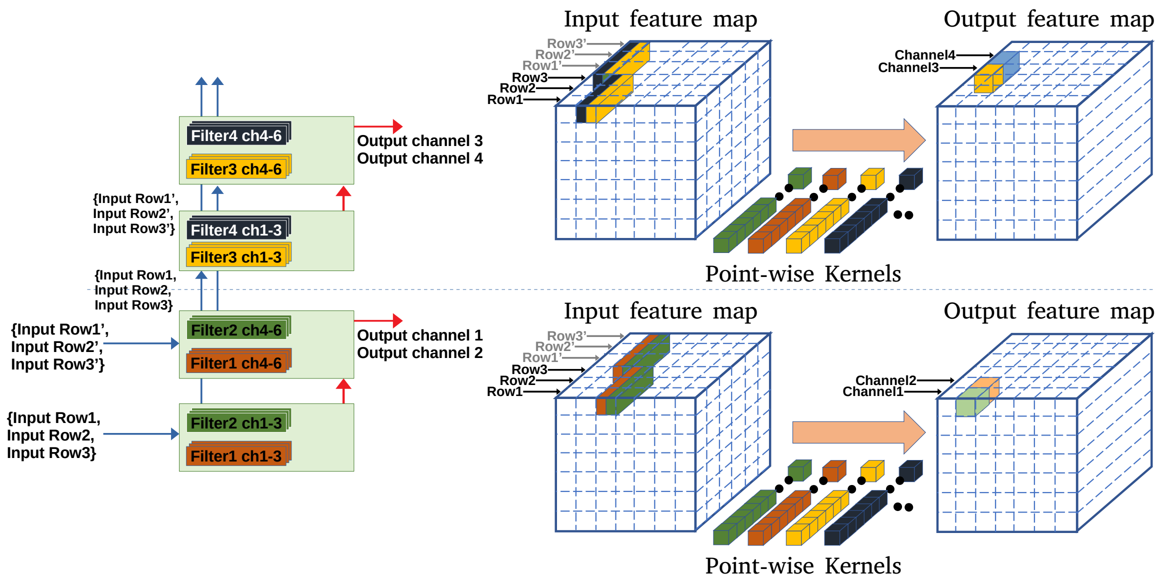


FIGURE 3.8. Proposed implementation for PW convolution layers (the computation assigned to each two PIR-DSPs is shown at right).

TABLE 3.6. Energy ratio of PIR-DSP optimisations for 9/4/2-bit precision (percent)(baseline = 100/100/100).

Modification			NASNetA4	Mobile	Shuffle	Squeeze
P	**I**	**R**	**@1056 [113]**	**Net-v2 [108]**	**Net-v2 [114]**	**Net [22]**
✗	✗	✗	baseline	baseline	baseline	baseline
✓	✗	✗	39/26/20	38/26/20	38/26/20	40/28/22
✓	✓	✗	33/20/14	33/21/15	33/21/15	31/20/14
✓	✓	✓	31/19/13	31/19/13	31/19/13	29/17/12

although BitFusion is more flexible as it supports more variations including 2×4, 2×8 and 4×8. Table 3.7 compares performance per area (PPA). We used the maximum frequency reported for a same implementation, DSP48E1, in three FPGAs, Virtex5/6/7, normalized to feature size [240] (area is scaled by 1/0.66/0.3 and maximum frequency by 1/1.1/1.35 respectively for 65/45/28 nm). BitFusion only applies the chopping technique, leading to high area overhead. The introduction of recursive decomposition better supports low and high-precision MAC operations.

TABLE 3.7. Comparison with previous work. Main entries are in (# of MAC per cycle / MACs per second per DSP (GOps/Sec)) format.

Module	**BitFusion [180]**	**MAC IP**	**PPA (ratio)**	**Boutros [209]**	**PIR DSP**	**PPA (ratio)**
freq./Tech (MHz/nm)	500 / 45	537 / 65		600 / 28	357 / 65	
Area um^2	2148	15759		9389	32505	
Area × Delay ratio	0.17	1		0.17	1	
Area × Delay ratio (norm)	0.24	1		0.77	1	
2×2/Bin./Ter.	(16/8)	(24/12.9)	0.4	-	**(24/8.5)**	-
4×4	(4/2)	(12/6.4)	0.7	(8/4.8)	(12/4.2)	**1.2**
8×8	(1/0.5)	(6/3.2)	**1.4**	(4/2.4)	(6/2.1)	**1.2**
16×16	-	**(1/0.5)**	-	(2/1.2)	(1/0.36)	0.4
18×18	-	**(1/0.5)**	-	(2/1.2)	(1/0.36)	0.4
27×18	-	**(1/0.5)**	-	(1/0.6)	(1/0.36)	0.8
27×27	-	-	-	1/0.6	-	-

Boutros et al. proposed improvements to the Intel DSP block [209], and is capable of 27×27 and reduced precision MACs down to 4-bit. In comparison, PIR-DSP can support precisions down to 2 bits, has better performance at 8×8 bits and lower, is generated using a flexible module generator, but is worse at 16×16 and higher PPA. It is not possible to compare energy but we would expect Boutros to be similar to the Baseline case in Table 3.5 with PIR-DSP having significant advantages due to the interconnect and reuse optimisations.

3.4 Summary

This chapter demonstrated the disadvantage of contemporary FPGA DSP blocks for applications, such as embedded DNNs. As a solution, a novel DSP block architecture, called PIR-DSP, was described, which incorporates precision, interconnect and reuse optimisations. When applied to the implementation of embedded DNNs, for which the bottleneck is the standard, PW and DW convolutions, it was shown that PIR-DSP block architecture significantly reduces the energy consumption of low-precision implementations, albeit requiring an extra cycle of latency and a 28% area overhead.

CHAPTER 4

LUXOR: An FPGA Logic Cell Architecture for Efficient Compressor Tree Implementations

This chapter proposes two tiers of modifications to FPGA logic cell architecture to deliver a variety of performance and utilisation benefits with only minor area overheads. In the first tier, the existing commercial logic cell datapaths are augmented with a 6-input XOR gate in order to improve the expressiveness of each element, while maintaining backward compatibility. This new architecture is vendor-agnostic, and it is referred to as LUXOR. Then, a secondary tier of vendor-specific modifications to both Xilinx and Intel FPGAs, which are respectively called X-LUXOR+ and I-LUXOR+, are presented. As demonstrated, compressor tree synthesis using GPCs is further improved with the proposed modifications.

The presentation of this chapter is based on the background given on FPGA architectures and compressor tree implementations in Chapter 2, and expands on work previously published in [2].

4.1 Introduction

The design of parallel computer arithmetic circuits is a well established field of research dating back to the works of Wallace [46], Dadda [47], Swartzlander [48], Verma [49], and others. In the context of FPGAs, there has always been interest in specialised arithmetic primitives which improve performance over a wide range of application domains. One such primitive, GPCs, enables fast accumulation of compressor trees. Work from Parandeh-Afshar et al. [227] motivated the use of GPCs on FPGAs, while Kumm et al. [197] demonstrated software techniques that automate the design of optimal compressor tree implementations for FPGAs. However, modern FPGA LUT based architectures are not particularly efficient for the implementation of compressor trees [50].

This coincides with the substantial application shift from high-precision digital signal processing algorithms to low-precision compute-intensive algorithms, including embedded DNNs. In the former cases, the computations normally map to FPGA DSP blocks. However, as presented before in Section 2 for DNNs, efficient deployments engage quantisation and customised parallel computing pattern techniques, which current DSP architectures do not efficiently support. Therefore, utilising LEs for low-precision fused arithmetic is a new demand trend that appears in a variety of applications. Parallel reduction circuits [29, 17] and pipelined dot-product architectures [30, 31] are great and frequently used compressor tree examples that also dominate the computations in the state-of-the-art embedded DNNs, which appear with various array dimensions and arithmetic precisions. This necessitates rethinking the LE architectures for more efficient support for generalised parallel counters and compressor trees.

4.1.1 Related Work

Parallel digital arithmetic circuits have been explored since the 1960s [46, 47, 48], but FPGA-based compressor trees were only popularised in the past two decades, primarily from work by Parandeh-Afshar et al. [195, 50] and Kumm et al. [196, 197]. In [50], the authors proposed architectural changes to the Intel ALM carry-chains such that large compressors like (6:2) and (7:2) can be efficiently mapped to single ALMs. Although their proposed compressor is very efficient, for modern applications such as BNN popcounting [198], these compressors would be significantly underutilised. Similarly, Kim et al. [194] and Boutros et al. [193] propose changes to the FPGA architecture, by adding sum-chain and extra carry chains, respectively, specifically for modern deep neural network applications, which do not necessarily benefit general-purpose compressor trees. The proposed changes are motivated by insight into modern GPC-based compressor tree designs, and benefit a larger suite of old and new benchmarks.

4.1.2 Contributions

This chapter shows that support for compressor trees in FPGAs could be significantly improved through minor modifications to the LE. This is beneficial for implementing low-precision and multi-operand operations. One example of interest is that compressor trees and GPCs can be used to accelerate the XnorPopcount operations within BNNs [242], which forms the critical path of the model's execution. BNNs enable neural networks to be utilised in resource constrained applications and can be deployed efficiently on FPGAs [152, 198]; the proposed optimisations would improve their performance further.

LUXOR is a portmanteau of the acronyms LUT and XOR. Its design is motivated by the observation that the Boolean XOR operation is very commonly found in optimised compressor trees. This is corroborated by Verma et al. in [228], where they exploited the correlations between the operands of the XOR function to improve delay for ASIC implementations. The goal is to utilise this insight in a similar vein, but optimised for FPGAs.

The proposed changes provide a means to efficiently implement compressor trees using new area-optimised GPCs, which can all be applied to a large variety and/or important classes of applications. The contributions can be summarised as follows:

- A new logic element, LUXOR, that integrates a 6-input XOR gate with commercial FPGA logic elements. This architecture independent modification improves the implementation of XnorPopcount operation and the most commonly used GPC.
- LUXOR+, an amalgamation of LUXOR with further Intel (I-LUXOR+) and Xilinx (X-LUXOR+) architecture-specific optimisations to achieve further resource reduction. This leads to the most efficient reported logic element based GPC, called C06060606, which can be mapped to just a single Xilinx slice.
- A novel integer linear programming (ILP) formulation based on the flexible Ternary Adder approach proposed in [222] to optimally map compressor tree problems to LUXOR cells.
- Quantitative investigation of the benefits of LUXOR and LUXOR+ architectures using a set of more than 50 micro-benchmarks. The results also show the positive

benefits of the proposed LUXOR and LUXOR+ enhancements in SMIC 65nm standard cell technology.

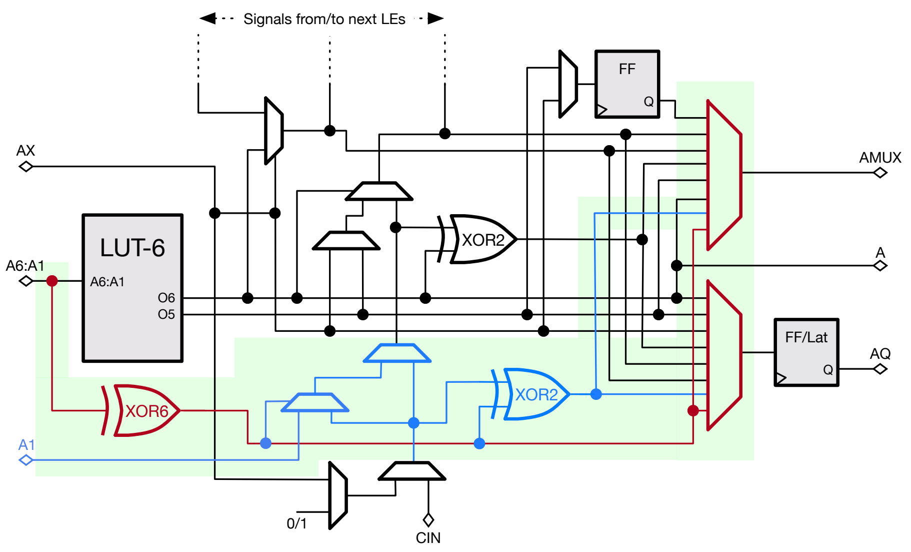
- The ILP-based compressor tree synthesiser, benchmarks and design files required to generate the results in this paper are open source to support reproducible research, and available at `www.github.com/raminrasoulinezhad/LUXOR_FPGA20`.

4.2 FPGA Logic Cell Enhancements

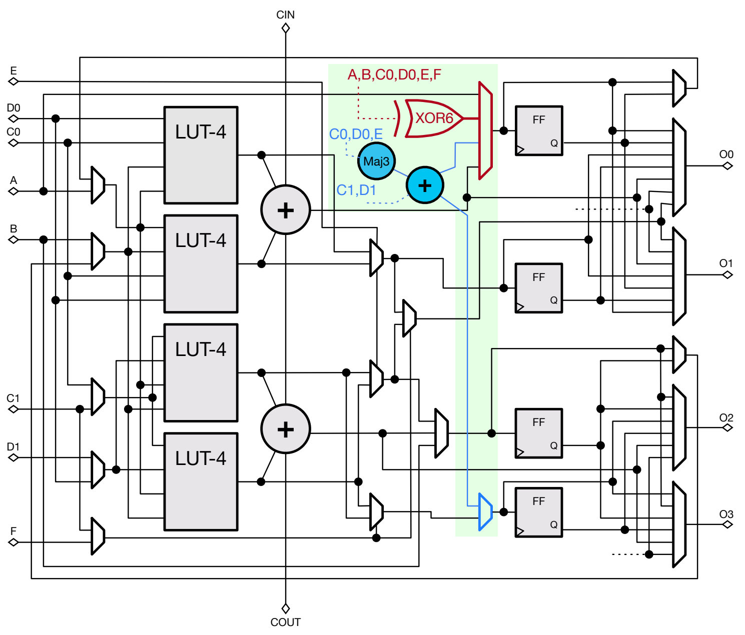
In this section, we describe in detail the proposed hardware architecture modifications that further improve the performance of GPCs on FPGAs. We focus our efforts on improving the design of the logic cell of FPGAs from the two major FPGA vendors, Intel and Xilinx. Our modifications are organized into two tiers: (1) A vendor-agnostic change to both Intel and Xilinx FPGA logic cells, and (2), a vendor-specific modification on top that further optimizes performance. We refer to these logic cell design tiers as LUXOR, and LUXOR+ respectively. Both LUXOR and LUXOR+ are backward-compatible and retain pin-interchangeability, *i.e.* any existing design maps equally well to these new architectures.

4.2.1 LUXOR

Our first proposed modification is to add a 6-input XOR gate (XOR6) to both Intel and Xilinx FPGA cells. The XOR6 is parallel to the LUT and re-uses its inputs and output path as shown in Figure 4.1. This modification is motivated by the observation that the C6:111 GPC is dominant in modern FPGA-based compressor tree designs. To quantify that claim, we analyzed optimal solutions of compressor trees from a set of 50+ micro-benchmarks that are commonly found in various domains (*e.g.* popcounting, multi-operand addition, FIR filters, etc) using efficient GPCs and compressors for Xilinx architecture from reference [222]. Figure 4.2 shows a histogram of the percentage count and cost (in LEs) for all GPCs across all solutions. Due to its compression efficiency, C6:111 is used more than a third of the time, and as a result, most of the hardware is dedicated towards its implementation. In modern FPGAs, the C6:111 maps to 3 LUTs, but by providing an explicit XOR6 datapath inside each logic



(A) Modifications to Xilinx UltraScale+ LE. A slice is composed of four LEs.



(B) Modifications to the Intel Stratix-10 ALM. Each ALM has 8-inputs and a fracturable LUT6 [42].

FIGURE 4.1. Basic LE for Xilinx and Intel FPGA architectures. LUXOR modifications are highlighted in red, while vendor-specific LUXOR+ modifications are coloured blue. Some signals are omitted for simplicity.

cell, we can bring that cost down to 2 LEs. This is done by mapping the first output bit to

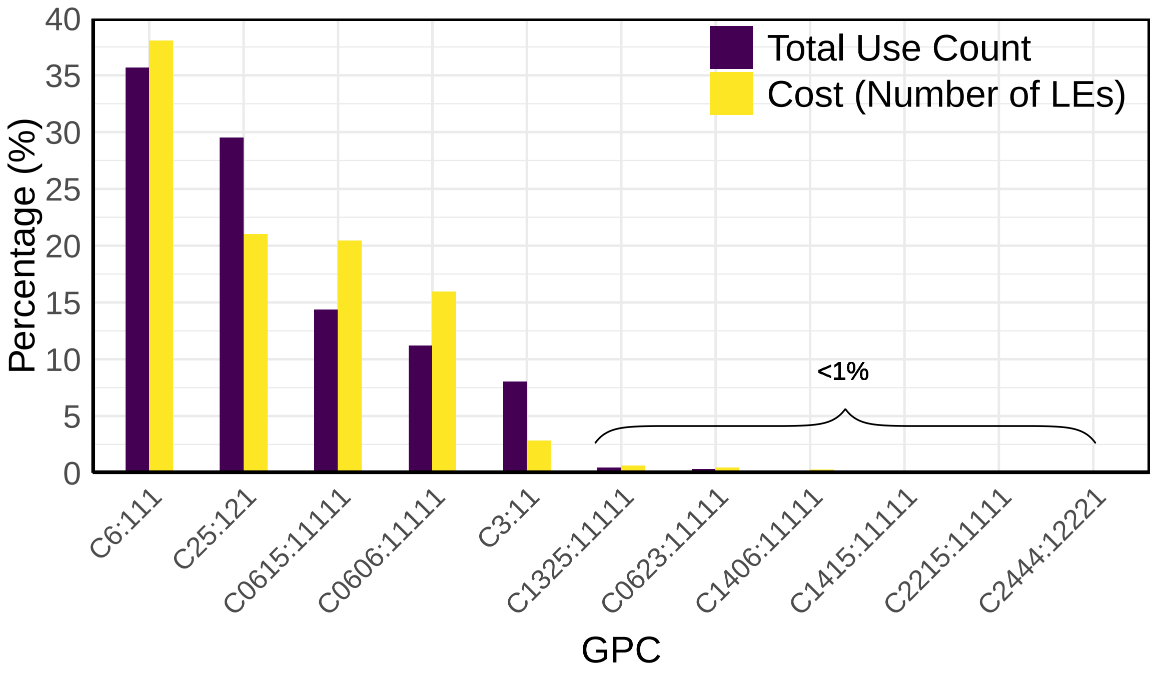


FIGURE 4.2. Total percentage count and cost of each GPC/compressor found in optimal solutions of compressor trees across 50+ Micro-Benchmarks from a variety of fields. The GPC/compressor list is according to [222]

the XOR6 rather than using a separate LE. Hence, LUXOR can deliver a resource utilization reduction for the most commonly-used GPC of up to 33$\%$.

Another very useful feature of the LUXOR design is its applicability to BNNs. In BNNs, the core computational workload is generated by the convolution layers, which are reduced via a XnorPopcount [17] operation for the binary case. Consider the XnorPopcount operation between three binary activations (x_0, x_1, and x_2) and their corresponding binary weights (w_0, w_1, and w_2). The required computation is:

$$\text{Sum} = (w_0 \overline{\oplus} x_0) \oplus (w_1 \overline{\oplus} x_1) \oplus (w_2 \overline{\oplus} x_2)$$

$$\text{Carry, C} = (w_0 \overline{\oplus} x_0) \cdot (w_1 \overline{\oplus} x_1) + (w_2 \overline{\oplus} x_2)[(w_0 \overline{\oplus} x_0) \oplus (w_1 \overline{\oplus} x_1)]$$

where $\overline{\oplus}$ and $\oplus$ represent the XNOR and XOR operations respectively.

This XnorPopcount operation gets mapped to 2 LEs on modern FPGAs, as shown in Figure 4.3A – one LE to compute the sum bit, and the other to compute the carry bit. With LUXOR, however, this computation can be mapped to just a single logic element via a

Boolean transformation, where the Sum bit (S) can now be expressed as:

$$\text{Sum} = (\overline{w_0} \oplus x_0) \oplus (\overline{w_1} \oplus x_1) \oplus (\overline{w_2} \oplus x_2)$$

which is essentially a XOR6 function where the complement of the weights are used. The LUT-6 implements the carry logic in this case, and both outputs from a single Xilinx slice can now be used to compute the partial products of the binarized convolution layer (see Figure 4.3B). Finally, to compute the output activations of the convolution layer, all the partial sums have to be summed, which can be visualized as a tall two-column many-operand instruction of carry and sum bits, as shown in Figure 4.3C. This can be efficiently reduced using a compressor tree, which is also improved by our proposed LUXOR modifications.

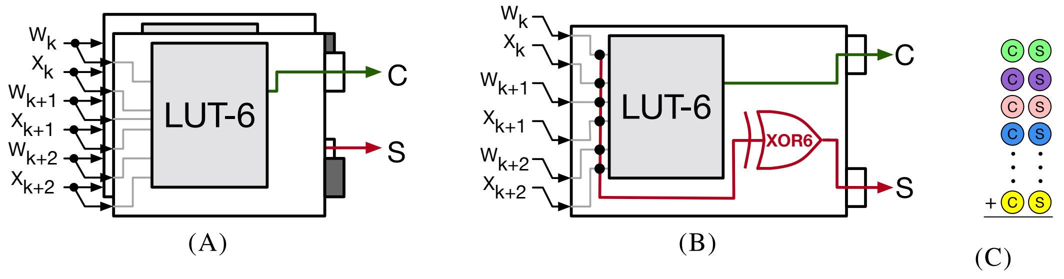


FIGURE 4.3. BNN implementation on Xilinx FPGAs: primary multiply and compressors of (A) XnorPopcount with 2 LEs, (B) XnorPopcount with 1 LUXOR LE. (C) Final two-column popcount to accumulate the partial sums (S), and carries (C)

4.2.2 LUXOR+

4.2.2.1 LUXOR+ for Xilinx FPGAs (X-LUXOR+)

Reference [222] proposes the *atoms* (–06–, –14–, –22–), as primitives to construct slice-based GPCs. Atoms are 2-column-input GPCs which mapped well to half of a slice (2 LEs) and can be connected via fast in-slice carry-chains to form wider GPCs, called couple. Note, the first atom in a couple can also accept one extra input in the first rank, except –06– for structural reasons. For instance –06– and –22– atoms builds two couples as C0623:11111 and C2206:11111. All combinations of these three atoms as well as C1325:11111 (which is also

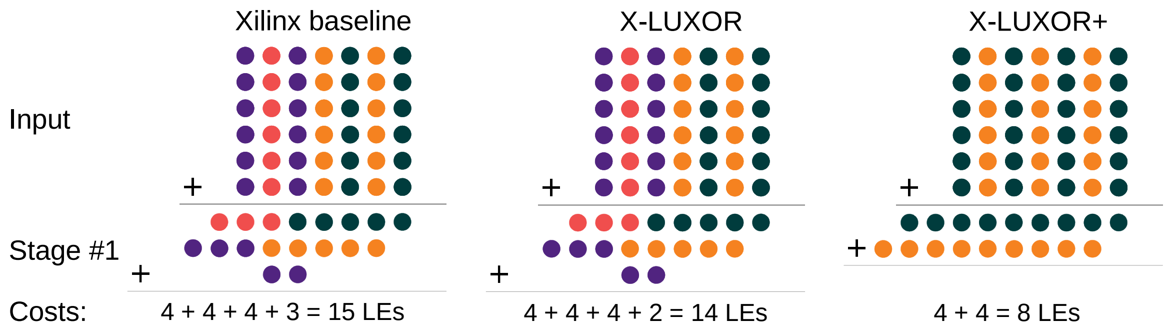


FIGURE 4.4. Example compressor tree for a 6-operand 7-bit addition using Xilinx baseline, X-LUXOR, and X-LUXOR+

a slice-based GPC but not decomposable) are listed in the baseline section of Table 4.1. To embrace the atom based structure, in cases where an –06– atom is placed in the higher rank (like C0615:11111), the preceding zero is not removed from the GPC names.

The blue datapath in Figure 4.1A highlights the proposed modification to the Xilinx FPGA slice. It involves modification of the carry chain datapath, introducing additional logic to allow the output from the XOR6 gate to be propagated into the carry-chain. This allows us to improve the implementation of slice-based GPCs. This enables us to map atom –06– to a quarter slice and consequently offers new set of slice-based GPCs such as C06060606:111111111, which can be mapped to just a single slice. This particular GPC has a very high compression efficiency of 3.75, which is more than any other existing GPCs in the literature. The X-LUXOR+ portion of Table 4.1 summarizes the characteristics of the new GPCs for Xilinx FPGAs.

We provide a simple illustration of the impact of our X-LUXOR and X-LUXOR+ optimizations in Figure 4.4. The penultimate (red) column can be implemented with a C6:111 compressor, requiring 2 LEs (instead of 3 in the unmodified case) in X-LUXOR. X-LUXOR+ is able to use the C06060606:111111111 GPC, which further reduces resource usage. In general, X-LUXOR has the greatest impact on tall-skinny compressor trees, which require significant use of C6:111, and hence has greater gains for wide compressor trees.

TABLE 4.1. Slice-based GPCs for Xilinx FPGAs. N.B. X-LUXOR+ area overhead is not considered in computing efficiency ($\boldsymbol{E}$), strength ($\boldsymbol{S}$), and arithmetic slack ($\boldsymbol{A}$) described in Equations 2.8, 2.9, 2.11.

	GPCs	p	q	**LUTs**	$\boldsymbol{E}$	$\boldsymbol{S}$	$\boldsymbol{A}$
Baseline [222, 196]	C0606:11111	12	5	4	1.75	2.40	0.031
	C1415:11111	11	5	4	1.50	2.20	0.000
	C2215:11111	10	5	4	1.25	2.00	0.000
	C0615:11111	12	5	4	1.75	2.40	0.000
	C1423:11111	10	5	4	1.25	2.00	0.000
	C2223:11111	9	5	4	1.00	1.80	0.000
	C0623:11111	11	5	4	1.50	2.20	0.000
	C1406:11111	11	5	4	1.50	2.20	0.031
	C2206:11111	10	5	4	1.25	2.00	0.031
	C1325:11111	11	5	4	1.50	2.20	0.063
X-LUXOR+	C06060606:111111111	24	9	4	**3.75**	**2.67**	0.002
	C140606:1111111	17	7	4	2.50	2.43	0.008
	C220606:1111111	16	7	4	2.25	2.29	0.008
	C060606:1111111	18	7	4	2.75	2.57	0.008
	C060615:1111111	18	7	4	2.75	2.57	0.000
	C060623:1111111	17	7	4	2.50	2.43	0.000
	C061406:1111111	17	7	4	2.50	2.43	0.008
	C062206:1111111	16	7	4	2.25	2.29	0.008

4.2.2.2 LUXOR+ for Intel FPGAs (I-LUXOR+)

Note that in Figure 4.2, the C25:121 GPC, originally suggested in [222], is also a very efficient. Figure 4.5 shows that it can be implemented using two sets of two 5-shared-input functions, occupying 2 ALMs. I-LUXOR+ introduces a majority circuit and full-adder to the ALM datapath, called MajFA (blue in Figure 4.1B), to explicitly implement S1 and C1 while S0 and C0 can be implemented in parallel with two 5-input LUT which shares the inputs in a ALM. This modification captures C25:121 in a single ALM instead of two. In summary, I-LUXOR+ reduces the cost of two highly used GPCs, C6:111 and C25:121, by one LUT (33% and 50% respectively).

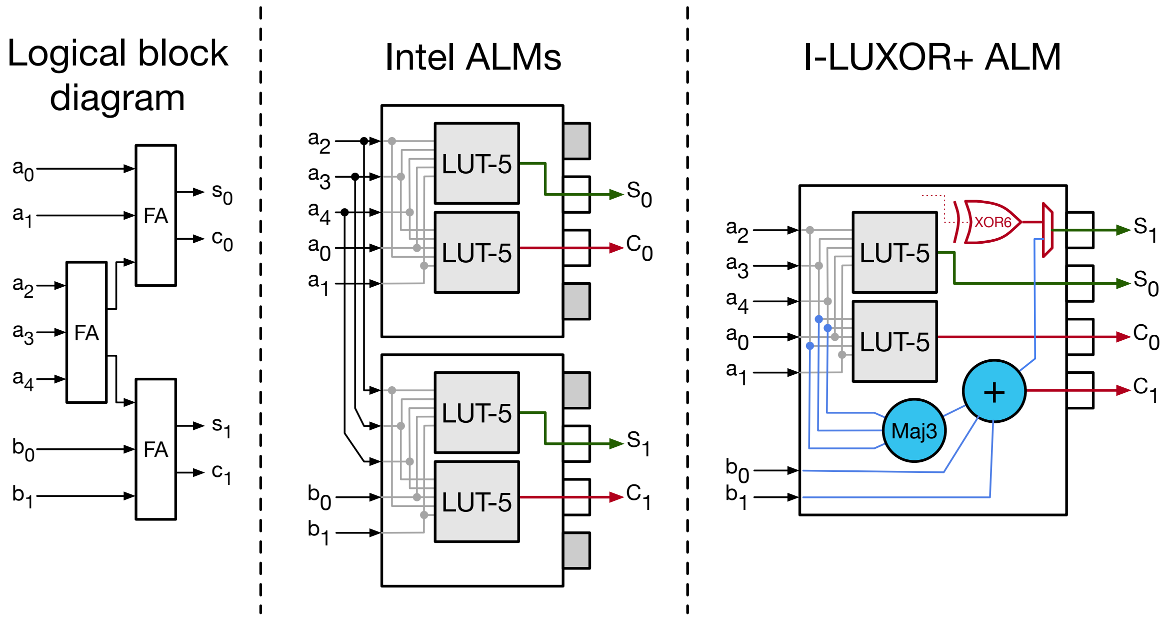


FIGURE 4.5. Efficient implementation of C25:121 GPC

4.3 ILP-based Compressor Tree Synthesis

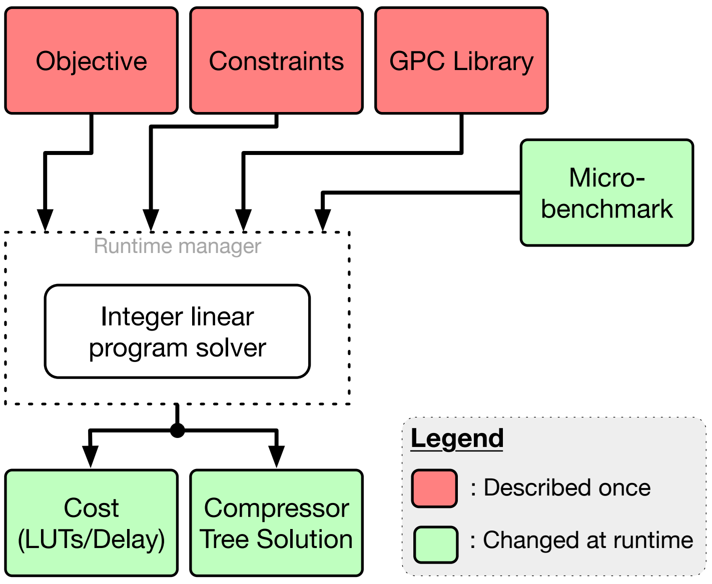


FIGURE 4.6. Flowchart of ILP-based compressor tree synthesis

Many commonly used arithmetic operations such as multiplications, multiply-add, or digital filters can be expressed compactly as compressor tree hardware implementations. However,

TABLE 4.2. Variables used in the ILP model

Var	Description
St	Number of stages in model
C	Maximum number of columns in model
X_c	Number of bits in column c of benchmark
T	Total number of compressors used
I_t	Total number of columns consumed by compressor t
V_t	Cost (in LUTs) of compressor t
$M_{t,c}$	Number of bits consumed by compressor t in column c
O_t	Total number of columns output by compressor t
$K_{t,c}$	Number of bits output by compressor t in column c
$N_{s,c}$	Number of bits in stage s of column c
$C_{s,c}$	Number of carry-bits in stage s of column c
$R_{s,t,c}$	Number of compressor t used in column c of stage s

realizing efficient compressor trees is a non-trivial task that typically requires software automation. Methods to do efficient compressor tree synthesis include heuristics-guided search [222, 243, 244], ILPs [228, 197], or hybrid approaches [223]. We opt for the ILP method in this work, and use ideas from [197] and [222] as inspiration. Our goal is to quantify the effect of our proposed LUXOR/LUXOR+ modifications on efficient compressor tree synthesis for commonly-used arithmetic operations in modern applications. Figure 4.6 encapsulates the workflow of our ILP formulation, and we detail each building block shown in the figure. Table 4.2 serves as a reference for all the variables used in this section. Note that, for clarity, all variable names in Table 4.2 are local to this section, and should not be confused with nomenclature in other sections.

4.3.1 Objective

There are two key metrics that quantify the effectiveness of a compressor tree implementation on FPGAs: area utilization in LUTs and the critical path delay, which is strongly correlated to the number of stages in the compressor tree. Hence, the objective function to an ILP program should be described in a way that minimizes these two metrics for each input micro-benchmark. To minimize the area cost, the objective function can be written as follows:

$$\min \sum_{s=0}^{St-1} \sum_{c=0}^{C-1} \sum_{t=0}^{T-1} V_t R_{s,t,c} \tag{4.1}$$

which sums the LUT-costs of all compressor instances (from all T types) placed in different columns at all logic circuit levels, called stages.

To model the number of stages in the objective function, the authors in [197] add the number of stages (St) as a heuristic to the cost function. However, we found this optimization strategy to be slow for difficult problems, and in some cases, the solver returns a solution that takes more stages than required. To tackle this issue, we design a runtime manager that improves the speed of the optimization process.

4.3.2 Runtime Manager and Solver

Instead of modeling St as a heuristic in the objective function, we rely on an iterative approach where we query the solver to find an optimal solution within a fixed maximum stage limit, St$_{max}$. This limit is relaxed incrementally until a feasible solution is found. In practice, we found that the solver was able to determine infeasibility within a few seconds, whilst being able to find a feasible integer solution within a few minutes. This iterative approach was also recently used by Kumm et al. [223] by combining the ILP optimality search with heuristics to guide the solver. We use the IBM CPLEX v12.9 [245] ILP solver (under academic license), and design a Python3-based interface for the runtime manager using the PuLP package [246].

4.3.3 Constraints

Since the input stage captures the input shape of the benchmark, we set constraints on the input stage as follows:

$$N_{0,c} = X_c \qquad \text{for } c = 0,1,2,...,C-1 \tag{4.2}$$

For subsequent stages, there are two constraints required to guide the solver towards a feasible compressor tree architecture, such that input/output requirements of each stage are met:

$$\sum_{t=0}^{T-1}\sum_{c'=0}^{O_t-1} M_{t,c'} * R_{s-1,t,c-c'} \geq N_{s-1,c} \tag{4.3}$$

$$\sum_{t=0}^{T-1}\sum_{c'=0}^{I_t-1} K_{t,c'} * R_{s-1,t,c-c'} = N_{s,c} \tag{4.4}$$

for c = 0, 1, 2, ..., $C-1$ and s = 1, 2, 3, ..., $St-1$.

The first constraint ensures that all bits in each column of every stage are used as inputs by compressors in the next stage. The second constraint ensures that the number of bits produced by the compressors in the previous stage matches the number of input bits in the following stage. Both these constraints can also be found in [197].

In each stage, the number of carry-bits in each column are computed in (4.5), where the division by two is due to the increase in the column's radix.

$$C_{s,c} = \left\lfloor \frac{C_{s,c-1} + N_{s,c-1}}{2} \right\rfloor \tag{4.5}$$

This can be formulated as an ILP constraint as follows:

$$C_{s,c} + 0.999 \geq \frac{1}{2}(C_{s,c-1} + N_{s,c-1}) \tag{4.6}$$

$$C_{s,c} \leq \frac{1}{2}(C_{s,c-1} + N_{s,c-1}) \tag{4.7}$$

$$C_{s,0} = 0 \tag{4.8}$$

for c = 0, 1, 2, ..., $C-1$ and s = 1, 2, 3, ..., $St-1$. Note that the number of input carry-bits into the first column is always set to 0.

When solving the model iteratively, as described above, the constraints on the final stage guide the solver to converge to the solution. In [222], the author proposes a novel ragged carry-propagate architecture for the final accumulation stage for Xilinx FPGAs. This architecture reduces the overall number of stages required, and hence, we opt for this strategy on Xilinx FPGAs. Unlike [222], where the author uses a heuristic solver, we model the ragged carry-propagate adder into our model for the final stage as three constraints:

$$N_{s,c} + C_{s,c} \leq 5 \tag{4.9}$$

$$C_{s,c} \leq 2 \tag{4.10}$$

$$N_{s,c} \leq 4 \tag{4.11}$$

when c = 0, 1, 2, ..., $C-1$ and s = $St-1$.

Finally, since Intel FPGAs cannot benefit from the ragged carry-propagate adder, we model the ILP constraints for Intel FPGAs as shown in [197] :

$$N_{s,c} \leq 3 \tag{4.12}$$

when c = 0, 1, 2, ..., $C-1$ and s = $St-1$.

4.3.4 GPC/Compressor Library

4.3.4.1 Xilinx Compressor Set

When targeting Xilinx architectures for our baseline, we use the GPC/compressor set defined by Preußer [222], who pruned a set from Kumm and Zipf [196]. For our LUXOR experiments, we reduce the cost of C6:111 GPCs from 3 to 2 logic elements, as described in Section 4.2.2.1. For LUXOR+, in addition to the smaller version of C6:111, we add all the new slice-based GPCs described in Table 4.1 to our model. We denote these results as X-LUXOR and X-LUXOR+ respectively.

TABLE 4.3. Comparison of different GPCs proposed in [222] and new GPCs supported by I-LUXOR and I-LUXOR+ (Strength ($\boldsymbol{S}$), and arithmetic slack ($\boldsymbol{A}$) are described in Equations 2.9 and 2.11).

GPCs		$\boldsymbol{S}$	$\boldsymbol{A}$	Delay	LUTs	APD
[225]	C6:111	2	0.13	0.38	3	7.9
	C15:111	2	0	0.38	3	7.9
	C23:111	1.67	0	0.38	2	5.3
[222]	C25:121	1.75	0	0.38	2	11.8
Ours	C6:111	2	0.13	0.39*	2	10.95*
	C25:121	1.75	0	0.39*	1	21.9*

*Area/delay overheads for I-LUXOR+ are included (Section 4.4).

4.3.4.2 Intel Compressor Set

When targeting Intel architectures, our baseline compressor set is based on a GPC set proposed by Parandeh-Afshar et al. [225], augmented with the C25:121 compressor from [222]. Since this GPC set is large, to minimise run-time of our ILP, we pruned this set using the GPC selection approach and metric described by Preußer [222].

Parandeh-Afshar et al. [225] have gathered a group of LUT-based and arithmetic-based GPCs for Intel architectures. In the first three line of Table 4.3, we show the efficiency and compression metrics of our selected GPCs according to the Area-Performance Degree (APD) (Equation 2.10) metric, which measures the efficiency of a GPC taking into account delay and resource usage. We also considered the delay itself, since some of the proposed GPCs, such as C7:111, offer slightly better S (compression rates), but their reported delay is 3.5$\times$ greater. In addition, we included C3:11 and C25:121 in the baseline GPC set for Intel architecture.

Similar to our X-LUXOR experiments with Xilinx architectures, we reduce the cost of C6:111 GPCs from 3 to 2 logic elements for I-LUXOR. For I-LUXOR+, as well as using the upgraded version of C6:111, we reduce the cost of C25:121 from 2 to 1 LE as described in Section 4.2.2.2. We also comment that the effect of the I-LUXOR and I-LUXOR+ enhancements are highlighted by the metrics, as demonstrated by the last two rows of Table4.3. Due to the lower logic element cost, the APD of both GPCs show significant improvement.

4.3.5 Micro-Benchmarks

To evaluate the improvements of our proposed architectures, we use different basic operations that are commonly found in various domains in three categories: 1) Low-rank inputs including pop-count and two-column count (based on [222], but with additional input sizes) 2) High-rank inputs including multi-addition [197], 3-MAC operation (described below), and a FIR-3 filter from [225], and 3) BNN XnorPopcount operation for various input sizes, where the filter sizes are taken from the networks in [242, 247]. These three categories highlight the benefits and limitations of LUXOR and LUXOR+ architectures, as the chosen operations appear in various applications, especially digital signal processing and neural networks, which are the most important concerns of new FPGA architectures [193, 1].

4.3.5.1 3-MAC Operation

The 3–MAC operation is modeled according to the following equation:

$$\text{3-MAC}_{(N\times N-bit)} = \sum_{i=0}^{2} A_{i(N-bits)} \times B_{i(N-bits)} \tag{4.13}$$

Note that since there are 3 pairs of inputs, instead of computing partial products then and summing their results, we can select partial products of the same rank and perform a primary compression. The cost of this step is included in our result. The resulting tree forms the input to the compressor. We repeat this for different input widths (N).

4.4 Results

In this section, we present results from experiments undertaken to evaluate the performance of the LUXOR and LUXOR+ architectural enhancements.

4.4.1 ASIC Modeling: Delay and Area Overheads

We model state-of-the-art Intel Stratix-10 ALM unit [42], and Xilinx UltraScale+ slice [192, 248] according to their respective data sheet descriptions. For the ASIC metric analysis, we synthesise our Verilog models using SMIC 65-nm technology standard cell by Synopsis Design Compiler 2013.12. Modeling with standard cell-based designs have certain limitations when compared to full-custom commercial ASICs. While standard cells excel in offering efficient logic gates, they are not as effective for multiplexing circuits. However, leveraging standard cells enables practical design exploration. Also, the LUT SRAM cells are modeled by registers. Higher efficiency would be expected in a full-custom design. Post-synthesis results are reported and the synthesis strategy was set to “Timing Optimisation” since it usually leads to a better $Area \times Delay$ product. We note that while our approach to estimating area and delay overheads using standard cells may differ slightly from a commercial full custom layout, in either case, the overhead is minimal.

Table 4.4 gives the post-synthesis area and timing results for the Intel baseline, I-LUXOR, Intel+MajFA and I-LUXOR+ modifications to the ALM. From the table, it can be seen that the delay increase of I-LUXOR is about 1% while the area increase is less than 0.5%. This demonstrates that there is little overhead associated with adding a 6-input XOR gate to the ALM unit. In contrast, adding MajFA circuits will increase the area and delay by 2% and 5% respectively (see description in Section 4.2.2.2). The full I-LUXOR+ implementation has 3% and 5% delay and area overhead, respectively. We believe that the unexpectedly large increase in area compared to the individual effect of each modification arises from the performance-driven synthesis optimisation. For measuring the critical path, we removed the multiplexers connecting the ALM’s outputs to its input, and thus it is measured from: an input, through a LUT and two-coupled FAs to an output multiplexer.

In a Xilinx slice, the critical path is from an input, passing through the first LUT (A) and four carry-chain circuits and ending with the last output multiplexer. This path is also the critical path after applying LUXOR(+) for both architectures.

TABLE 4.4. ASIC results for the Intel Stratix-10 ALM architecture

		Intel	I-LUXOR	Intel+MajFA	I-LUXOR+
Area	um^2	1680	1687	1715	1767
	ratio	1	1.00	1.02	1.05
Delay	ns	1.42	1.44	1.49	1.46
	ratio	1	1.01	1.05	1.03

TABLE 4.5. ASIC results for the Xilinx UltraScale+ slice architecture

		Xilinx	X-LUXOR	X-LUXOR+
Area	um^2	6045	6002	6361
	ratio	1.00	0.99	1.06
Delay	ns	0.84	0.89	0.92
	ratio	1.00	1.06	1.09

The synthesised Xilinx baseline slice model has an area of 6045 um^2. We compare the reported critical path with that from the Virtex-5 datasheet, which was a device that was also manufactured with a similar 65 nm process. Reference [233] reports the critical path from an input, through four carry circuits to the output (T_{ITO}) as 0.67, 0.77, or 0.90-ns for three different speed grades. Comparing these values with our value of 0.84 ns from Table 4.5, consistency with our synthesis results was verified. The same table shows that X-LUXOR has similar area utilisation and a 6% increase in delay, while X-LUXOR+ has 6% area and 9% delay overheads.

Since the routing delay strongly contributes to the total delay, the LUXOR(+) delay advantages are diluted in practice. Although the X-LUXOR+ overheads are notable, because of the significant resource and performance benefits, new trade-offs are offered. For example, partially upgrading the LEs to LUXOR(+) architectures is another option. Also, with more effort in layout and buffer sizing, area and delay overheads can be recovered/balanced.

At a higher level of abstraction, LUXOR(+) does not require any I/O scheme modifications. However, they increase the logic implementation density leading to higher connectivity per LE/ALM. Thus, routing limitations may slow down LUXOR(+) enhancements. LUXOR(+) adds to the input load, which also slows down the LE. This was not measured directly but taken into account in the LE measurements.

TABLE 4.6. A comparison of solutions from our ILP-based synthesis compared with those reported in [222]

Test cases	[1]H1/H2/H3[222]		Our ILP Solver					
			Baseline		X-LUXOR		X-LUXOR+	
	LE	[2]Stage	LE	Stage	LE	Stage	LE	Stage
S128	101/102/101	4/3/4	100	3	79	3	78	3
S256	209/209/209	4/4/4	195	4	159	4	154	4
S512	418/422/418	5/5/5	380	5	319	5	312	4
D128	178/205/178	5/4/5	168	4	156	4	150	4
D256	360/417/360	6/5/6	328	5	315	5	298	4
D512	721/839/721	7/6/7	709	5	631	5	586	5

[1]Heuristics used in [222]: Efficiency/Strength/Product, reported in that order.

[2]Stage = # of compressor tree stages

4.4.2 Benchmark Performance

The effect of our ILP approach on resource utilisation in logic elements is affected by the choice of primitives in the primary stage (if applicable), compression tree stages, and the last stage (final ternary adder in Intel or the equivalent relaxed ternary adder for Xilinx architectures as proposed in [222]). Table 4.6 compares our technique with that of [222] for X-LUXOR and X-LUXOR+ where test cases are popcount and double column popcount operations indicated respectively by S and D, concatenated with input size. As can be seen in the baseline column, our ILP approach uses fewer LEs and stages for all benchmark problems compared with the heuristic approach, since an optimal solution is found. While the X-LUXOR enhancement significantly reduces the number of LEs compared with the baseline, X-LUXOR+ achieves a further reduction in the number of stages.

Figure 4.7 shows the savings in LEs for Xilinx architectures over a larger micro-benchmark set, with the red star also indicating a reduction in the number of stages by one. For low-rank inputs (i.e. popcount and two-column popcount), the C6:111 and C25:121 compressors are heavily used. X-LUXOR improves the resource efficiency of C6:111 implementations and achieves the best savings for the 1024-input popcount problem at a 22% reduction. Less improvement is seen for two-column popcount, as in the first stage, C25:121 has better arithmetic slack (A) while offering the same efficiency. This observation was also made in [242]. X-LUXOR+ offers a new set of the state of the art compression rate and compression

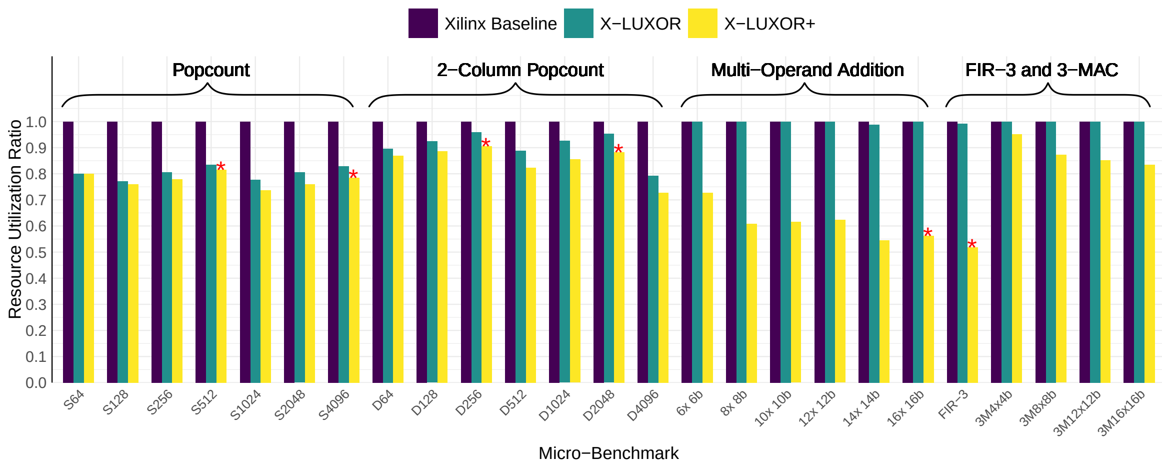


FIGURE 4.7. Resource reduction on Xilinx UltraScale+, X-LUXOR, and X-LUXOR+ architectures for various micro-benchmarks. The * indicates that the proposed solution required one less logic stage in the compression tree.

efficiency. On average, X-LUXOR+ can reduce area utilisation on the low-rank input popcount and two-column popcount benchmarks by 22% and 15%, respectively.

For the high-rank benchmarks (multi-operand addition, FIR-3 and 3-MAC), the inputs are wide enough to benefit from the slice based GPCs. The C6:111 compressor is not significantly utilised. However, X-LUXOR+ offers higher compression rates and hence achieves 39% and 18% improvement in multi-addition and 3-MAC benchmarks, respectively, and in some cases, the required number of stages is also reduced by one.

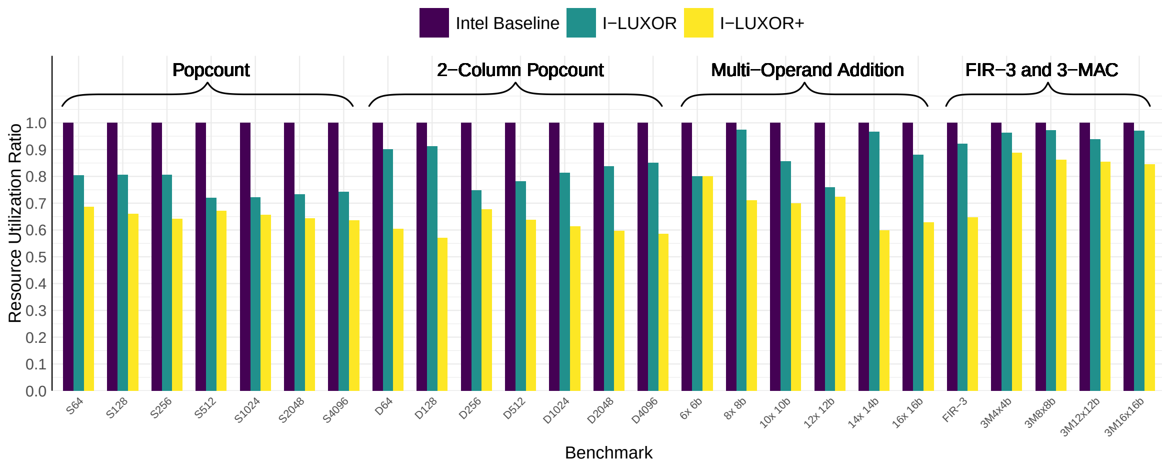


FIGURE 4.8. Resource reduction on Intel Stratix-10, I-LUXOR, and I-LUXOR+ architecture for our selected micro-benchmarks.

Figure 4.8 shows the same result for Intel I-LUXOR and I-LUXOR+ architectures. More dramatic resource savings are apparent over Xilinx, particularly for low-rank problems using I-LUXOR+. Since I-LUXOR and I-LUXOR+ do not present new compressors, no reduction in the number of stages is achieved. However, because the baseline offers no wide GPC, the resource reduction of I-LUXOR is more significant (averaging 24% and 17% for popcount and double popcount). I-LUXOR+ offers an enhanced C25:121 GPC which is the most efficient GPC for the Intel architecture. This leads to 35% and 39% resource savings for popcounting and two-column counting. As results suggest, Intel's LE architecture benefits more from both LUXOR and I-LUXOR+ modifications. This is mainly due to the limited number of efficient parallel counters in its baseline LE. Our suggested alterations significantly enhance the efficient GPC set for this architecture.

4.4.3 Performance on BNNs

Binarised neural networks offer a new challenge for FPGA architectures as 1-bit multiply-accumulate operations require XNOR and popcount operations to be efficient. As explained in Section 4.2.1 the first computation stage (Multiplication) should be merged with the early compression circuits, leading to an efficient implementation (as illustrated in Figure 4.3(a)). If the number of input pairs is N, $N/3$ fused units are required in the primary stage. LUXOR can implement this fused computation using a single LE rather than two LEs in the baseline architectures leading to $N/3$ fewer LE utilisation. In addition, after the implementation of the primary stage, a two-column counting problem with the height of $N/3$ is encountered.

As shown in Figure 4.9, these two optimisations lead to almost the same 34% resource reduction for LUXOR modification on both Xilinx and Intel architectures. Moreover, as described before, X-LUXOR+ cannot reduce the number of LEs significantly for low-rank inputs, and hence, the best area savings for BNNs plateaus at 37%. In the case when the input size is 3×3×256, the number of stages is reduced by one, which would give us a significant improvement in delay. In comparison, I-LUXOR+ reduces the number of LEs significantly at an average of 47%, but without reducing the number of stages.

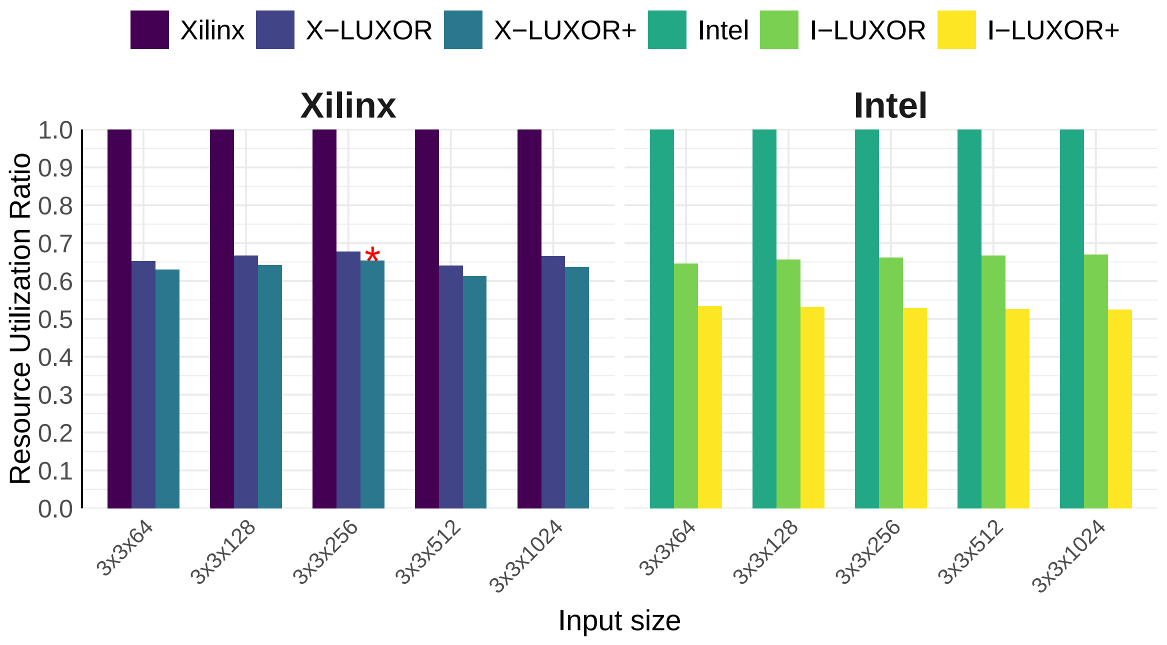


FIGURE 4.9. XnorPopcount micro-benchmarks found in BNN Convolution Layers in [29, 247]

4.5 Summary

This chapter has discussed several low-cost FPGA logic cell modifications that can lead to significantly improved performance GPCs and the XnorPopcount operation. By adding these primitives to a set of state-of-the-art compressor tree primitives (adders, GPCs, and compressors) described in the literature, an ILP model for finding optimal solutions for FPGA-based compressor tree implementations for both Xilinx and Intel FPGAs is built. Using this ILP, it is shown that proposed modifications lead to substantial performance gains. LUXOR is a vendor-agnostic modification, which augments each logic cell datapath with a dedicated 6-input XOR circuit, reduces the cost of the commonly used (C6:111) GPC from 3 LEs to just 2 and enables efficient XnorPopcount implementations. Over a benchmark set, this reduces the logic utilisation cost of compressor trees by up to 36% (average 12–19%) on both Intel and Xilinx FPGAs, with a silicon area overhead of $<$0.5%. The architectural re-design is taken a step further with LUXOR+, which proposes carefully crafted vendor-specific modifications. LUXOR+ requires an additional 3–6% silicon area, can improve our micro-benchmark results to up to 48% (average 26–34%). BNN benchmarks benefit the most with an average reduction of 37–47% in logic utilisation, which is due to the highly-efficient mapping of the XnorPopcount operation on our proposed LUXOR+ logic cells.

CHAPTER 5

MLBlocks: Rethinking Embedded Blocks for Machine Learning Applications

This chapter presents a systematic methodology for deriving efficient coarse-grained compute block architectures from a benchmark set of algorithms with the formulation of a MAC operation within a number of nested loops, together with a generic EB architecture resulting in a family of new embedded blocks, called MLBlocks. This enables automation in design space exploration for EB architectures in contrast to previous expert handcraft design suggestions.

The presentation of this chapter is based on the background given on deep neural networks, and FPGA architectures in Chapter 2 and expands on work previously published in [3].

5.1 Introduction

The ultimate goal in FPGA architecture design is to provide a reconfigurable and flexible platform that can implement a wide variety of circuits. Existing commercial FPGA architectures are mature and have evolved from decades of optimisation for traditional networking, image processing and signal processing applications. Typically, these circuits use relatively high-precision arithmetic and have been addressed by DSP units that support high-precision MAC operations.

Recent research has shown the efficacy of low-precision fixed point and block floating point [249] arithmetic for the implementation of DNNs in both inference [250] and training tasks [127]. This created a new set of demands characterised by: 1) higher computational density and 2) low precision arithmetic requirements. However, these MAC operation-dominated circuits do not map efficiently to the contemporary FPGA resources [43]. In particular, DSPs are heavily underutilised for low-precision DNNs. While the Xilinx DSP48E2 is capable of executing a 27×18 multiply and 48-bit accumulate operation, for the low precision case, it only delivers two 8×8 multiplies (with shared multiplicand) with 24-bit accumulation. This is roughly a third of its potential since a 27×18-bit multiplier occupies the area of roughly six 9×9-bit ones [44]. As a result, even in a state-of-the-art accelerator, DSPs impose a performance limit [45]. The Intel DSP blocks also have similar limitations.

This inefficiency causes a barrier to higher performance low-precision implementations compared to GPUs, e.g., for INT8 operations, the embedded Jetson Xavier NX GPU [251] offers 21-TOPs which would require a high-end Virtex UltraScale+ FPGA with 6,840 DSPs for comparable performance [252]. Nonetheless, the importance of these solutions for compute-bound DNN applications is likely to grow in the future, which encourages new architectural studies aiming to improve FPGA performance with coarse-grained blocks that provide better support for low precision since any dedicated course-grained blocks are likely to outcompete any LUT-based solution.

Designing the EBs requires making tradeoffs between 1) flexibility to support a wide range of domains and 2) specialization to efficiently support selected applications. This is conventionally done by suggesting hand-crafted designs and then evaluating their utility over a set of benchmark problems. However, to date, the benchmarks for FPGA evaluation have yet to include low-precision, quantized DNN applications except for rare academic examples like [253, 254]. Furthermore, finding the optimum structure for a compute-dense unit that can effectively handle multiple configurations while maintaining certain design constraints, such as limited IO budget, circuit area, operating frequency, etc., is a complex task that becomes tedious by repetitive manual try-and-evaluate steps (e.g. the presented design in Chapter 3).

The missing solution is a benchmark-driven architecture search automation. However, developing such blocks in a systematic fashion requires: 1) a generalized problem formulation that covers the relevant range of computations, and 2) an efficient mapping to configurable architectures. With the goal of exploring this new design space in a methodical manner, this Chapter presents a problem formulation involving computing nested loops over MAC operations, which covers many basic linear algebra primitives and standard DNN kernels. A quantitative methodology for deriving efficient coarse-grained compute block architectures from benchmarks is then proposed together with a family of new embedded blocks, called MLBlocks. Simply, the challenging design question that the MLBlock designing method addresses is, "What computations should a distributed course-grained FPGA block for low-precision support?"

An MLBlock instance includes several multiply-accumulate units connected via a flexible routing, where each configuration performs a few parallel dot-products in a systolic array fashion. This architecture is parameterized with support for different data movements, reuse and precisions, utilizing a columnar arrangement that is compatible with existing FPGA architectures. On synthetic benchmarks, it is demonstrated that for 8-bit arithmetic, MLBlocks offer $6\times$ improved performance over the commercial Xilinx DSP48E2 architecture with smaller area and delay; and for time-multiplexed 16-bit arithmetic, achieves $2\times$ higher performance per area with the same area and frequency.

5.1.1 Previous Works

Compared to ASICs and GPUs, FPGAs suffer from lower maximum clock frequency and larger area, which subsequently impacts performance. A solution to these issues is to increase the compute capacity. Routing restrictions and efficiency make it impractical to add a sea of additional multipliers or MAC units to FPGA architectures, as programmable routing is both expensive and power-hungry. However, it is possible to include dense compute blocks while meeting the IOs and area budget limits.

As described in Chapter 2, solutions to this issue are categorised into three main tiers, where the MLBlocks method belongs to the group "designing a new embedded block". However, the MLBlocks approach differs from all introduced works as this technique systematically derives a near-optimal EB by considering the mapping of an arbitrary set of benchmarks of a certain design pattern onto a flexible MLBlock fabric. Considering nested loops as a generalisation for DNN implementation was first proposed by Yang et al. [31], and the presented approach of optimising designs by considering different tiling patterns over a set of benchmarks for DNNs is a generalisation of that described by Zhang et al. [255].

5.1.2 Contributions

This chapter presents the development of a tool to create a generalised EB architecture called an MLBLock, constructed from low-precision MAC units and programmable routing. The difficulty in its design lies in ensuring maximum utilisation of the MAC units over a benchmark set with minimal routing overhead. To achieve this, we introduce a methodology that allows all potential loop unrollings to be enumerated and analysed over a range of DNN computations. These computations include standard, depth-wise, dilated and point-wise convolutions, fully-connected layers, and recurrent neural networks (RNNs), including vanilla RNN, long-short-term memory (LSTM), and gated recurrent unit (GRU) layers [59]. Using this analysis, the tool generates an MLBlock instance that supports a subset of loop unrollings that best satisfy the area/performance trade-offs. This serves as the target architecture to which algorithms can be efficiently mapped.

Specifically, the contributions are as follows:

- A methodology in which different algorithms in the form of a benchmark set are mapped to MLBlocks. Algorithm descriptions are of the form of MAC operations within a number of nested loops that is a generalisation of convolution computation. Using a set of loop transformations that covers a solution space, projections with different hardware tradeoffs are generated, and a specific configuration of an MLBlock which supports all the benchmarks identified.
- A case study demonstrating the application of this methodology, with some restrictions, to automatically find a DSP-like replacement block that demonstrates higher performance and efficiency across a benchmark suite in comparison to existing expert-designed architectures.
- Confirmation that MLBlocks are suitable for implementation in the familiar columnar manner and compatible with existing FPGA architectures.

- Greedy and heuristic approaches to select an implementable projection set from the set of possible projections, which achieve a balance between flexibility and performance.
- A parameterised FPGA module generator for MLBlocks which compiles a projection set to Verilog, creating an instance that uses a specified number of MAC units (called MLBlock-M). The suggested EB architecture offers a bijection from a projection to an MLBlock configuration.
- A quantitative architectural study of the performance benefits of augmenting the Xilinx FPGA architecture with MLBlocks.

Open-source Python tools to implement the techniques described in this chapter, together with results, are available at `www.github.com/raminrasoulinezhad/MLBlocks`.

The remainder of the chapter is organised as follows. In Section 5.2, we describe the proposed design methodology having an algorithm template that covers a superset of DNN computations and many basic linear algebra subprograms (BLAS) functions, and techniques to analyse the potential computation of an EB, their costs and methods to find the right trade-off for different implementation candidates. Section 5.3 describes a generalised EB target architecture to which a set of projections can be mapped. Results are presented in Section 5.4 and conclusions in Section 5.5.

5.2 MLBlocks Design Methodology

5.2.1 Overview

An EB must support all algorithms in a user-specified benchmark suite and we assume all algorithms can be written as a set of nested constant loops. The innermost computation could be any arithmetic operation; however, in this paper, we only consider low-precision MAC operations with three inputs (**I**, **W**, and **O**) and one output (**O**). A template for our algorithm descriptions is given in Algorithm 1 (explained in detail in Section 5.2.2). While

Algorithm 1 Pseudo code for the generalized nested loop model (loop format for each $v \in \mathbb{V}$ is $v \leftarrow V^{init}$:V^{stride}:V^{limit})

```
for $g_0 \leftarrow 0 : G_0^{stride} : G_0^{limit}-1$
 for $g_1 \leftarrow 0 : G_1^{stride} : G_1^{limit}-1$
  ...
   for $g_n \leftarrow 0 : G_n^{stride} : G_n^{limit}-1$

    for $b_0 \leftarrow 0 : B_0^{stride} : B_0^{limit}-1$
     for $b_1 \leftarrow 0 : B_1^{stride} : B_1^{limit}-1$
      ...
       for $b_m \leftarrow 0 : B_m^{stride} : B_m^{limit}-1$

        for $e_0 \leftarrow 0 : E_0^{stride} : E_0^{limit}-1$
         for $e_1 \leftarrow 0 : E_1^{stride} : E_1^{limit}-1$
          ...
           for $e_p \leftarrow 0 : E_p^{stride} : E_p^{limit}-1$

            for $r_0 \leftarrow 0 : R_0^{stride} : R_0^{limit}-1$
             for $r_1 \leftarrow 0 : R_1^{stride} : R_1^{limit}-1$
              ...
               for $r_q \leftarrow 0 : R_q^{stride} : R_q^{limit}-1$

        $\mathbf{O}[g_0, g_1, .., b_0, b_1, .., e_0, e_1, ..] \mathrel{+}= \mathbf{I}[g_0, g_1, .., b_0 \pm r_0, b_1 \pm r_1, ..] \times \mathbf{W}[g_0, g_1, .., e_0, e_1, .., r_0, r_1, ..]$
```

straightforward, this is sufficient to describe a broad range of DNN related computations as demonstrated by our chosen benchmark suite in Section 5.4.

An overview of our approach to determine the optimal EB is given in Figure 5.1. Assuming a fixed unrolling factor (M), we generate a list of *unrollings* for each algorithm in the benchmark suite. We describe each unrolling instance through a compact representation that we call a *projection*. Each projection has a one-to-one mapping with a hardware EB configuration. We then perform *selection* by analyzing the list of all projections to identify a subset that will cover all algorithms in the given benchmark suite, maximise MAC utilisation, and minimise the implementation cost. We call the resulting subset the *selected projections*. In the generation stage, a tool merges all configurations to produce a Verilog description, namely an MLBlock-M.

5.2.2 Benchmark Algorithm Template

Algorithm 1 is our generalized algorithm template that takes advantage of the fact that in DNN layers and BLAS functions, many loop variables have similar data access patterns; these result

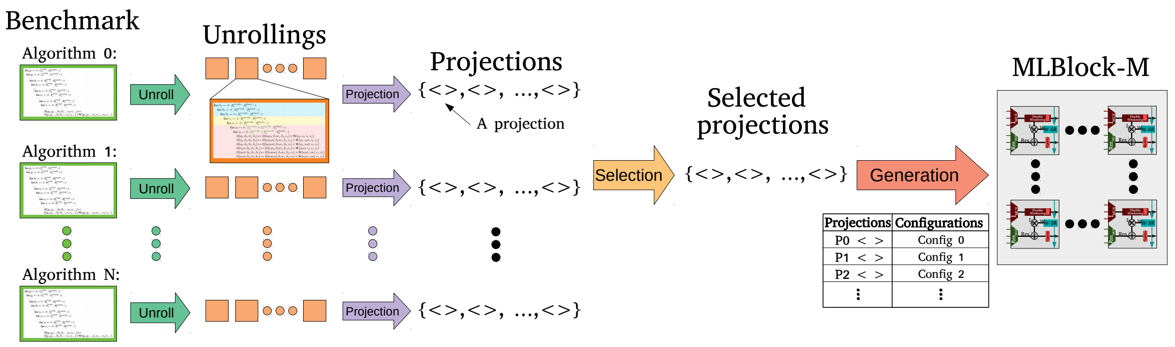


FIGURE 5.1. An overview of MLBlock generation for a given benchmark suite

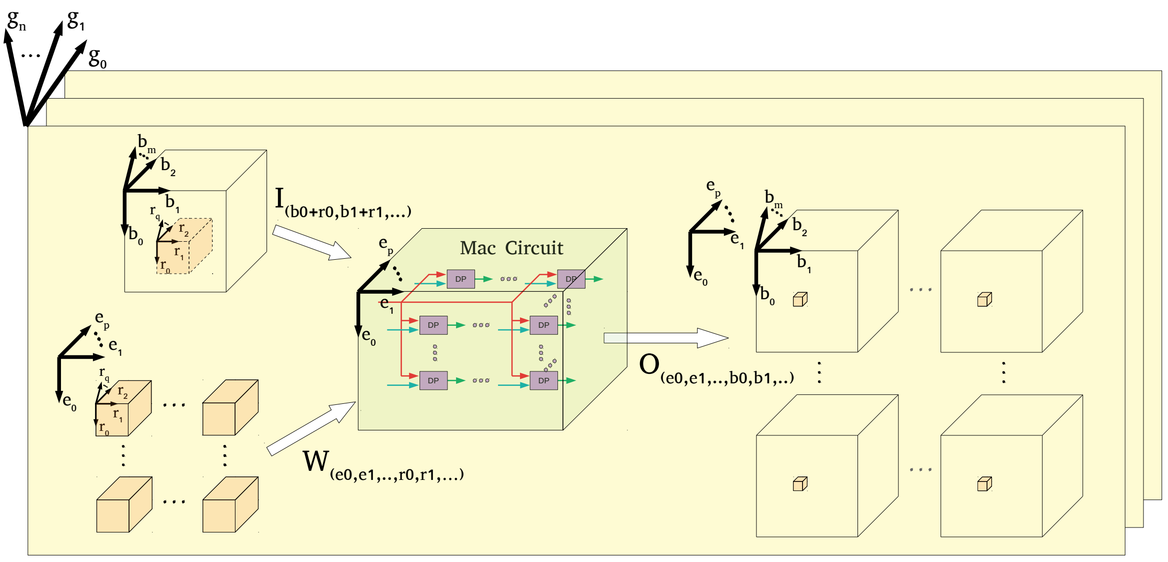


FIGURE 5.2. A visualization for algorithms covered by Algorithm 1.

in similar hardware realizations. To characterise these access patterns, we define that loop variables (r_i, e_i, b_i, g_i) $\forall i$, *accesses* an input/output if it reads the input/output. Considering MAC operation input and outputs, the loop variables can then be classified into one of four variable group sets $(\{R, E, B, G\})$, the first character being used to identify the variable names according to:

(1) **Reduction (R)**: The loop variables index elements of **I** and **W** to produce a single output in **O**, e.g. a dot-product.
(2) **Expansion (E)**: The loop variables index elements of **W** and **O** to produce multiple values of **O** reusing **I**, e.g. processing different kernels for the same input.

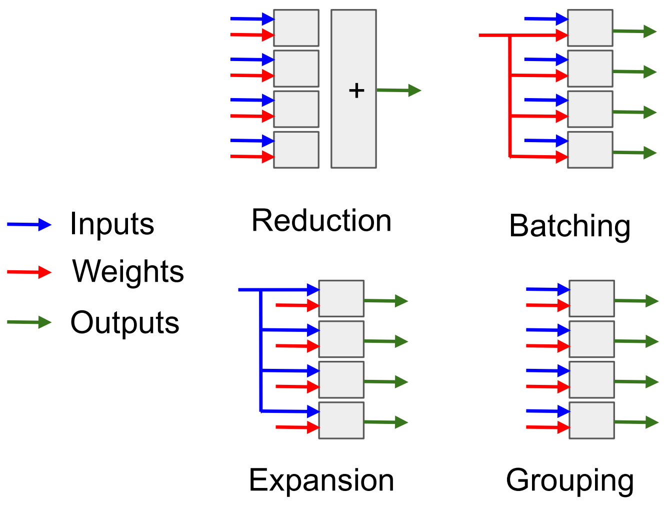


FIGURE 5.3. Data access scheme for reduction, expansion, batching, and grouping

Algorithm 2 Pseudo code for a batched standard 2D convolution layer

```
for $x \leftarrow 0 : X^{\text{stride}} : X^{\text{limit}} - 1$   {This loop variable is of type Batching ($b_0$)}
  for $y \leftarrow 0 : Y^{\text{stride}} : Y^{\text{limit}} - 1$   {This loop variable is of type Batching ($b_1$)}
    for $b \leftarrow 0 : B^{\text{stride}} : B^{\text{limit}} - 1$   {This loop variable is of type Batching ($b_2$)}
      for $k \leftarrow 0 : K^{\text{stride}} : K^{\text{limit}} - 1$   {This loop variable is of type Expansion ($e_0$)}
        for $f_x \leftarrow 0 : F_X^{\text{stride}} : F_X^{\text{limit}} - 1$   {This loop variable is of type Reduction ($r_0$)}
          for $f_y \leftarrow 0 : F_y^{\text{stride}} : F_Y^{\text{limit}} - 1$   {This loop variable is of type Reduction ($r_1$)}
            for $c \leftarrow 0 : C^{\text{stride}} : C^{\text{limit}} - 1$   {This loop variable is of type Reduction ($r_2$)}
              $\mathbf{O}[b, x, y, k] \mathrel{+}= \mathbf{I}[b, x + f_x, y + f_y, c] \times \mathbf{W}[k, f_x, f_y, c]$
```

(3) **Batching (B)**: The loop variables index elements of **I** and **O** to generate multiple values of **O**, reusing **W**, e.g. inference over different inputs.

(4) **Grouping (G)**: The loop replicates computations performed in other loops, e.g. depth-wise and grouped convolutions.

The data accesses for each group are also depicted in Figure 5.3.

Mapping different algorithms to this format may require arbitrary numbers of variables in each group, where this is parameterized by n,m,p, and q respectively for grouping, batching, expansion, and reduction variable groups in Algorithm 1. Let $\mathbb{V} = R \cup E \cup B \cup G$ denote the set of all loop variables, where each $v \in \mathbb{V}$ iterates according to an initial, stride, and limit value of V^{init}, V^{stride}, and V^{limit}. Without loss of generality, we assume the initial values are all zero.

A visualization of algorithms covered by this generalization of convolutional layer in a DNN is presented in Figure 5.2. Inputs I and W are convolved to produce output O. Briefly, the number of input groups and their sizes are described by variables from grouping and batching variable groups. Also, the number of convolution kernels for each input group and their sizes are expressed by variables from expansion and reduction variable groups respectively. This model can represent all the common DNN layers such as standard, depth-wise and point-wise convolutions, as well as many BLAS functions that can be used to implement fully connected layers. For instance, the computation of a batched standard 2D convolutional layer, shown in Algorithm 2, can be mapped to this model using seven nested loops where variables b_0, b_1, b_2, e_0, r_0, r_1, r_2, are input height (X), width (Y) and batch size (B), number of filters (K) and their height (F_X), width (F_Y), and depth (C) respectively. In this example, b_0, b_1 and b_2 are the batching variables; e_0 is an expansion variable; r_0, r_1 and r_2 are reduction variables; and there is no grouping variable. Loop variables of the same group define dimensions of the corresponding access pattern.

5.2.3 Unroll

For each algorithm in the benchmark set, we study the application of the following techniques: 1) tiling via loop splitting, 2) fully/partially unrolling loops, 3) reordering loops, and 4) partitioning the scheduling and compute boundary. This is similar to the approach of Yang et al. [31]. However, we introduce an extra partitioning level which defines the computation to be performed on EBs, accelerator and data scheduler. Our technique restructures the loops so that the EBs can receive input data in parallel from an on-FPGA scheduler using on-chip memory. The on-chip memory receives data from off-chip through a software data scheduler. In this design space, we seek the instances where the EB computation is fully unrolled with unrolling factor M. This parameter defines the parallelism in EBs and is equal to the number of MAC operations in each EB. In our accelerator, all EBs perform the same computation. As detailed later, explicit formulas for utilisation and I/O requirements can be determined, allowing design tradeoffs to be optimized prior to hardware translation.

Algorithm 3 Pseudo code of an unrolling instance for a batched standard 2D convolution layer ($M = 6$, $\hat{F}_X^{\text{unroll}} = 3$, $\hat{B}^{\text{unroll}} = 2$). The blue, red, and yellow area defines the computation partitions assigned to data scheduler, accelerator, and EBs, respectively. The computation on EBs can be performed in parallel (see Figure 5.4A).

```
for $x \leftarrow 0 : X^{\text{stride}} : X^{\text{limit}} - 1$
  for $y \leftarrow 0 : Y^{\text{stride}} : Y^{\text{limit}} - 1$
    for $k \leftarrow 0 : K^{\text{stride}} : K^{\text{limit}} - 1$

      for $f_y \leftarrow 0 : F_Y^{\text{stride}} : F_Y^{\text{limit}} - 1$
        for $c \leftarrow 0 : C^{\text{stride}} : C^{\text{limit}} - 1$
          for $b \leftarrow 0 : B^{\text{stride}} \times B^{\text{unroll}} : B^{\text{limit}} - 1$
            for $f_x \leftarrow 0 : F_X^{\text{stride}} \times F_X^{\text{unroll}} : F_X^{\text{limit}} - 1$

              $\mathbf{O}[b, x, y, k] \mathrel{+}= \mathbf{I}[b, x{+}f_x, y{+}f_y, c] \times \mathbf{W}[k, f_x, f_y, c]$
              $\mathbf{O}[b, x, y, k] \mathrel{+}= \mathbf{I}[b, x{+}f_x{+}1, y{+}f_y, c] \times \mathbf{W}[k, f_x{+}1, f_y, c]$
              $\mathbf{O}[b, x, y, k] \mathrel{+}= \mathbf{I}[b, x{+}f_x{+}2, y{+}f_y, c] \times \mathbf{W}[k, f_x{+}2, f_y, c]$
              $\mathbf{O}[b{+}1, x, y, k] \mathrel{+}= \mathbf{I}[b{+}1, x{+}f_x, y{+}f_y, c] \times \mathbf{W}[k, f_x, f_y, c]$
              $\mathbf{O}[b{+}1, x, y, k] \mathrel{+}= \mathbf{I}[b{+}1, x{+}f_x{+}1, y{+}f_y, c] \times \mathbf{W}[k, f_x{+}1, f_y, c]$
              $\mathbf{O}[b{+}1, x, y, k] \mathrel{+}= \mathbf{I}[b{+}1, x{+}f_x{+}2, y{+}f_y, c] \times \mathbf{W}[k, f_x{+}2, f_y, c]$
```

Algorithm 3 shows one potential unrolling instance of Algorithm 2, where the the variables f_x and b are unrolled. We describe the unrolling by the variables $\hat{F}_X^{\text{unroll}} = 3$ and $\hat{B}^{\text{unroll}} = 2$. To support this unrolling, the computation is partitioned into three shaded areas, assigned to the off-chip data scheduler, scheduler on the FPGA accelerator, and each EB in blue, yellow, and red respectively. The EB computations in red describe a unique, spatially parallel data path that can be translated to a hardware accelerator. The corresponding computation is depicted in Figure 5.4A. Note, due to the unrolling, the loop stride size also increases by the same factor.

However, for $M = 6$, there are many other potential unrolling instances. Figure 5.4 shows five of these for Algorithm 2. Figure 5.4B unrolls c and b, Figure 5.4C unrolls k and c, Figure 5.4D unrolls x and b, and Figure 5.4E unrolls b and f_x. Note that Figure 5.4A and Figure 5.4B share the same MAC arrangement and interconnection scheme; indeed there are other unrolling instances that could also share the same datapath. It follows that the same circuit can be used to support multiple unrolling instances, discussed further in Section 5.2.4. Alternatively, there is only a minor routing overhead to support Figure 5.4C and Figure 5.4E, this can be considered when selecting an optimal subset of desirable projections, discussed further in Section 5.2.5.

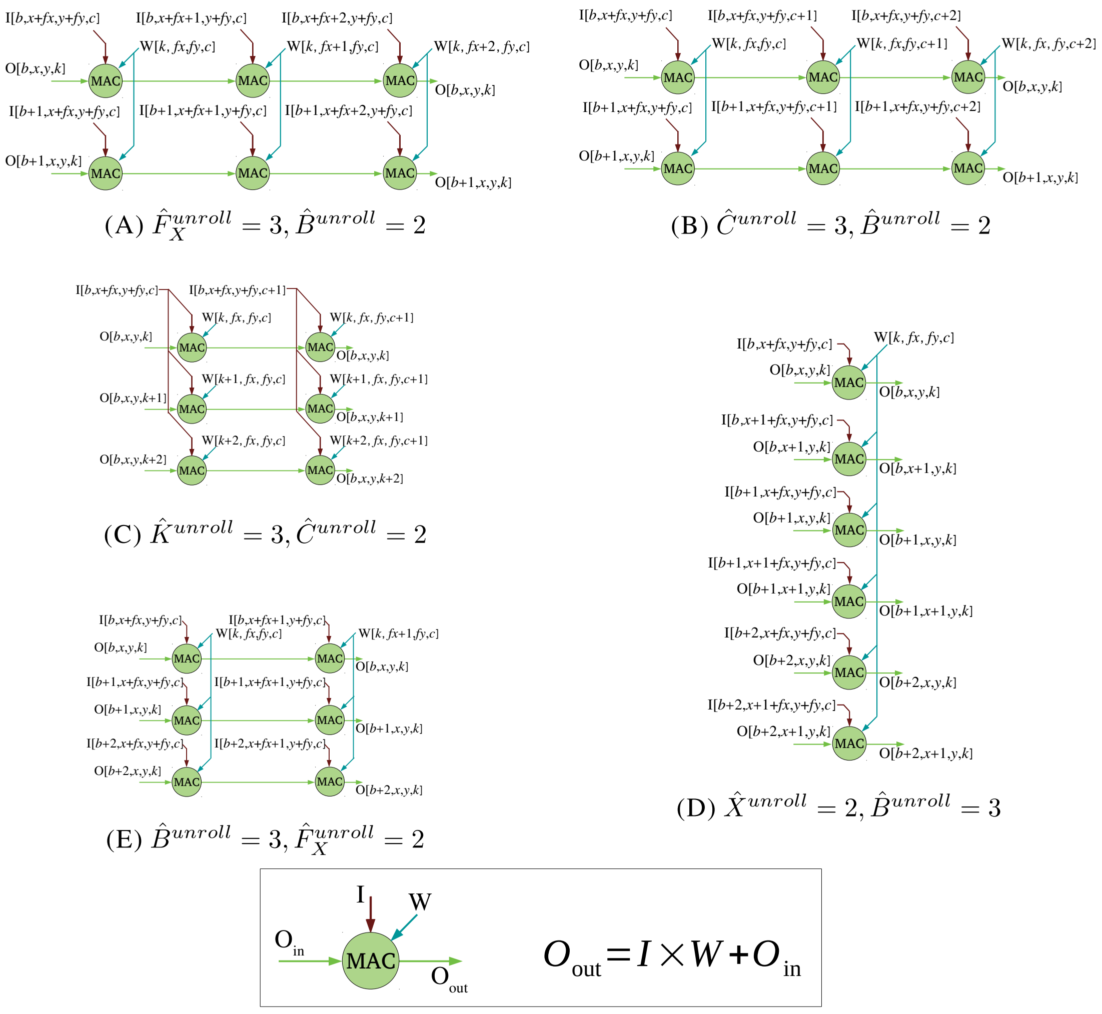
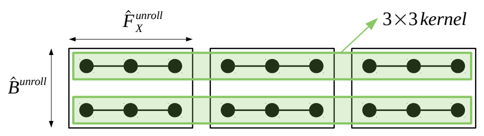

(A) $\hat{F}_X^{unroll} = 3, \hat{B}^{unroll} = 2$
(B) $\hat{C}^{unroll} = 3, \hat{B}^{unroll} = 2$
(C) $\hat{K}^{unroll} = 3, \hat{C}^{unroll} = 2$
(D) $\hat{X}^{unroll} = 2, \hat{B}^{unroll} = 3$
(E) $\hat{B}^{unroll} = 3, \hat{F}_X^{unroll} = 2$
MAC
I
W
O_{in}
O_{out}
$O_{out} = I \times W + O_{in}$

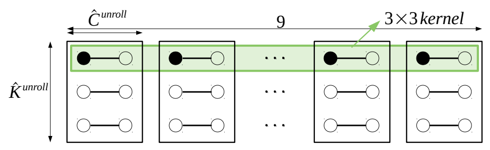
FIGURE 5.4. EB computations for four unrolling instances of Algorithm 2 assuming $M = 6$.

Figure 5.5 and Figure 5.6 illustrate how the utilisation rate of these unrolling instances will differ depending on the target algorithm from the benchmark set for depth-wise and point-wise convolution kernels, respectively.

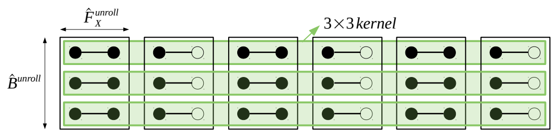
First let us compare how depth-wise convolution with 3×3 kernels can be scheduled based on unrollings of Figure 5.4A, Figure 5.4D and Figure 5.4E; their utilisation is described by Figure 5.5A, Figure 5.5B and Figure 5.5C respectively. A depth-wise convolution multiplies inputs by weights and accumulates over the size of the kernel. Since Figure 5.4A provides parallel access over the variable f_x (also with access size suitable to 3×3 kernels), 100%



(A) Scheduling using the unrolling in Figure 5.4A



(B) Scheduling using the unrolling in Figure 5.4C



(C) Scheduling using the unrolling in Figure 5.4E

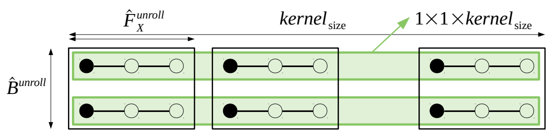
● Used MAC
○ Unused or multiply-by-zero MAC
▨ Used or Unused MAC - benchmark driven

FIGURE 5.5. Scheduling depth-wise convolution kernels on different unrolling instances

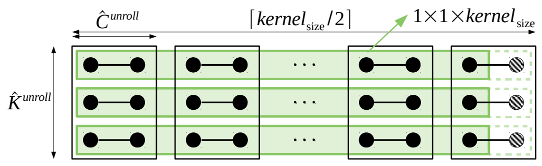
utilisation is achieved; the parallelism over b and shared f_x enables two batches to be computed simultaneously. The other rows of the kernel can be computed with separate EBs again with 100% efficiency. However, the unrolling of Figure 5.4C does not provide parallel access to f_x, meaning accumulation must be done using separate EBs. Furthermore, there is no need to access k in parallel resulting in very low utilisation. A better unrolling with $f_x^{\text{unroll}} = 2$, $B_y^{\text{unroll}} = 3$, as shown in Figure 5.4E achieves 75% utilisation as there are still unused multipliers due to the 3×3 kernel size. Note, for low-batch inputs, unrolling variable b may decrease utilisation.

Now let us compare how point-wise convolution tiles with these same unrollings. Point-wise convolution accumulates over channels (c). This maps well to the unrolling of Figure 5.4C, which will achieve 100% utilisation with an even number of channels; with odd numbers of channels, the final EB will not be fully utilised, as seen in the Figure 5.6B. The unrolling of Figure 5.4A achieves poor utilisation as accumulation over channels must be between EBs. Figure 5.4B performs better as channels can be accumulated over EBs, but this may result in lower efficiency in the final EB, as shown in Figure 5.6C.

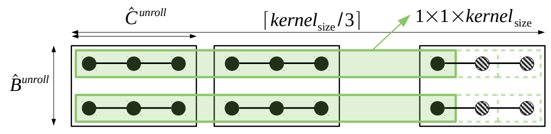
Given these examples, if our benchmark suite were to only consist of depth-wise convolutions, the EBs to support the unrolling of Figure 5.4A would be a good solution. Alternatively, if our benchmark suite were to only consist of point-wise convolutions, we would only be interested



(A) Scheduling using the unrolling in Figure 5.4A


$\hat{C}^{unroll}$
$\lceil kernel_{size}/2\rceil$
$1\times1\times kernel_{size}$
$\hat{K}^{unroll}$

(B) Scheduling using the unrolling in Figure 5.4C


$\hat{C}^{unroll}$
$\lceil kernel_{size}/3\rceil$
$1\times1\times kernel_{size}$
$\hat{B}^{unroll}$

(C) Scheduling using the unrolling in Figure 5.4B

● Used MAC
○ Unused or multiply-by-zero MAC
▨ Used or Unused MAC - benchmark driven

FIGURE 5.6. Scheduling point-wise convolution kernels on different unrolling instances

in developing EBs to support the unrolling of Figure 5.4C. If the benchmark suite were to consist of both point-wise and depth-wise convolutions, we may wish to develop EBs that can support the unrollings of both Figure 5.4A and Figure 5.4C. However, this comes at a hardware cost of additional routing logic. An alternative would be to support unrollings of Figure 5.4A and Figure 5.4B. While this would have slightly worse utilisation, supporting either pair would introduce no additional hardware cost, relying on scheduling to ensure correct data patterns.

The rest of this section describes how we navigate these trade-offs. In the unrolling stage, we simply explore all potential unrollings and evaluate their utilisation. In the projection stage, we identify unique computation and data access patterns among the unrollings, called projections (e.g. Figure 5.4A and Figure 5.4B can be described by the same projection). In the selection stage, we describe how we identify the best projection set to achieve high utilisation with minimal hardware cost.

5.2.4 Projections

In order to uniquely identify the computation of unrolling instances and analyse their performance, we first develop a compact representation that we describe as a *projection*. The foundation for this format is the fact that loop variables belonging to the same variable groups access the IOs similarly for the same computation while in different dimensions. Thus, the computation and IO bandwidth requirements of the EB remain the same.

Specifically, we define unrolling degree for each variable group as the product of all unrollings for the relevant variable group. We then employ a list to distinguish between unique computation projections as follows:

$$< U_{\mathrm{R}}, U_{\mathrm{E}}, U_{\mathrm{B}}, U_{\mathrm{G}} > \tag{5.1}$$

For instance, recall from Algorithm 2 that the loop variables x, y and b are Batching variables, k is an Expansion variable, f_x, f_y and c are Reduction variables. Since f_x is unrolled by a factor of 3 and b is unrolled by a factor of 2, the relevant projection for Figure 5.4A and Figure 5.4B is <3,1,2,1>.

5.2.4.1 Computation Model

The projection encodes the necessary information to realize a unique computation data path. Given a projection, the number of MACs, M is given by (5.2).

$$M = U_{\mathrm{R}} \times U_{\mathrm{E}} \times U_{\mathrm{B}} \times U_{\mathrm{G}} \tag{5.2}$$

5.2.4.2 I/O Constraints

Furthermore, the projection defines the required IO bandwidth for each inputs and outputs from the matrices I, W and O. We consider the bandwidth for each input separately as follows:

$$Bandwidth_{\mathrm{I}} = U_{\mathrm{G}} \times U_{\mathrm{B}} \times U_{\mathrm{R}} \times P_{\mathrm{I}} \tag{5.3}$$

$$Bandwidth_{\mathrm{W}} = U_{\mathrm{G}} \times U_{\mathrm{R}} \times U_{\mathrm{E}} \times P_{\mathrm{W}} \tag{5.4}$$

$$Bandwidth_{\mathrm{O}} = U_{\mathrm{G}} \times U_{\mathrm{B}} \times U_{\mathrm{E}} \times P_{\mathrm{O}} \tag{5.5}$$

, where the P parameters represent the data precision for the corresponding IO. This assumes all data must be streamed into the EB in parallel each cycle, without serialization or double buffering. Delivering the data to this dense compute unit relies on the flexible data movements of the FPGA fabric [31]. The total bandwidth is then described by (5.6). Note, the bandwidth for O is counted twice since it appears in the both inputs and output.

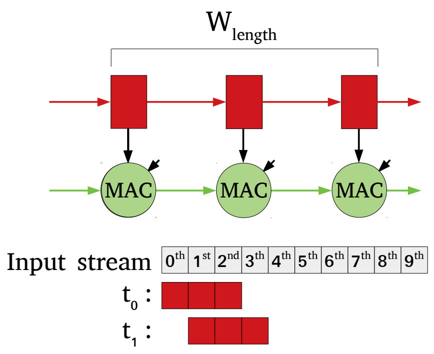
$$Bandwidth = Bandwidth_{\mathrm{I}} + Bandwidth_{\mathrm{W}} + 2 \times Bandwidth_{\mathrm{O}} \tag{5.6}$$

5.2.4.3 Windowing

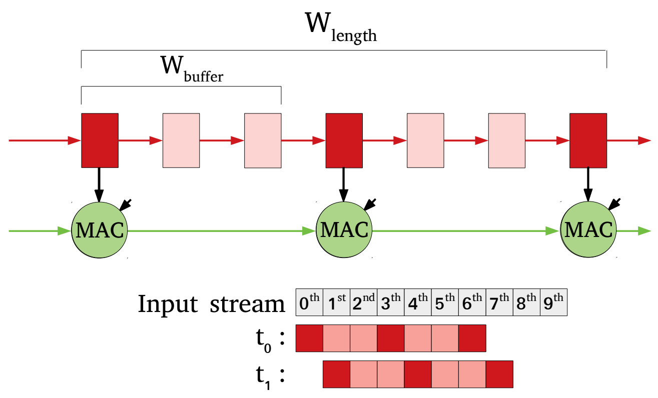
In practice, the required bandwidths limit the unrolling degrees when an EB with a constrained IO budget is desired. However, in the case of neural networks, spatial locality is often exploited to reduce I/O requirements by using windowing to reuse of input data. In our algorithm template, it is feasible only for batching variables where they access an index in combination with a reduction variable, i.e., X and Y access two different dimensions of input I in combination with F_X and F_Y respectively.

In this work, windowing in only one dimension is considered. For higher dimensions, line buffers are required, and we assume that these are implemented outside of the EBs and streamed. We support multiple types of windowing that are common in modern CNNs, as shown in Figure 5.7. Figure 5.7A illustrates the simplest form, where the streamed data is delayed by one cycle via a shift register. In the case of Figure 5.7B, the computation is performed over a larger window, but not all values stored in this window are utilised. Comparing Figure 5.7A and Figure 5.7B, the same input bandwidth is required, but the hardware cost to support the latter is higher. Figure 5.7C shows windowing with a larger stride. In comparison to Figure 5.7A, the hardware requirements are approximately the same, but the routing is slightly different. Figure 5.7D shows a combination of a larger window with strided input, where not all values are utilised. Once again, the hardware cost is similar to Figure 5.7B, but the routing is different.

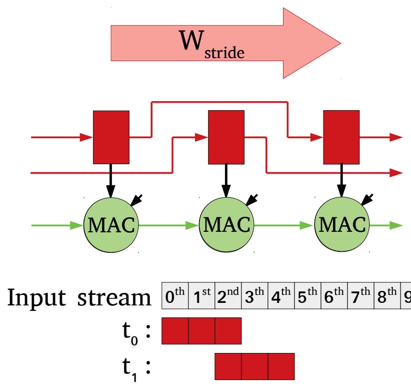
When enumerating all types of unrolling, it is important also to keep track of the relevant windowing. This will ensure that the final EB architecture can support all the desired forms of windowing. Referring to Figure 5.7, a window can be described by 1) window length, W_{length} 2) number of samples utilised per window, W_{samples} 3) window stride, W_{stride}. We define



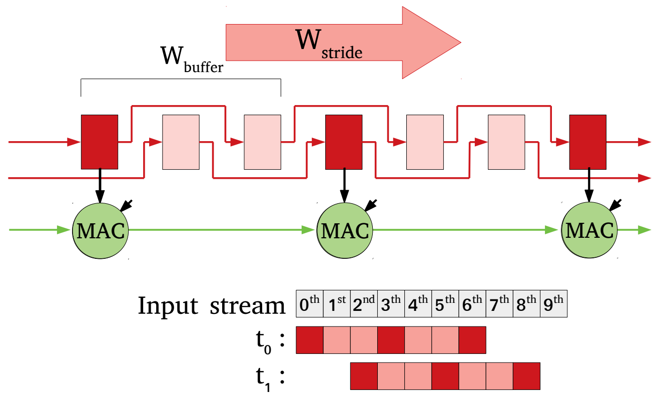
(A) $W_{\text{length}} = 3$, $W_{\text{stride}} = 1$



(B) $W_{\text{length}} = 7$, $W_{\text{stride}} = 1$



(C) $W_{\text{length}} = 3$, $W_{\text{stride}} = 2$



(D) $W_{\text{length}} = 7$, $W_{\text{stride}} = 2$

FIGURE 5.7. Few sample of Generalized windowing regarding Algorithm 2 (In all cases $W_{\text{sample}} = 3$ and only the dark red buffer entries are involved in MAC operations).

W_{buffer} as the number of storage elements required to delay the data between two cascaded MAC operations. This will be used for the implementation of the windowing circuit.

To include the windowing parameters in the projection, we update the initial projection notation in Equation 5.1 to Equation 5.7, where $U_{\text{R}} = U_{\text{R}}^{\text{W}} \times U_{\text{R}}^{\text{N}}$, where superscript $^{\text{W}}$ and $^{\text{N}}$ distinguish the unrolling in windowing and non-windowing part. This includes all potential forms of windowing.

$$< (U_{\text{R}}^{\text{W}}, W_{\text{buffer}}, W_{\text{stride}}), U_{\text{R}}^{\text{N}}, U_{\text{E}}, U_{\text{B}}, U_{\text{G}} > \tag{5.7}$$

For example, in the case of Algorithm 3, if windowing is applied, the relevant projection is <(3,1,1),1,1,2,1>; if windowing is not applied, the relevant projection is <(1,-,-),3,1,2,1>.

Note, Equations 5.2, 5.4, and 5.5 are still valid. However, due to the optimization for input I, the $Bandwidth_{\mathrm{I}}$ is calculated as follows:

$$Bandwidth_{\mathrm{I}} = \begin{cases} U_{\mathrm{G}} \times U_{\mathrm{B}} \times U_{\mathrm{R}}^{\mathrm{N}} \times W_{\mathrm{stride}} \times P_{\mathrm{I}} & \text{if } (W_{\mathrm{stride}} \leq W_{\mathrm{length}}) \\ U_{\mathrm{G}} \times U_{\mathrm{B}} \times U_{\mathrm{R}}^{\mathrm{N}} \times W_{\mathrm{length}} \times P_{\mathrm{I}} & \text{other wise} \end{cases} \tag{5.8}$$

Our generalized windowing approach enlarges the design search space but supports new types of computations such as dilated convolutions [256, 257], recently proposed for temporal convolutional networks [258].

5.2.5 Selection

Each projection eventually maps to a hardware configuration regardless of the EB architecture (Figure 5.1). Due to the design constraints, performance, and implementation costs, supporting all possible projections in an EB is not necessarily desirable or even feasible. Selection is the process of picking a reasonable subset of possible projections, called selected projections. This should exclude projections that do not meet *design constraints* and find the best tradeoff between projection subset, performance, and implementation costs according to the optimization objective.

Design constraints: This includes hard limits on EB interface and implementation cost. The required Bandwidth for I, W, and O is given by the maximum bandwidth of the corresponding inputs and outputs among the selected projections. Note, that the limit for I, W and O are not necessary identical, and will be determined by the available on-chip memory interfaces. Implementation costs, such as area and clock frequency limit can be used to further limit the projection set.

Objective: To take into account both performance and implementation costs while comparing two different projection subsets, we define compute density as the selection objective as follows:

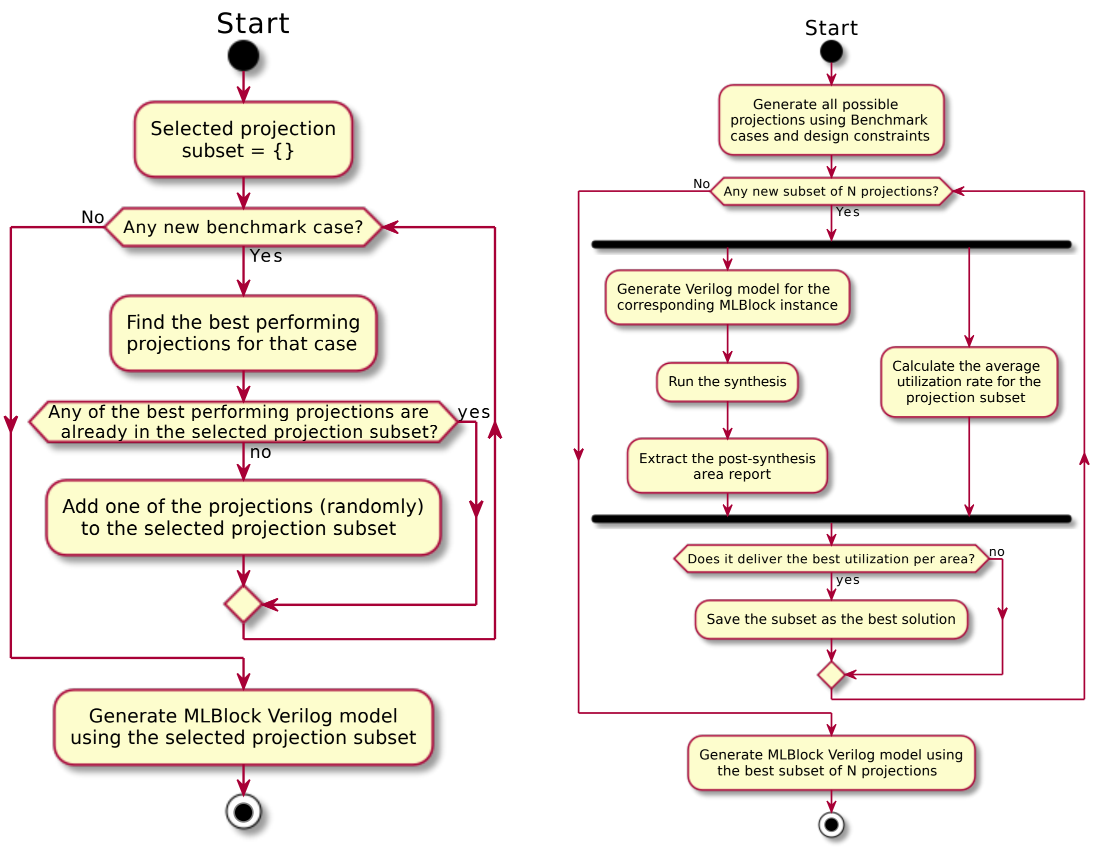
$$\text{compute density} = M \frac{set_{\mathrm{utilization}}}{set_{\mathrm{area}}} \tag{5.9}$$

where set_{area} is the area cost over the projection subset (computed by synthesis) and $set_{\text{utilization}}$ is the average utilization rate over the benchmark. The utilization rate for each algorithm is the maximum utilization rate of that algorithm over each of the projections in the projection subset, *i.e.* that corresponding to the best projection for that algorithm. This measures the highest achievable percentage of time EB MACs are performing useful computation. To find the utilization rate of a projection for implementing an algorithm, we iterate all unrolling instances which map to that projection to find the best tiling using that projection for the given algorithm.

Using these metrics, we can analyse the search space, which involves millions of possible projection subsets. We comment that it is not possible to simply enumerate and synthesize all potential designs. Due to the number of possible subsets, we implemented two different selection strategies 1) a fast greedy and 2) a heuristic, called N-Config.

Greedy: This approach incrementally builds a subset of projections by iterating over benchmark cases one by one. In each iteration, it initially finds the best performing projections for a benchmark case, only considering utilization rate, available MACs and IO requirements. It then checks whether any of the best performing projections is already a member of the selected projection. If not, it adds one of them to the selected subset randomly. This approach optimizes the average utilization rate without considering implementation cost. Figure 5.8A illustrates the process.

N**-Config:** This technique, shown in Figure 5.8B, considers both performance and implementation area. N-Config exhaustively searches for the best solution in the space of T projections considering subsets of N projections at a time. To do so, it generates the Verilog model for the MLBlock instance defined by the selected N configurations, runs synthesis and uses the post synthesis area to optimize for the best compute density (5.9). This requires $\binom{T}{N}$ calls to the synthesis process and as N is increased, the search quickly becomes intractable. Thus only a relatively small N is used. Increasing T also increases the number of possible solutions, which in turn depends on 1) the unrolling factor M, 2) and the computation diversity among the benchmark cases, e.g. various strides and dilations.


Start
Selected projection subset = {}
Any new benchmark case?
No
Yes
Find the best performing projections for that case
Any of the best performing projections are already in the selected projection subset?
yes
no
Add one of the projections (randomly) to the selected projection subset
Generate MLBlock Verilog model using the selected projection subset
Start
Generate all possible projections using Benchmark cases and design constraints
Any new subset of N projections?
No
Yes
Generate Verilog model for the corresponding MLBlock instance
Run the synthesis
Extract the post-synthesis area report
Calculate the average utilization rate for the projection subset
Does it deliver the best utilization per area?
no
yes
Save the subset as the best solution
Generate MLBlock Verilog model using the best subset of N projections

(A) Greedy search (B) N-Config search

FIGURE 5.8. Projection selection techniques

5.2.6 Generation

The generation step in Figure 5.1 takes the selected projections, i.e. a set of projections in the form of Equation 5.7 as inputs, and outputs an EB, where each projection is mapped to a hardware configuration offering the corresponding computation. This requires a parameterized EB architecture based on the projection format. We will suggest an example for such architecture in Section 5.3.

5.3 Architecture

Generation involves providing the flexible routing (via direct connections and multiplexers) required to connect input streams to the MAC unit and implement the selected projections.

The MLBlock architecture is based on a desire to balance two key features: 1) **flexibility** and 2) **modularity**. Our approach has a precision agnostic routing scheme and allows the integration of multi-precision operations using serial multiplication.

Choosing the dataflow for our systolic implementation is an important high-level design decision. We selected a W-stationary approach because: 1) FPGA memory architectures can efficiently implement streaming and windowing (using BRAMs as line-buffers) for $\mathbf{I}$ signals, and 2) $\mathbf{O}$-stationary requires both $\mathbf{I}$ and $\mathbf{W}$ data movements. The chosen approach relaxes the IO requirements of delivering W values every cycle by feeding them in a serial manner and reusing them.

5.3.1 Parameterized MLBlock-M

An MLBlock-M is defined by the number of MAC units (M), and a set of configurations. By changing M, we explore different MLBlocks. A computation projection is a specific routing instance of MAC units. Figure 5.9 shows a parameterized datapath for a given computation projection of the form of Equation 5.7. Parameters $U_{\mathrm{R}}^{\mathrm{W}}$ and $U_{\mathrm{R}}^{\mathrm{N}}$ describe the reduction circuit (dot-product), in which $U_{\mathrm{R}}^{\mathrm{N}}$ groups of $U_{\mathrm{R}}^{\mathrm{W}}$ **I**-**O**-cascaded MAC units are **O**-cascaded. This computation is shown in the purple and red areas. The required windowing values are described by W_{buffer} and W_{stride} which is the same for all MAC units.

The other three parameters, U_{E}, U_{B}, and U_{G}, define the number of copies of the dot-product circuits along three different dimensions. The inputs and outputs to/from each copy are different except for U_{E} and U_{B} directions where $\mathbf{I}$ and $\mathbf{W}$ signals are shared by broadcasting along those directions. In our architecture, MAC units have dedicated $\mathbf{W}$ memories. Thus they do not share the $\mathbf{W}$ values in the U_{B} direction (orange areas).

Our MAC unit is illustrated in Figure 5.10A. It computes $O_{\text{cascade}} = I \times W + O$. I and O signals come from separate multiplexing circuits which are designed to provide different signal sources including other MAC units and block ports per configuration. These allow rearrangement of the MAC units for implementing different projections as configurations. This flexibility is run-time controllable by dedicated mode signals.

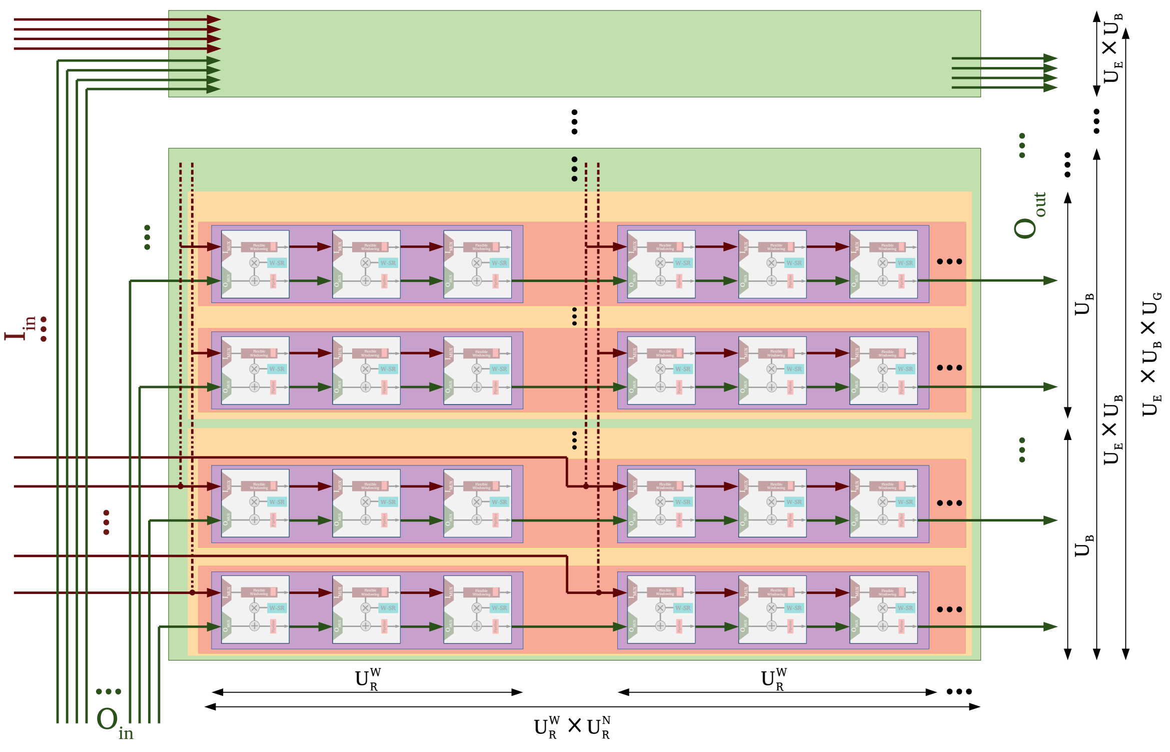
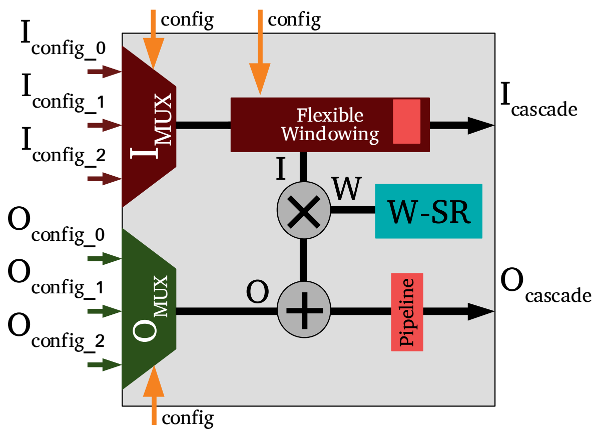


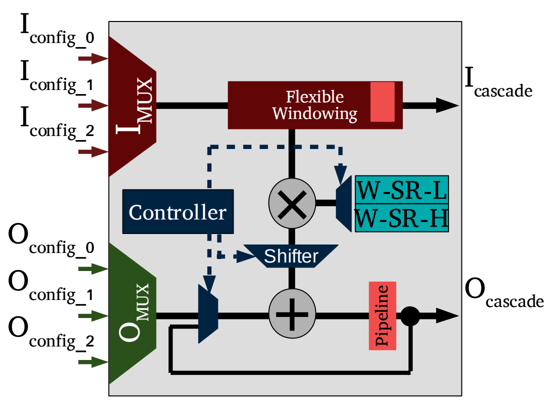
FIGURE 5.9. Parameterized flexible systolic architecture for a configuration expressed by Notation 5.7

The I signal is also registered in the flexible windowing circuit and this circuit can handle windowing for different W_{stride} and W_{buffer}. Its design supports different W_{buffer} and W_{stride} across all desired configurations. An example of this circuit which supports $W_{\text{stride}} = 1$ or 2 and $W_{\text{buffer}} \leq 3$ is depicted in Figure 5.10C. This generates I_{cascade}, which should be connected to the next PE where windowing is required. Registering I prevents excessive propagation delay. To store the $\mathbf{W}$ values, each MAC unit has a shift register. Finally, O_{cascade} carries partial results. To maximize the frequency of a MAC computation, this signal is also registered. The corresponding register for the I_{cascade} signal is augmented with a windowing circuit.

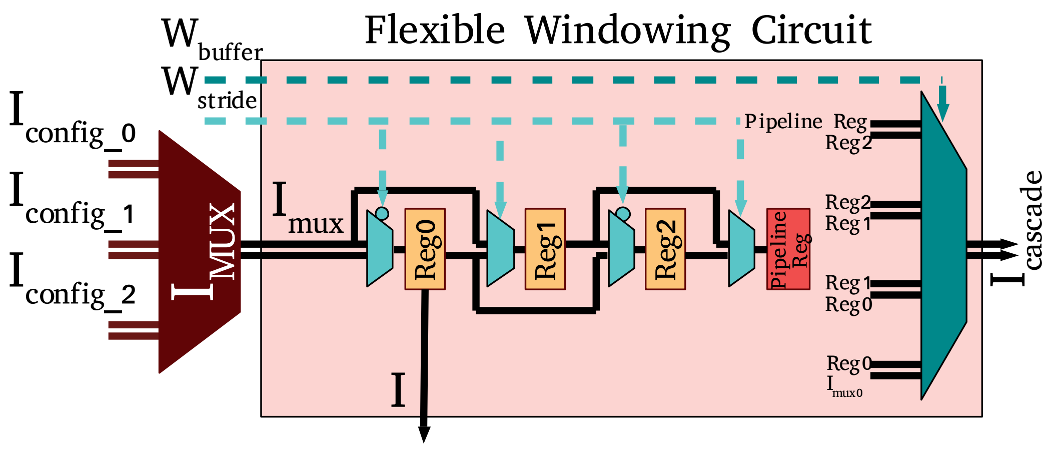
By cascading the O signals of N MAC units, an N-point dot-product computation is implemented. By including the cascade of I signals, the dot product computations with input windowing is achievable. Figure 5.10D shows the cascading arrangement for $N = 3$.



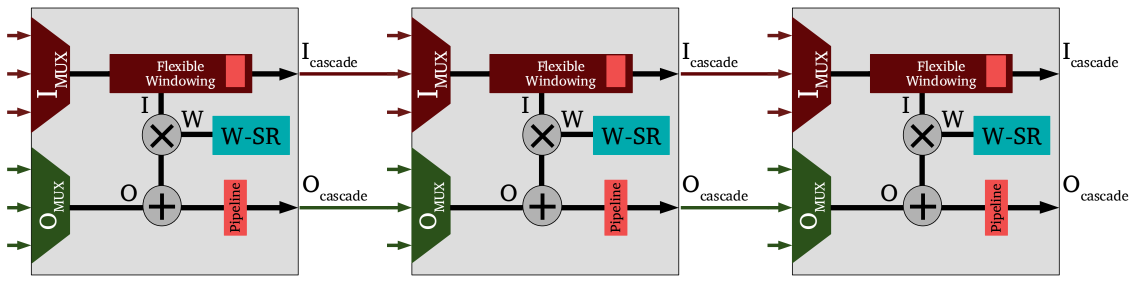
(A) A MAC unit



(B) A serial multiplier-armed MAC unit



(C) Flexible windowing circuit



(D) Cascading MAC units using $I_{cascade}$ and $Res_{cascade}$ signals

FIGURE 5.10. A MAC unit and cascading technique

Figure 5.10B shows the modifications of the MAC unit to support higher-precision computations through serial multiplication. We include a deeper shift register for saving the higher and lower parts of the weight, multiplexing, shifting circuit and an extra multiplexer for reusing the partial result. These circuits require a small sequencer which is compile-time configurable and includes a run-time enable signal.

The operation of MAC units with high precision support is slightly different compared to the baseline version. Considering P_I and P_W as the input precisions, this type of MAC unit can compute $P_I \times P_W$, $P_I \times 2P_W$, $2P_W \times P_W$, or $2P_W \times 2P_W$ MAC operations in 1, 2, 2, or 4 cycles respectively. This means the MLBlock's computation pipeline delivers new output in 1, 2, 2, or 4 cycles. On the other hand, loading the deeper weight shift register also requires $2\times$ increase in bandwidth or the loading latency.

We should note that this architecture does not practically scale with the number of MAC units and configurations due to implementation costs. As the focus of this work is on FPGA EBs with area approximately the same as a DSP48E2, the number of MACs are limited and only a small number of configurations are required. This also raises the necessity for smart configuration selection. Fortunately, many routing resources are shared between different configurations, i.e, wide partial result signals which are always connected to the previous MAC unit. This leads to significant optimisation opportunities for the synthesis process.

5.3.2 MLBlock-M Optimisations

In addition to the introduced general block designing procedures, we applied some optimisations to make the MLBlocks a practical solution as an FPGA EB. In our models, an MLBlock has a configuration signal mode to change between configurations at run-time. This also controls the block output multiplexers.

To place MLBlocks on FPGA architectures, we suggest a DSP-like columnar placement. Similar to DSP dedicated cascading routing between two adjacent MLBlocks for $\mathbf{O}$ signals is possible. This offers dot-product computation expansion using multiple MLBlocks. We removed input port for $\mathbf{O}_{\text{in}}$ while providing internal multiplexer to select between the cascaded $\mathbf{O}_{\text{cascade}}$ signal from the previous MLBlock and constant zero. The main reason behind this is managing IO requirements where, in practice, MLBlocks will be instantiated in the cascade mode and on average, these very wide inputs are not used.

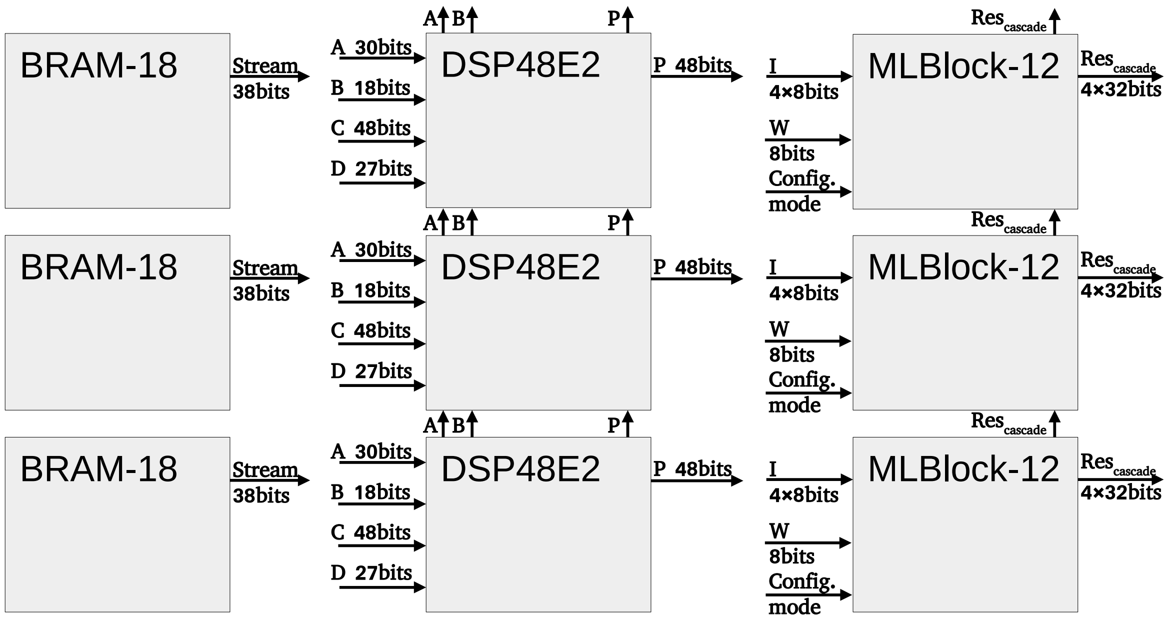


FIGURE 5.11. Columnar placement of BRAMs, DSP48E2, and MLBlock-12

Efficient distributed interface to/from memory blocks, the same for DSP, is imaginable as depicted in Figure 5.11 for MLBlock-12. This enables efficient cascading of the MLBlock computations where partial results are passed to the next MLBlock via dedicated cascade interconnections. It is crucial as using FPGA fabric for high-precision partial result signals pose significant routing challenges.

Initially, we assume a MAC unit comprises 8×8-bit multiplication followed by 32-bit accumulation. Using a serial multiplier configuration, we expand the supported precisions to the three cases of 8×16, 16×8, and 16×16 multiplication followed by 32-bit accumulation.

To define IO constraints, we used the Xilinx FPGA Ultrascale+ architecture as a model. Since MLBlock is a (partial) replacement for DSP blocks, we select constraints similar to a DSP block. Although the memory to DSP block ratio varies among the different part numbers, we assume a $1 : 1$ ratio of DSP48E2 to BRAM18 blocks, and use the same ratio for MLBlock and BRAM18. Recently, Samajdar et al. [45] showed how dedicated cascade interconnections of BRAM18s could be used to implement an efficient 18-bit streaming FIFO which is matched to DSP requirements. In contrast, MLBlocks require higher bandwidth. As tested using Vivado 2018.3, a 36-bit FIFO implementation by a single BRAM18 and custom FIFO controller circuitry using LEs is achievable working with highest routing fabric clock rate. Thus, we limit the bandwidth for $\mathbf{I}$ signals to 36-bits. Similar to [45], to stream in the $\mathbf{W}$ signals, we use URAMs which can provide 8-bit signals per DSP/MLBlock [45] (one 72-bit width URAM per 9 DSPs). Since DSP48E2 have more than 171 input and output data signals, we limit the $\mathbf{O_{out}}$ signal up to 4×32 bits to maintain the same range number of data signal IOs. Note, we do not consider a limit for input O, as it will either enter the MLBlocks using the cascade routing or it would be constant zero.

5.4 Experiments and Results

In this section, we first demonstrate configuration selection. Next, the performance of MLBlock architectures over the Xilinx DSP48E2 for low-precision arithmetic is presented. Then, we integrate the serial multiplication technique with MLBlocks and evaluate performance for both high and low-precision computation. Next, we study MLBlock architectures as an overlay. Afterwards, we study the performance enhancement of replacing DSP48E2 blocks with MLBlock instances for Xilinx Ultrascale+ architecture in details, considering limitation on the number of EBs and the data scheduling overheads. Finally, we compare our MLBlock instances with other solutions for both Xilinx and Intel architectures using well-known CNN benchmarks.

For evaluation, we use the Baidu DeepBench benchmark [259], a collection of computation kernels from three categories of CNNs, RNNs, and general matrix multiplys (GEMMs) selected from real image detection, voice recognition, and text processing applications. Both inference and training tasks are represented. Table 5.1 lists the selected kernels and their loop variable iteration limit and stride. We report the results in three ways: averaged over all benchmark instances, averaged over the instances of each group (GEMM, CNN, RNN), and performance of individual instances (instance names are used individually).

We synthesized MLBlocks using Encounter(R) RTL Compiler RC14.11 targeting STMicro 28nm technology, with 750 MHz as the synthesis clock frequency target. This is the maximum practical frequency for Xilinx DSP48E1 on Virtex-7 using the same technology node [191].

TABLE 5.1. Loop variable iteration limit and stride, (limit, stride), for selected kernels from DeepBench [259]. (GEMMs: [row,column]×[row,column], CNNs: $B[X,Y,C]*\ K[F_X,F_Y,C]$, RNNs: (Hidden layer size, batch size))

	ID	Details	b_0	b_1	b_2	e_0	r_0	r_1	r_2
GEMM	0	[1760,1760]×[1760,128]	(1760,1)	-	-	(128,1)	-	-	(1760,1)
	1	[7860,2560]×[2560,64]	(7860,1)	-	-	(64,1)	-	-	(2560,1)
	2	[2560,2560]×[2560,64]	(2560,1)	-	-	(64,1)	-	-	(2560,1)
	3	[5124,2560]×[2560,9124]	(5124,1)	-	-	(9124,1)	-	-	(2560,1)
	4	[3072,1024]×[1024,128]	(3072,1)	-	-	(128,1)	-	-	(1024,1)
	5	[5124,2048]×[2048,700]	(5124,1)	-	-	(700,1)	-	-	(2048,1)
	6	[35,2048]×[2048,700]	(35,1)	-	-	(700,1)	-	-	(2048,1)
	7	[3072,1024]×[1024,3000]	(3072,1)	-	-	(3000,1)	-	-	(1024,1)
	8	[512,2816]×[2816,6000]	(512,1)	-	-	(6000,1)	-	-	(2816,1)
	9	[7680,2560]×[2560,1]	(7680,1)	-	-	(1,1)	-	-	(2560,1)
	10	[7680,2560]×[2560,2]	(7680,1)	-	-	(2,1)	-	-	(2560,1)
	11	[7680,2560]×[2560,1500]	(7680,1)	-	-	(1500,1)	-	-	(2560,1)
	12	[10752,3584]×[3584,1]	(10752,1)	-	-	(1,1)	-	-	(3584,1)
	13	[5124,2048]×[2048,700]	(5124,1)	-	-	(700,1)	-	-	(2048,1)
	14	[35,2048]×[2048,700]	(35,1)	-	-	(700,1)	-	-	(2048,1)
	15	[3072,1024]×[1024,1500]	(3072,1)	-	-	(1500,1)	-	-	(1024,1)
	16	[7680,2560]×[2560,1]	(7680,1)	-	-	(1,1)	-	-	(2560,1)
	17	[7680,2560]×[2560,1500]	(7680,1)	-	-	(1500,1)	-	-	(2560,1)
	18	[7680,2560]×[2560,1]	(7680,1)	-	-	(1,1)	-	-	(2560,1)
CNN	0	32[700,161,1]∗32[5,20,1]	(700,2)	(161,2)	(32,1)	(32,1)	(5,1)	(20,1)	(1,1)
	1	8[54,54,64]∗64[3,3,64]	(54,1)	(54,1)	(8,1)	(64,1)	(3,1)	(3,1)	(64,1)
	2	16[224,224,3]∗64[3,3,3]	(224,1)	(224,1)	(16,1)	(64,1)	(3,1)	(3,1)	(3,1)
	3	16[7,7,512]∗512[3,3,512]	(7,1)	(7,1)	(16,1)	(512,1)	(3,1)	(3,1)	(512,1)
	4	16[28,28,192]∗32[5,5,192]	(28,1)	(28,1)	(16,1)	(32,1)	(5,1)	(5,1)	(192,1)
	5	4[341,79,32]∗32[5,10,32]	(341,2)	(79,2)	(4,1)	(32,1)	(5,1)	(10,1)	(32,1)
	6	1[224,224,3]∗64[7,7,3]	(224,2)	(224,2)	(1,1)	(64,1)	(7,1)	(7,1)	(3,1)
	7	1[56,56,256]∗128[1,1,256]	(56,2)	(56,2)	(1,1)	(128,1)	(1,1)	(1,1)	(256,1)
	8	2[7,7,512]∗2048[1,1,512]	(7,1)	(7,1)	(2,1)	(2048,1)	(1,1)	(1,1)	(512,1)
	9	1[112,112,64]∗64[1,1,64]	(112,1)	(112,1)	(1,1)	(64,1)	(1,1)	(1,1)	(64,1)
	10	1[56,56,256]∗128[1,1,256]	(56,2)	(56,2)	(1,1)	(128,1)	(1,1)	(1,1)	(256,1)
	11	1[7,7,512]∗2048[1,1,512]	(7,1)	(7,1)	(1,1)	(2048,1)	(1,1)	(1,1)	(512,1)
RNN	0	RNN (1760, 16)	(1760,1)	-	(16,1)	(1760,1)	-	-	(3520,1)
	1	RNN (2560, 32)	(2560,1)	-	(32,1)	(2560,1)	-	-	(5120,1)
	2	LSTM (1024, 128)	(4096,1)	-	(128,1)	(1024,1)	-	-	(2048,1)
	3	GRU (2816, 32)	(8448,1)	-	(32,1)	(2816,1)	-	-	(5632,1)
	4	LSTM (1536, 4)	(6144,1)	-	(4,1)	(1536,1)	-	-	(3072,1)
	5	LSTM (256, 4)	(1024,1)	-	(4,1)	(256,1)	-	-	(512,1)
	6	GRU (2816, 1)	(8448,1)	-	(1,1)	(2816,1)	-	-	(5632,1)
	7	GRU (2560, 2)	(7680,1)	-	(2,1)	(2560,1)	-	-	(5120,1)

Please note, we use post-synthesis results to report area in um^2 and power in mW directly according to the reports. The DSP48E2 Verilog model was an open source version available through [1].

5.4.1 Configuration Selection Methods

TABLE 5.2. Selection methods for MLBlock-12 (results based on average over all benchmark cases)

Method	Utilization	Area[†]	Obj[‡]	# Synth.	time
Greedy	88.241	6093	1	1	1–2 mins
1-Config	72.000	5243	0.94	28	3–4 mins
2-Config	86.019	5245	1.13	378	1–2 hours
3-Config	88.192	5634	1.08	3276	$\approx$ a day
4-Config	88.241[§]	-	-	20475	$\approx$ a week
5-Config	88.241[§]	-	-	98280	$\approx$ a month

[†]um^2, [‡]normalized objective: $\frac{Utilization}{Area}$

Table 5.2 summarizes results for MLBlock-12 using different design exploration techniques. The greedy approach chooses four configurations leading to higher area while delivering the highest utilization rate. In contrast, the N-Config approach maximises compute density ($\frac{\text{Utilization}}{\text{Area}}$) as shown in column Obj. The N-Config approach requires synthesis to be executed within its search loop and is therefore time consuming. However, calculating utilization rate is quick. We comment that as reported in Table 5.2 the utilization rate plateaus for $N \geq 4$, but at the same time, in general supporting more complex configurations results in a higher implementation cost, so little improvement is expected. The search space of 2/3-Config is also shown in Figure 5.12. The wide range of utilization rate is due to a mismatch in supported stride or unrolling of the selected projection and benchmarks. The final compute density (utilization/area) can vary by up to two orders of magnitude, highlighting the need for automated design space exploration. Although 2-Config delivers the best performing MLBlock for our benchmark, for the remaining experiments we used the Greedy method for its speed. The resource utilization range is also larger for Greedy since supporting more configurations requires more routing and buffering resources.

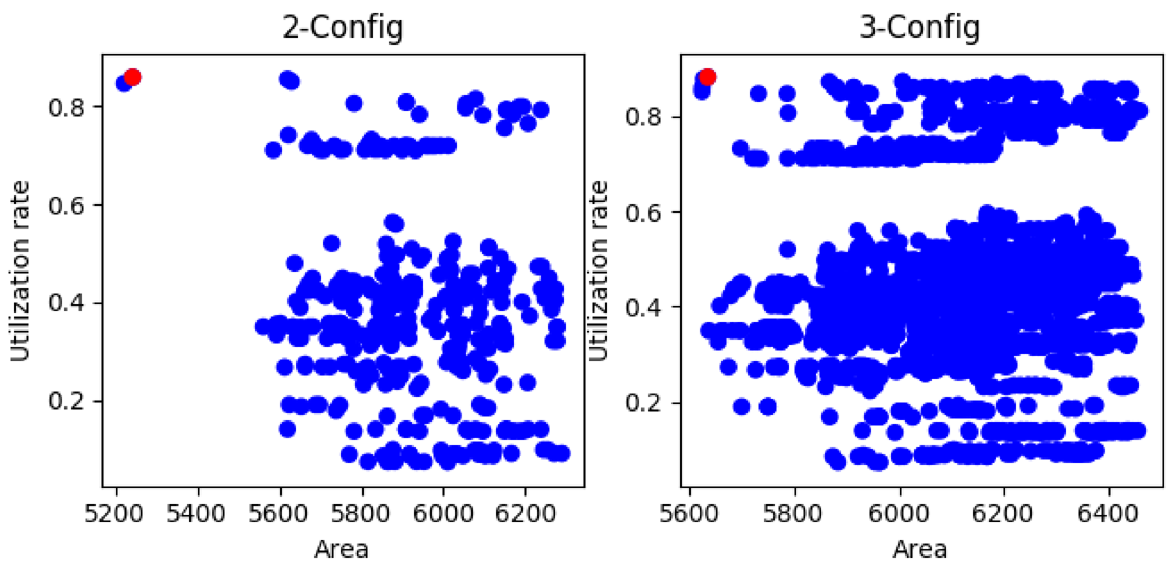


FIGURE 5.12. Search space for MLBlock-12 using 2 and 3-Config technique (results based on average over all benchmark cases). The most efficient architecture is pointed by red (Area: um^2).

5.4.2 MLBlocks vs. DSP48E2

Using the same IO constraints as DSP48E2, we generate MLBlocks for different numbers of MAC units and results are summarized in Table 5.3. We normalized the area and power values based on the results for our DSP48E2 model, where the absolute numbers are 7208.87 um^2 and 19.516 mW, respectively. From the table, it can be seen that in an MLBlock-M, area and power scale linearly with M. All 8×8 precision configurations have significantly lower power and area than the DSP48E2. It is also important to note that the MLBlock-M can perform $M/2$ times more MACs per cycle at 8-bit precision.

Compute density (or performance per area) can be calculated as $\frac{\text{Utilization}\times\text{MACs}}{\text{Area}}$. Figure 5.13 shows this metric for the different benchmark classes. The performance of MLBlocks can be seen to be around $6\times$ higher than DSP48E2 for 8-bit multiply and 32-bit accumulate operations. This is because the MLBlock architecture can provide more MACs without increasing routing. Also, DSP48E2 includes some unneeded circuits such as a 48-bit comparator, wide logic functions, pre-adder, and wider accumulator. It is notable that MLBlock-12 is also 15% smaller than a DSP48E2. To be fair, these extra features are required for other

TABLE 5.3. Post-synthesis results for different EB architectures

EB name	Precisions	Utilization	Area*	Power*
DSP48E2	27×18 or two 8×8	≈ 1	1	1
MLBlock-12	8×8	88%	0.85	0.98
MLBlock-9	8×8	86%	0.66	0.81
MLBlock-8	8×8	92%	0.56	0.65
MLBlock-6	8×8	93%	0.46	0.54
MLBlock-12	$(16/8) \times (16/8)$	88%	1.44	1.81
MLBlock-9	$(16/8) \times (16/8)$	86%	1.20	1.57
MLBlock-8	$(16/8) \times (16/8)$	92%	0.96	1.20
MLBlock-6	$(16/8) \times (16/8)$	93%	0.80	1.02

*normalized

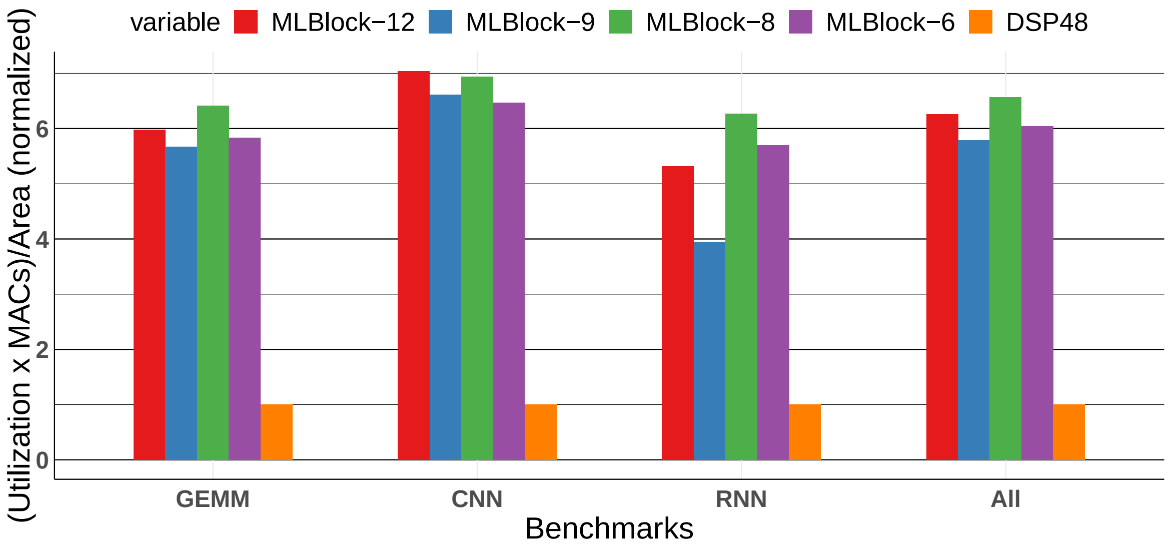


FIGURE 5.13. Compute density of various MLBlocks for low precision computations (results based on average for each benchmark category and over all benchmark cases)

applications not considered in this study and replacing a percentage of DSP48E2s with MLBlocks could achieve a compromise.

Figure 5.13, also shows how the number of MAC units affects the efficiency. MLBlock-12 is a particularly good choice as 12 it is divisible by 2, 3, 4 and 6, making the mapping of different loop counts more efficient. In general, choosing an M with a diverse set of factors leads to a higher utilization rate. Thus, MLBlock-9 shows a lower performance in general and is particularly bad for the LSTM cases; its excellent performance in CNN benchmarks is due to the fact that 3 is a common window size.

The other aspect of MLBlocks is IO efficiency. Physically this corresponds to providing local connections and run-time configuration flexibility inside the block, instead of relying on FPGA fabric and LE-based multiplexing. For instance, MLBlock-12 requires 24-bit data signals per MACs while this number is 93 for the DSP48E2. In both cases, the computations themselves are still dot-product computations. The key to MLBlock's efficiency lies in avoiding global routing by providing internal broadcasting or using windowing.

5.4.3 MLBlocks with High-precision Support

Adding some additional circuitry to the 8×8 bit multiplier can enable its use as a serial-parallel multiplier to achieve higher precision. We now describe a high precision MLBlock which supports $8/16 \times 8/16 + 32$-bit MAC operations. The effect of this change on area, utilization and power is shown in the bottom half of Table 5.3. As expected, the area increases and performance of low-precision arithmetic is reduced to around $3.5\times$ since multiplication becomes a multi-cycle operation. Figure 5.14 shows the compute density of different high precision MLBlocks. While adding high precision support necessarily decreases the performance advantage and increases area, the overall compute density is still a significant improvement over DSP482E. Indeed, MLBlock-8 is 4% smaller than DSP48E2 while having approximately $3\times$ higher compute density for 8-bit operations and $2\times$ higher compute density for 16-bit operations.

5.4.4 MLBlocks as Overlays

MLBlock models can also be used as soft IP cores which utilize FPGA logic elements. Although these models are not optimized for look-up table based implementations, studying them as overlays brings insight into the MLBlock architecture and its variations.

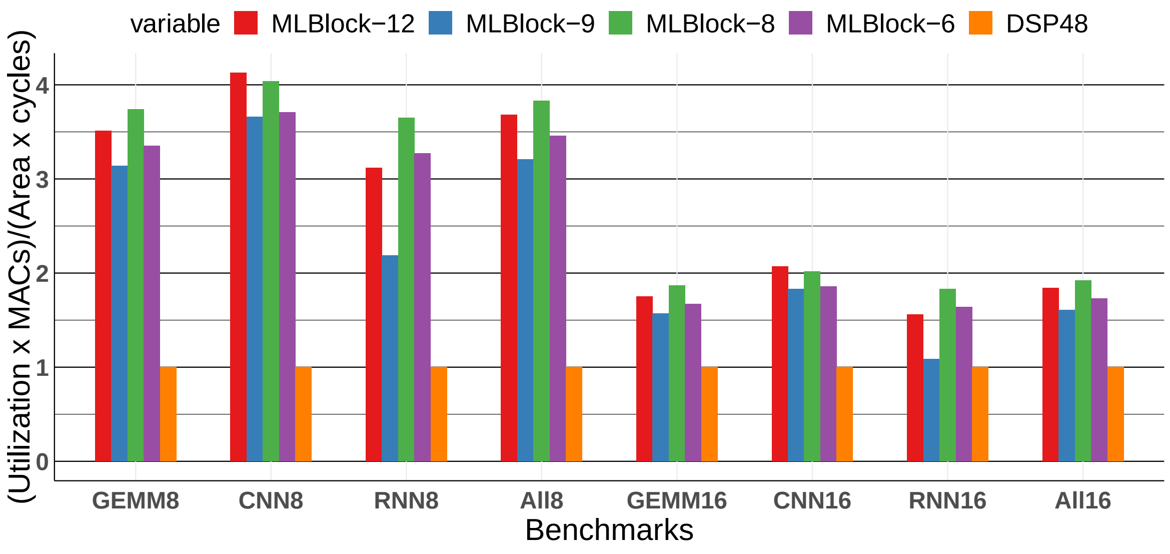


FIGURE 5.14. Compute density of various MLBlocks using serial-parallel multipliers for high/low precision computations (results based on average for each benchmark category and over all benchmark cases)

TABLE 5.4. Post implementation resource and timing results for different EB architectures as overlays

EB name	Precision	CLBs	Registers	f_{Max} (MHz)	Performance/CLB
DSP48E2	27×18	338	564	154	0.91
MLBlock-12	8×8	324	649	221	7.20
MLBlock-9	8×8	243	650	251	7.99
MLBlock-8	8×8	187	553	222	**8.74**
MLBlock-6	8×8	160	386	223	7.78
MLBlock-12	$(16/8) \times (16/8)$	605	1054	223	3.89
MLBlock-9	$(16/8) \times (16/8)$	419	941	211	3.90
MLBlock-8	$(16/8) \times (16/8)$	383	674	225	**4.32**
MLBlock-6	$(16/8) \times (16/8)$	291	606	197	3.78
8×8 MAC (32-bit acc.)	8×8	19	80	378	19.89

Table 5.4 summarizes the required FPGA resources for implementing MLBlock-12, 9, 8, 6 with and without high precision support. We report post-implementation results targeting the Virtex UltraScale+ architecture (part number xcvu5p-flva2104-1-i).

Considering the resource utilization of a single-cycle signed 8×8 multiply and 32-bit accumulation unit, we calculate the resource overhead due to the additional routing for MLBlock-M (without high precision support) by $\frac{\text{CLB}_{\text{MLBlock-}M} - M \times \text{CLB}_{8\times 8\text{MAC}}}{M \times \text{CLB}_{8\times 8\text{MAC}}}$, where $\text{CLB}_{\text{MLBlock-}M}$ and $\text{CLB}_{8\times 8\text{MAC}}$ are the CLB costs for MLBlock-M and the single-cycle signed 8×8 MAC unit, respectively. Using that, MLBlock models (without high precision support) introduce 37% resource overhead on average. This means MLBlocks as EBs encapsulate significant run-time configurable routing resources, it could subsequently reduce routing pressure in deployments where run-time reconfiguration is required.

In addition, using the utilization rates from Table 5.3, we calculate performance per CLB as $\frac{\text{MACs} \times \text{Utilization} \times F_{Max}}{\text{CLB}}$. MLBlock-8 delivers the highest score, due to 1) high utilization rate over the benchmarks, and 2) an efficient configuration set. This is the same conclusion as for the standard cell synthesis results.

5.4.5 Performance Analysis Considering Data Scheduling

Increasing compute density does not necessarily translate to higher performance. Utilizing the added compute power relies on efficient data scheduling and becomes crucial when the number of EBs are limited and data movement overheads are significant. In MLBlocks, there is a narrow port for streaming in one of the inputs, which means an MLBlock-M requires M cycles to be (re)initialized. The pipeline registers within MAC units causes a number of cycles of delay before the first output appears. This latency may vary between the configurations of an MLBlock.

To explore the effect of data scheduling, we developed an open source Verilog generator [253] for circuits expressed as nested loops according to Algorithm 1. The generator instantiates the data path (EB instances, memories interconnections and memories) together with a corresponding state machine to implement the algorithm. The design space exploration is constrained by selected data flow, the number of EBs, memory components, and EB interfaces. We select a weight stationary data flow for this experiment. Assuming Ultrascale+ architecture, BRAM and URAM blocks are used to define the memory components. The required clock

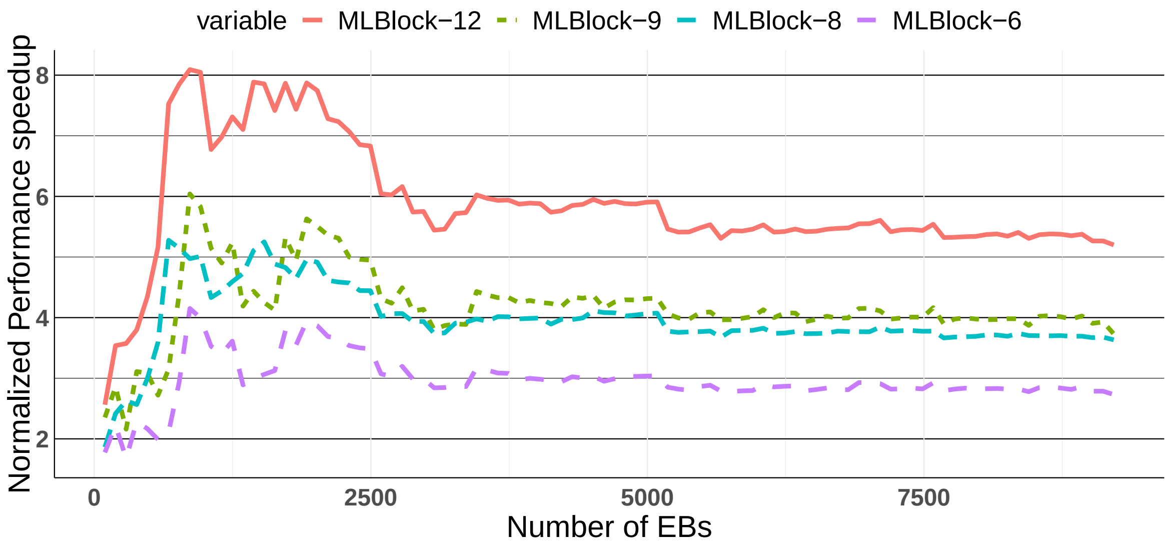


FIGURE 5.15. Performance speedup of various MLBlock instances comparing to DSP48E2 for various budget for number of available EBs with narrowing the EB column's width

cycles is the optimization objective as a proxy for performance. This considers both 1) (re)initialization, and 2) the computation pipeline latencies for MLBlocks. We modelled an MLBlock by describing each compute projection as a separate logical EB. The best performing projection represents the MLBlock performance for the given input algorithm.

Number of EBs vs performance: Generally, weight stationary data flow works better when compute units have sufficient memory to buffer the fetched kernel parameters and avoid unnecessary data transfers. Increasing the number of EBs or their embedded memories leads to scheduling schemes with higher data reuse. Figure 5.15 shows the normalized speedup for different MLBlock instances comparing to the implementations with the same number of the DSP48E2 blocks. We swept the EB budget through the multiples of 96, where 96 is the common number of DSPs in a DSP column for this FPGA family. In this experiment, we again used the DeepBench benchmarks.

We first replace each DSP48E2 block with an MLBlock-M without changing the EB column size. This is possible as MLBlock instances are all smaller than the DSP48E2. Hence, with the same area footprint, MLBlock-12, 9, 8, and 6, are 6, 4.5, 4, 3$\times$ denser than the DSP48E2 for low-precision arithmetic.

Generally, the performance gain is expected to be the same as the compute density for an MLBlock-M architecture. However, for small numbers of EBs/DSP48E2s, the MLBlock-M architecture struggles with the high number of costly data transfers. Although MLBlock-M offers $M\times$ larger memories suitable for reusing a larger portion of the kernels, it also requires $M\times$ longer (re)initialization phases compared to their DSP48E2 counterpart. This trade-off restricts the performance speedup of MLBlocks to about 2-$3\times$.

As the number of EBs/DSP48E2s are increased, the performance speedup for both MLBlocks and DSP48E2 columns improve. However, due to the higher memory and compute power density, the MLBlock architecture avoids unnecessary data movements, leading to higher performance. In fact, the growth surpasses the performance speedup expectations because of benchmark instances where the MLBlock memories are sufficient to save and reuse the kernel parameters, while DSP48E2 must incur the overhead of extra data transfers. Eventually both architectures reach the point that weights can be stored on the compute unit memories without excessive data movements and overheads. As can be seen in Figure 5.15, the performance speedups converge to the compute density ratios.

Figure 5.16 displays the speedups for each benchmark case, assuming the fixed budget of 4800 EBs. As expected, the average speedups should be consistent with the abovementioned computation density rates. Somewhat surprisingly, it varies among the benchmark groups and their members. For CNNs, the speedups are consistently close to the computation density ratios. However, for GEMMs, this is not always the case as some kernels have very low reuse factors or extremely narrow dimensions, e.g. GEMM-9, 10, 12, 16, 18. In contrast, due to the very large matrix multiplications in RNNs, there are some cases (e.g. RNN-1, 3, 6, 7) in which MLBlocks surpass the nominal speedup rates. This is because of their larger memories to save and reuse weights, which leads to a higher utilization rate while having more MAC units.

If narrowing the EB column's width is allowed, a small reduction in the FPGA footprint is expected. As given in reference [260], the DSP blocks contribute to only 5% of the total FPGA fabric. Using the estimated area ratios, the total FPGA chip area reduction is in the

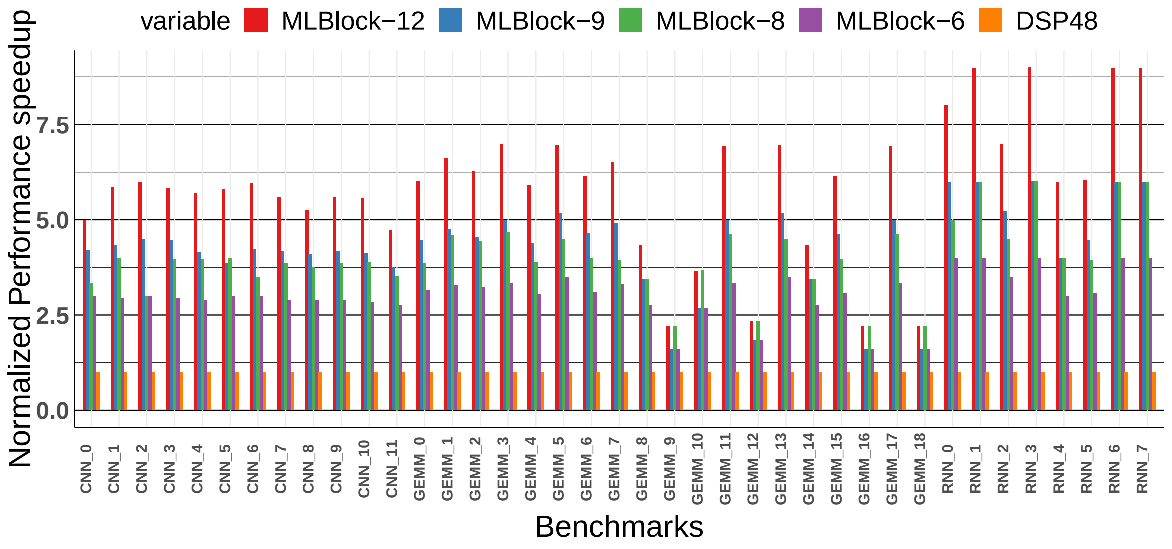


FIGURE 5.16. Performance speedup of various MLBlock instances comparing to DSP48E2 for fixed budget of 4800 EBs

range of 1-2.7%, which slightly enhance the results on Figure 5.15 without affecting the trends.

It may also be possible to use the saved area to instantiate more MLBlocks in a column. Figure 5.17 shows the performance comparison where each DSP48E2 column is replaced by an MLBlock column with 122, 145, 171, or 208 instances of MLBlock-12,-9,-8, or -6 respectively. As expected, similar to Figure 5.15, initially, the number of EBs limits the performance to the range of 2.5-5×. Since each column includes more EBs; increasing the number of columns enhances the performance at higher rates. Thus, the same trend with higher performance gains is achieved. Finally, the performance speedups converge to the compute density rates, which are shown in Figure 5.13. Note, in this setting, each MLBlock column may require more than a column of BRAM support as the 1:1 BRAM-MLBlock ratio is not preserved. Consequently, denser memory units are required to achieve such gains.

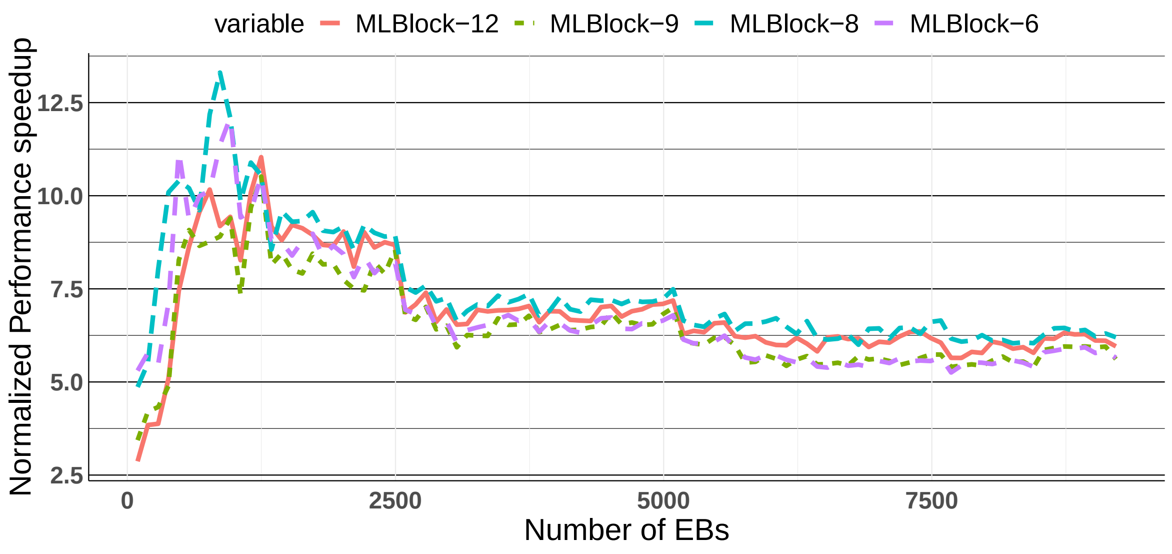


FIGURE 5.17. Performance speedup of various MLBlock instances comparing to DSP48E2 for various budget for number of available EBs without narrowing the EB column's width

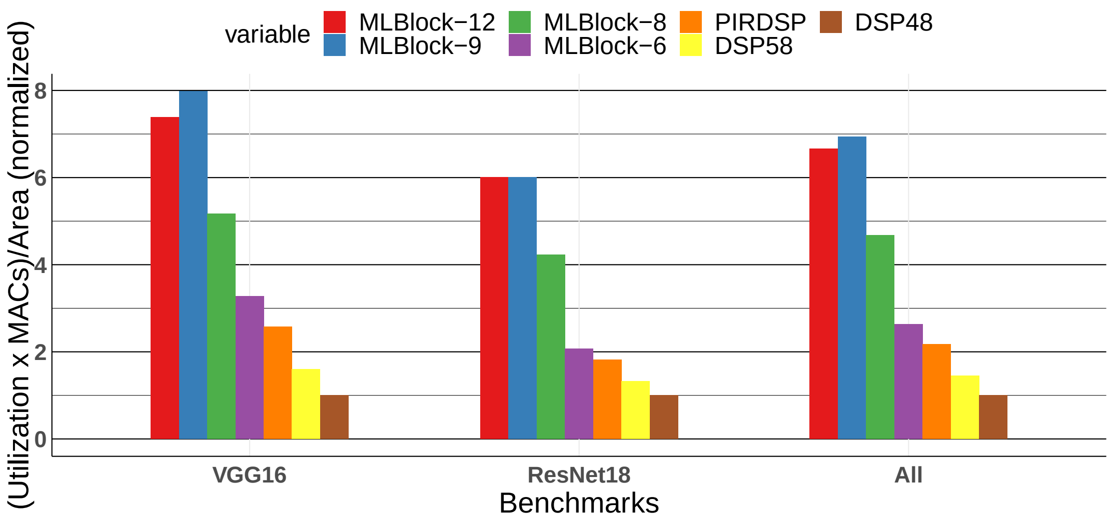
5.4.6 Comparison with Commercial Devices

5.4.6.1 Xilinx Architecture:

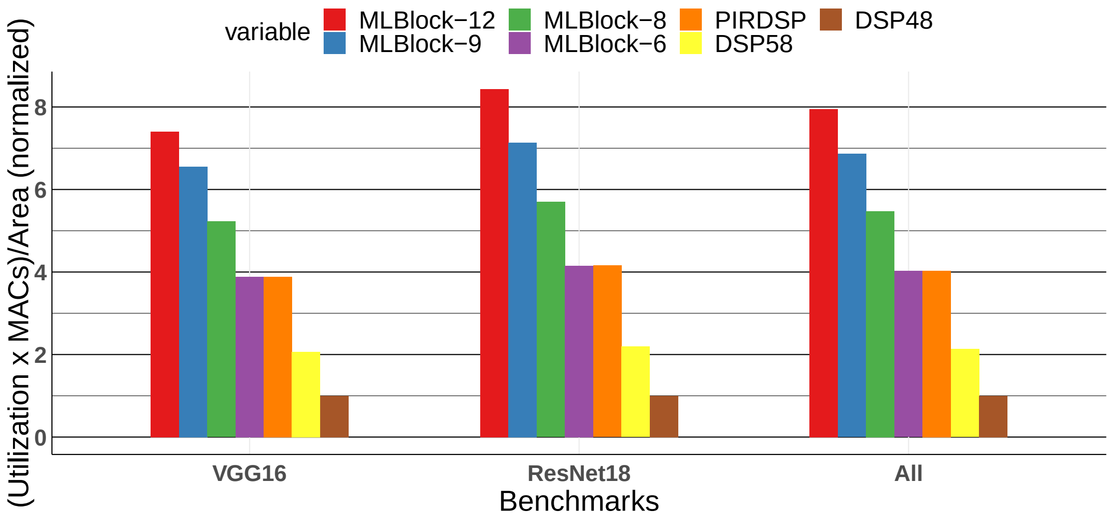
To demonstrate the effectiveness of the suggested design methodology, we first compare MLBlocks with other expert-designed alternatives for DSP48E2 from industry (Xilinx DSP58) and academia (PIR-DSP[1]). We use a new benchmark comprising computation kernels of convolution and fully-connected layers for two well known CNNs (VGG16 [90] and ResNet18 [91]) (batch size 2), while keeping the same MLBlocks as for the DeepBench [259] kernels.

The plots in Figure 5.18 compare the average speedup of VGG16 and ResNet18 kernels using MLBlocks, PIR-DSP, DSP58, and DSP48E2, assuming a fixed budget of 360 (Figure 5.18A) and 4272 (Figure 5.18B) EBs. These represent small embedded (Ultra 96v2 [261]) and large, high-end (ZCU111 [262]) FPGA development board specifications respectively.

As expected, similar to Figure 5.15, MLBlock modes achieve higher speedups for low precision arithmetic in both scenarios. In the embedded scenario, as the benchmark set is heavily weighted with standard 2D convolution layers having 3×3 kernels, MLBlock-9 is a



(A) Considering Xilinx Zynq UltraScale+ MPSoC ZU3EG resources as an embedded FPGA, used in U96v2 [261]



(B) Considering Xilinx Zynq UltraScale+ XCZU28DR resources as a high-end FPGA, used in RFSoC development kit, ZCU111 [262]

Figure 5.18. Performance comparison between MLBlocks and expert-designed replacement for Xilinx DSP48E2 from both academia and industry considering both embedded and high-end FPGAs

better fit for the computations and delivers the best performance. In contrast, by increasing the size of EBs (high-end scenario), the performance speedups converge to the compute density ratios, and MLBlock-12 achieves the best performance.

PIR-DSP [1] and MLBlock-6 both can do six low-precision (8×8-bit) MACs. Interestingly, one of the MLBlock-6 configurations exactly matches the low precision mode in PIR-DSP (two parallel 3-element dot-products). However, other configurations of MLBlock-6 result in more efficient computation tiling comparing to PIR-DSP. Furthermore, PIR-DSP's commitment to backward compatibility meant retaining extra circuits such as a large pre-adder, comparator, and wide logic circuits. Consequently, it has about 18% larger area than DSP48E2. In our comparison, we dismiss this area overhead. This gap is larger when the number of EBs are limited, as extra configurations help to boost the performance. By increasing the number of EBs, the speedups converge to compute density rates which are the same for both architectures.

DSP58, with three low-precision MAC operations, shows performance between PIR-DSP and DSP48E2 with 6 and 2 MACs, respectively. Similar to PIR-DSP, it has an area overhead due to backward compatibility with the DSP48E2. Since there is no published information available, we assume that similar DSP/CLB/BRAM ratios for both DSP48E2 and DSP58, and also assume the same area and working frequency for these two DSPs.

5.4.6.2 Intel Architecture:

The suggested design methodology can be applied to other FPGA architectures, such as Intel FPGAs. Although it is better to recreate MLBlock instances according to the area and IO constraints of the new architecture, for consistency we use our previously generated instances. We gathered implementation results for MLBlocks, Stratix-10 DSP, an alternative DSP proposed in [209], as well as Tensor Slice [219] all in Table 5.5. Since the other DSPs support high precision, they are compared with MLBlocks with high-precision support.

The last column of this table presents our comparison metric calculated as $\frac{\text{\# of MACs}}{\text{Area}\times\text{Delay}}$ for both high and low-precision arithmetic, which considers both performance and implementation costs. MLBlocks (especially MLBlock-8 and 12) delivers the highest score for both high and low-precision. As expected, Tensor Slice, with its high number of MACs, is the next best design. To be fair, this architecture supports floating-point and element-wise operations, which are not included in MLBlocks. The next architecture is that of Boutros et al. [209].

TABLE 5.5. Implementation results comparison between MLBlocks, Intel Stratix-10 DSP and its alternatives

EB name	Precision	# of MACs		Area	Tech.	f_{max}	$\frac{\text{\# of MACs}}{\text{Area}\times\text{Delay}}$ *	
		8×8	16×16	(um^2)	(nm)		8×8	16×16
Stratix-10 DSP [209]	27×27,18×18	2	1	8404	28	600	1	1
MLBlock-6	8×8	6	-	3315	28	750	9.5	-
MLBlock-8	8×8	8	-	4026	28	750	10.4	-
MLBlock-9	8×8	9	-	4789	28	750	9.9	-
MLBlock-12	8×8	12	-	6093	28	750	10.3	-
MLBlock-6	$(16/8)\times(16/8)$	6	6 (4cyc.)	5776	28	750	5.5	2.7
MLBlock-8	$(16/8)\times(16/8)$	8	8 (4cyc.)	6907	28	750	6.1	3.1
MLBlock-9	$(16/8)\times(16/8)$	9	9 (4cyc.)	8643	28	750	5.5	2.7
MLBlock-12	$(16/8)\times(16/8)$	12	12 (4cyc.)	10378	28	750	6.1	3.0
Boutros et al. [209]	27×27,18×18,9×9,4×4	4	2	8810	28	600	1.9	1.9
Tensor Slice [219]	FP16×FP16, 8×8	64	-	50032	22	371	3.3	-

*normalized

Although it is a suitable replacement for Intel DSPs, backward compatibility prevents it from offering a dense low-precision compute unit. However, in high-precision mode, its performance (similarly for Stratix-10 DSP) only halves whereas MLBlocks performance is reduced by a factor of 4. Nevertheless, while [209] narrows the gap for high-precision computation, MLBlocks still obtain the best performance.

Due to the lack of public information, we were not able to compare our work with other commercial solutions such as the Achronix MLB72 [217] or AI-tensor from Intel [216]. We also note that the AI-tensor implementation is different to MLBlocks. For instance, AI-tensor uses a pipelined adder-tree structure whereas MLBlocks are based on systolic arrays. AI-tensor also includes a double buffering system to cover reloading latencies, which is not present in our architecture.

However, aside from these differences, the computation of these two units are special cases of our architecture, which our tool explores, based on a systolic array architecture. For instance, AI-tensor is able to compute three 10-element dot-products (one input is shared). Focusing only on the computation, we can describe this single computation by a couple of projections: 1) <(1,-,-),10,3,1,1> (one shared 10-element input I and three sets of 10-element weights), and 2) <(1,-,-),10,1,3,1> (10-element shared input W and three batches of 10-element of I). If the number of MAC units is 30, our tool automatically considers these two projections along with other choices such as <(1,-,-),3,10,1> and <(3,1,1),10,1,1>.

5.5 Summary

This chapter proposed a novel methodology for designing coarse-grained embedded blocks for machine learning applications. The procedure is based on modelling the computations and providing a mapping to different configurations of a flexible architecture. Using a set of benchmarks applicable to text, voice and image processing applications, together with design constraints from a commercial Xilinx FPGA, different instances of MLBlock architecture are generated. The results show that MLBlock instances can provide a $6\times$ improvement in compute density over the Xilinx DSP48E2 in the same technology for 8-bit arithmetic without increasing port requirements. MLBlocks with 16-bit configuration, which use serial multipliers, can achieve $2\times$ more computation performance per area compared to the Xilinx DSP48E2. This approach generalises earlier work which either focused on the implementation of a single type of DNN, or creating an FPGA/ASIC generator for arbitrary DNNs; rather, the presented method discovers an efficient embedded block that covers a representative set of algorithms represented by the benchmark set.

CHAPTER 6

Conclusion

This research investigated the suitability of traditional FPGA architectures for low-precision deep learning applications and identified a number of opportunities for FPGA architecture specialization to better support deep learning. Based on quantitative studies, it was found that modification of conventional EBs (Chapters 3 and 4) or substituting them with specialised ones (Chapter 5) lead to significant gains without substantial overheads. Results indicate that minor specialisation in FPGA architectures can dramatically enhance the compute density for embedded DNN applications.

First, the PIR-DSP architecture was introduced that incorporates precision, interconnect and reuse optimisations to better support 2-dimensional low-precision DNN applications. A performance analysis using embedded DNN workloads shows that this architecture can significantly reduce the energy consumption for low-precision implementations, albeit requiring an extra cycle of latency and a 28% area overhead. The main disadvantages of this approach originated in the assumption of backward compatibility, as supporting previous modes limits circuit flexibility and performance gains. While PIR-DSP improves the performance of DNN kernels such as SConv, PW, DW, and FC layers, the extra reconfigurability impacts all applications, most of which do not receive any benefit.

The limitations of traditional FPGA LUT-based architectures for the implementation of compressor trees were then discussed, and instead, several low-cost modifications that lead to significantly improved-performance GPCs and the XnorPopcount operation were proposed. It was shown through the developed ILP technology mapper model that the suggested vendor-agnostic structure, called LUXOR, has minimal implementation overhead and reduces the logic utilisation for compressor trees by up to 36% (average 12–19%) over both Intel and

Xilinx FPGAs. The suggested LUXOR architecture has two key advantages over other solutions reviewed in Chapter 2: 1) its vendor-agnostic characteristic makes it applicable to virtually any LE architecture, and 2) its applicability to a wide range of applications. However, to comprehensively examine this feature, the suggested configurations should be integrated into synthesisers. This was not studied in this work. Afterwards, two carefully-crafted vendor-specific modifications for Xilinx and Intel architectures were introduced, which enhanced the performance gains up to 48% (average 26–34%) at the cost of an additional 3–6% silicon area. Although these modifications are particularly presented for these two architectures, they are still likely to be applicable to LEs from other vendors because of the overall architectural similarities between LE architecture and the solution simplicity.

Finally, a novel methodology for designing coarse-grained embedded blocks for machine learning applications was proposed. The presented technique is the first automatic design space exploration for EB blocks targeting ML and BLAS applications that enables finding near-optimum architectures from a given benchmark set. The approach was based on generalised modelling of the computations and mapping to different configurations of a flexible architecture. Using a benchmark-driven flow, instances of our architecture, called MLBlocks, were generated that offer higher compute density compared to commercialised and academic DSP blocks without increasing port requirements.

Highly specialised EBs's highlighted in this study should not be seen as a replacement for traditional DSP blocks as high-precision arithmetic is still relevant to many FPGA application domains including ML and linear algebra. A partial replacement of DSP blocks with MLBlocks may be a good compromise, with the best ratio depending on the benchmark set. Indeed, MLBlocks could be readily extended to explore this problem.

In summary, contemporary FPGA architectures are heavily optimised for applications that were historically dominated by high-precision digital signal processing applications. The undeniable and fast-growing demand for deploying DNN applications in embedded devices is shifting the required characteristics to low-precision arithmetic and memory-intensive data paths. Current FPGA blocks offer poor native support for such problems, which demands rethinking the architecture at all levels.

6.1 Future Outlook

Moving forward, there are still numerous unanswered research questions regarding FPGA architectures. These, in a sense, also apply to all reconfigurable architectures. First and foremost, the ongoing shift in on-demand applications is trending toward more compute and memory-intensive data paths that necessitate architecture specialisation. Although this thesis focused on compute blocks, rethinking the entire FPGA architecture, including memory and routing resources, is necessary.

Next, as also raised in [184], the new AI and high performance computing (HPC) workloads themself have different characteristics, which poses challenges to supporting such diversity. Future work may also study new EB architectures that can effectively support these two application classes. This will clarify to what degree the FPGAs should maintain their spirit of generality and whether the move towards increasingly domain-specific FPGA architectures will continue.

PIR-DSP and LUXOR architectures that were respectively discussed in Chapters 3 and 4 are handcrafted optimisations open to further improvements. In particular, our initial investigations show that optimizing the critical path of the PIR-DSP by providing a bypass path for the unused pre-adder could enhance frequency by about 25% and potentially remove the extra cycle of latency introduced by our additional pipeline stage. A thorough study of opportunities to further optimise data paths could be undertaken.

Finally, the proposed model in Chapter 5 assumes the algorithm only comprises MAC operations, and there are no output data dependencies apart from accumulation. Future work will involve generalisation to support other classes of problems, such as digital signal processing and stencil computations. Meanwhile, the suggested MLBlock architecture is only a proof of concept. Expanding the design search space by improved input/output schemes, double buffering, and various data stationary types can greatly enhance performance and efficiency. A careful comparison of EBs for low-precision arithmetic compared to implementations using fined-grained logic elements is also planned.

Acronyms

AI: artificial intelligence

ALM: adaptive logic module

ALU: arithmetic logic unit

ANN: artificial neural network

APD: Area-Performance Degree

ASIC: application-specific integrated circuit

BLAS: basic linear algebra subprograms

BN: batch normalisation

BNN: binarised neural network

CGRA: coarse-grained gate array

Cin: carry-in

CLB: configurable logic element

CNN: convolutional neural network

Conv: convolutional

Cout: carry-out

CPLD: complex PLD

CPU: central processing unit

CSA: carry-save adder

DFT: discrete fourier transformation

DNN: deep neural network

DSP: digital signal processing

DW: depth-wise convolution

eASIC: embedded ASIC

EB: embedded block

eFPGA: embedded FPGA

FA: full-adder

FC: fully connected

FF: flip-flop

FIFO: first in first out

FP: floating-point

FPGA: field-programmable gate array

GAN: generative adversarial network

GEMM: general matrix multiply

GPC: generalised parallel counter

GPGPU: general-purpose computing on graphics processing unit

GPU: graphics processing unit

GRU: gated recurrent unit

HA: half-adder

HPC: high performance computing

ILP: integer linear programming

ILSVRC: ImageNet Large Scale Visual Recognition Competition

IO: Input/Output

LAB: logic array block

LB: logic block

LE: logic element

LSTM: long-short-term memory

LUT: look-up table

MAC: multiply-accumulate

ML: machine learning

MLP: multi-layer perceptron

MSB: most significant bit

MVU: matrix-vector unit

NAS: network architecture search

NN: neural network

PAL: programmable array logic

PLA:	programmable logic array
PLD:	programmable logic device
PPA:	performance per area
PW:	point-wise convolution
RCA:	ripple-carry adder
ReLU:	rectified linear unit
RF:	register file
RNN:	recurrent neural network
SConv:	standard convolutional
SPLD:	simple PLD
SR:	shift-register
XPE:	Xilinx Power Estimator

Bibliography

[1] SeyedRamin Rasoulinezhad et al. 'PIR-DSP: An FPGA DSP Block Architecture for Multi-precision Deep Neural Networks'. In: *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*. IEEE, 2019, pp. 35–44. DOI: 10.1109/FCCM.2019.00015. URL: https://doi.org/10.1109/FCCM.2019.00015.

[2] SeyedRamin Rasoulinezhad et al. 'LUXOR: An FPGA Logic Cell Architecture for Efficient Compressor Tree Implementations'. In: *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*. Ed. by Stephen Neuendorffer and Lesley Shannon. ACM, 2020, pp. 161–171. DOI: 10.1145/3373087.3375303. URL: https://doi.org/10.1145/3373087.3375303.

[3] Seyedramin Rasoulinezhad et al. 'Rethinking Embedded Blocks for Machine Learning Applications'. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.1 (2022), 9:1–9:30. DOI: 10.1145/3491234. URL: https://doi.org/10.1145/3491234.

[4] Sorin Mihai Grigorescu et al. 'A survey of deep learning techniques for autonomous driving'. In: *J. Field Robotics* 37.3 (2020), pp. 362–386. DOI: 10.1002/rob.21918. URL: https://doi.org/10.1002/rob.21918.

[5] Angela Zhang et al. 'Shifting machine learning for healthcare from development to deployment and from models to data'. In: *Nature Biomedical Engineering* (July 2022). ISSN: 2157-846X. DOI: 10.1038/s41551-022-00898-y. URL: https://doi.org/10.1038/s41551-022-00898-y.

[6] Andre Esteva et al. 'A guide to deep learning in healthcare'. In: *Nature Medicine* 25.1 (Jan. 2019), pp. 24–29. ISSN: 1546-170X. DOI: 10.1038/s41591-018-0316-z. URL: https://doi.org/10.1038/s41591-018-0316-z.

[7] Ahmet Murat Özbayoglu, Mehmet Ugur Gudelek and Omer Berat Sezer. 'Deep learning for financial applications : A survey'. In: *Appl. Soft Comput.* 93 (2020), p. 106384. DOI: 10.1016/j.asoc.2020.106384. URL: https://doi.org/10.1016/j.asoc.2020.106384.

[8] Allison McCarn Deiana et al. 'Applications and Techniques for Fast Machine Learning in Science'. In: *Frontiers Big Data* 5 (2022), p. 787421. DOI: 10.3389/fdata.2022.787421. URL: https://doi.org/10.3389/fdata.2022.787421.

[9] Alen Rajšp and Iztok Fister. 'A Systematic Literature Review of Intelligent Data Analysis Methods for Smart Sport Training'. In: *Applied Sciences* 10.9 (2020). ISSN: 2076-3417. DOI: 10.3390/app10093013. URL: https://www.mdpi.com/2076-3417/10/9/3013.

[10] Kamran Shaukat et al. 'A Survey on Machine Learning Techniques for Cyber Security in the Last Decade'. In: *IEEE Access* 8 (2020), pp. 222310–222354. DOI: 10.1109/ACCESS.2020.3041951. URL: https://doi.org/10.1109/ACCESS.2020.3041951.

[11] Kaiyuan Guo et al. 'A Survey of FPGA Based Neural Network Accelerator'. In: *CoRR* abs/1712.08934 (2017). arXiv: 1712.08934. URL: http://arxiv.org/abs/1712.08934.

[12] Naveen Mellempudi et al. 'Mixed Low-precision Deep Learning Inference using Dynamic Fixed Point'. In: *CoRR* abs/1701.08978 (2017). arXiv: 1701.08978. URL: http://arxiv.org/abs/1701.08978.

[13] Mohammad Ghasemzadeh, Mohammad Samragh and Farinaz Koushanfar. 'ReBNet: Residual binarized neural network'. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2018, pp. 57–64.

[14] Shibo Wang and Pankaj Kanwar. ‘BFloat16: The secret to high performance on Cloud TPUs’. In: (2019). URL: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

[15] Sean Fox et al. ‘A Block Minifloat Representation for Training Deep Neural Networks’. In: *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL: https://openreview.net/forum?id=6zaTwpNSsQ2.

[16] Matthieu Courbariaux and Yoshua Bengio. ‘BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1’. In: *CoRR* abs/1602.02830 (2016). arXiv: 1602.02830. URL: http://arxiv.org/abs/1602.02830.

[17] Mohammad Rastegari et al. ‘XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks’. In: *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*. 2016, pp. 525–542. DOI: 10.1007/978-3-319-46493-0_32. URL: https://doi.org/10.1007/978-3-319-46493-0_32.

[18] Julian Faraone et al. ‘AddNet: Deep Neural Networks Using FPGA-Optimized Multipliers’. In: *IEEE Trans. Very Large Scale Integr. Syst.* 28.1 (2020), pp. 115–128. DOI: 10.1109/TVLSI.2019.2939429. URL: https://doi.org/10.1109/TVLSI.2019.2939429.

[19] Chenzhuo Zhu et al. ‘Trained Ternary Quantization’. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=S1%5C_pAu9xl.

[20] Shuchang Zhou et al. ‘DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients’. In: *CoRR* abs/1606.06160 (2016). arXiv: 1606.06160. URL: http://arxiv.org/abs/1606.06160.

[21] Andrew G. Howard et al. ‘MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications’. In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[22] Forrest N. Iandola et al. ‘SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size’. In: *CoRR* abs/1602.07360 (2016). arXiv: 1602.07360. URL: http://arxiv.org/abs/1602.07360.

[23] Bichen Wu et al. ‘Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions’. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 2018, pp. 9127–9135. DOI: 10.1109/CVPR.2018.00951. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Wu%5C_Shift%5C_A%5C_%20Zero%5C_CVPR%5C_2018%5C_paper.html.

[24] Xiangyu Zhang et al. ‘ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices’. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 2018, pp. 6848–6856. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/%20Zhang%5C_ShuffleNet%5C_An%5C_Extremely%5C_CVPR%5C_2018%5C_paper.html.

[25] Jonghoon Jin, Aysegul Dundar and Eugenio Culurciello. ‘Flattened Convolutional Neural Networks for Feedforward Acceleration’. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.5474.

[26] Liqiang Lu et al. ‘Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs’. In: *25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017, Napa, CA, USA, April 30 - May 2, 2017*. 2017, pp. 101–108. DOI: 10.1109/FCCM.2017.64. URL: https://doi.org/10.1109/FCCM.2017.64.

[27] Chi Zhang and Viktor K. Prasanna. ‘Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System’. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*. 2017, pp. 35–44. URL: http://dl.acm.org/citation.cfm?id=3021727.

[28] Yu-Hsin Chen et al. 'Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks'. In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138. DOI: 10.1109/JSSC.2016.2616357. URL: https://doi.org/10.1109/JSSC.2016.2616357.

[29] Yaman Umuroglu et al. 'FINN: A Framework for Fast, Scalable Binarized Neural Network Inference'. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 65–74. ISBN: 9781450343541. DOI: 10.1145/3020078.3021744. URL: https://doi.org/10.1145/3020078.3021744.

[30] Eric S. Chung et al. 'Serving DNNs in Real Time at Datacenter Scale with Project Brainwave'. In: *IEEE Micro* 38.2 (2018), pp. 8–20. DOI: 10.1109/MM.2018.022071131. URL: https://doi.org/10.1109/MM.2018.022071131.

[31] Xuan Yang et al. 'Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators'. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 369–383. ISBN: 9781450371025. DOI: 10.1145/3373376.3378514. URL: https://doi.org/10.1145/3373376.3378514.

[32] Tushar Krishna et al. *Data Orchestration in Deep Learning Accelerators*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020. DOI: 10.2200/S01015ED1V01Y202005CAC052. URL: https://doi.org/10.2200/S01015ED1V01Y202005CAC052.

[33] Song Han, Huizi Mao and William J. Dally. 'Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding'. In: *CoRR* abs/1510.00149 (2015). arXiv: 1510.00149. URL: http://arxiv.org/abs/1510.00149.

[34] Baoyuan Liu et al. 'Sparse Convolutional Neural Networks'. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June*

7-12, 2015. 2015, pp. 806–814. DOI: 10.1109/CVPR.2015.7298681. URL: https://doi.org/10.1109/CVPR.2015.7298681.

[35] Mohammad Samragh, Mohammad Ghasemzadeh and Farinaz Koushanfar. 'Customizing Neural Networks for Efficient FPGA Implementation'. In: *25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017, Napa, CA, USA, April 30 - May 2, 2017*. 2017, pp. 85–92. DOI: 10.1109/FCCM.2017.43. URL: https://doi.org/10.1109/FCCM.2017.43.

[36] Jiantao Qiu et al. 'Going Deeper with Embedded FPGA Platform for Convolutional Neural Network'. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 21-23, 2016*. 2016, pp. 26–35. DOI: 10.1145/2847263.2847265. URL: https://doi.org/10.1145/2847263.2847265.

[37] Julian Faraone et al. 'Customizing Low-Precision Deep Neural Networks for FPGAs'. In: *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. 2018, pp. 97–100. DOI: 10.1109/FPL.2018.00025. URL: https://doi.org/10.1109/FPL.2018.00025.

[38] Song Han et al. 'ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA'. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*. 2017, pp. 75–84. URL: http://dl.acm.org/citation.cfm?id=3021745.

[39] Erwei Wang et al. *LUTNet: Rethinking Inference in FPGA Soft Logic*. 2019. arXiv: 1904.00938 [cs.LG].

[40] Seyedramin Rasoulinezhad et al. 'MajorityNets: BNNs Utilising Approximate Popcount for Improved Efficiency'. In: *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*. IEEE, 2019, pp. 339–342. DOI: 10.1109/ICFPT47387.2019.00062. URL: https://doi.org/10.1109/ICFPT47387.2019.00062.

[41] Intel Corp. *UG-S10-DSP Intel Stratix 10 Variable Precision DSP Blocks User Guide*. Tech. rep. Intel, 2018.

[42] Intel. *UG-S10LAB Intel®Stratix®10 Logic Array Blocks and Adaptive Logic Modules User Guide*. Sept. 2018. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/ug-s10-lab.pdf.

[43] Martin Langhammer, Sergey Gribok and Gregg Baeckler. 'High Density Pipelined 8bit Multiplier Systolic Arrays for FPGA'. In: *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. Seaside, CA, USA: Association for Computing Machinery, 2020, p. 322. ISBN: 9781450370998. DOI: 10.1145/3373087.3375352. URL: https://doi.org/10.1145/3373087.3375352.

[44] Xilinx. *Deep Learning with INT8 Optimization on Xilinx Devices (WP486)*. Version 1.0.1. https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf. 2017.

[45] Ananda Samajdar et al. 'Scaling the Cascades: Interconnect-Aware FPGA Implementation of Machine Learning Problems'. In: *29th International Conference on Field Programmable Logic and Applications, FPL 2019, Barcelona, Spain, September 8-12, 2019*. 2019, pp. 342–349. DOI: 10.1109/FPL.2019.00061. URL: https://doi.org/10.1109/FPL.2019.00061.

[46] Christopher S. Wallace. 'A Suggestion for a Fast Multiplier'. In: *IEEE Trans. Electronic Computers* 13.1 (1964), pp. 14–17. DOI: 10.1109/PGEC.1964.263830. URL: https://doi.org/10.1109/PGEC.1964.263830.

[47] Luigi Dadda. 'Some schemes for parallel multipliers'. In: *Alta frequenza* 34 (1965), pp. 349–356.

[48] Earl E. Swartzlander Jr. 'Parallel Counters'. In: *IEEE Trans. Computers* 22.11 (1973), pp. 1021–1024. DOI: 10.1109/T-C.1973.223639. URL: https://doi.org/10.1109/T-C.1973.223639.

[49] Ajay K. Verma, Philip Brisk and Paolo Ienne. 'Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits'. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 27.10 (2008), pp. 1761–1774. DOI: 10.

1109/TCAD.2008.2003280. URL: https://doi.org/10.1109/TCAD.2008.2003280.

[50] Hadi Parandeh-Afshar, Philip Brisk and Paolo Ienne. 'An FPGA Logic Cell and Carry Chain Configurable as a 6:2 or 7:2 Compressor'. In: *TRETS* 2.3 (2009), 19:1–19:42. DOI: 10.1145/1575774.1575778. URL: https://doi.org/10.1145/1575774.1575778.

[51] Vivienne Sze et al. 'Efficient Processing of Deep Neural Networks: A Tutorial and Survey'. In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. DOI: 10.1109/JPROC.2017.2761740. URL: https://doi.org/10.1109/JPROC.2017.2761740.

[52] Murray Campbell, A. Joseph Hoane and Feng-hsiung Hsu. 'Deep Blue'. In: *Artif. Intell.* 134.1–2 (Jan. 2002), pp. 57–83. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(01)00129-1. URL: https://doi.org/10.1016/S0004-3702(01)00129-1.

[53] David Silver et al. 'A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play'. In: *Science* 362.6419 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404. URL: https://www.science.org/doi/abs/10.1126/science.aar6404.

[54] Shiv Ram Dubey, Satish Kumar Singh and Bidyut. B. Chaudhuri. 'A Comprehensive Survey and Performance Analysis of Activation Functions in Deep Learning'. In: *ArXiv* abs/2109.14545 (2021).

[55] Vinod Nair and Geoffrey E. Hinton. 'Rectified Linear Units Improve Restricted Boltzmann Machines'. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*. 2010, pp. 807–814. URL: https://icml.cc/Conferences/2010/papers/432.pdf.

[56] Kaiming He et al. 'Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification'. In: *2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015*. 2015, pp. 1026–

1034. DOI: 10.1109/ICCV.2015.123. URL: https://doi.org/10.1109/ICCV.2015.123.

[57] Djork-Arné Clevert, Thomas Unterthiner and Sepp Hochreiter. 'Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)'. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. 2016. URL: http://arxiv.org/abs/1511.07289.

[58] David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams. 'Neurocomputing: Foundations of Research'. In: *Learning Representations by Back-propagating Errors*. Ed. by James A. Anderson and Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988, pp. 696–699. ISBN: 0-262-01097-6. URL: http://dl.acm.org/citation.cfm?id=65669.104451.

[59] Yong Yu et al. 'A Review of Recurrent Neural Networks: LSTM Cells and Network Architectures'. In: *Neural Comput.* 31.7 (2019), pp. 1235–1270. DOI: 10.1162/neco_a_01199. URL: https://doi.org/10.1162/neco%5C_a%5C_01199.

[60] Paul A. Merolla et al. 'A million spiking-neuron integrated circuit with a scalable communication network and interface'. In: *Science* 345.6197 (2014), pp. 668–673. ISSN: 0036-8075. DOI: 10.1126/science.1254642. eprint: https://science.sciencemag.org/content/345/6197/668.full.pdf. URL: https://science.sciencemag.org/content/345/6197/668.

[61] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton. 'ImageNet Classification with Deep Convolutional Neural Networks'. In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. 2012, pp. 1106–1114. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

[62] Sebastian Ruder. ‘An overview of gradient descent optimization algorithms’. In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[63] Diederik P. Kingma and Jimmy Ba. ‘Adam: A Method for Stochastic Optimization’. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[64] Shiliang Sun et al. ‘A Survey of Optimization Methods From a Machine Learning Perspective’. In: *IEEE Trans. Cybern.* 50.8 (2020), pp. 3668–3681. DOI: 10.1109/TCYB.2019.2950779. URL: https://doi.org/10.1109/TCYB.2019.2950779.

[65] Leslie Pack Kaelbling, Michael L Littman and Andrew W Moore. ‘Reinforcement learning: A survey’. In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[66] Arild Nøkland. ‘Direct Feedback Alignment Provides Learning in Deep Neural Networks’. In: *Annual Conference on Neural Information Processing Systems*. 2016, pp. 1037–1045. URL: https://proceedings.neurips.cc/paper/2016/hash/d490d7b4576290fa60eb31b5fc917ad1-Abstract.html.

[67] Yoshua Bengio. ‘Learning Deep Architectures for AI’. In: *Found. Trends Mach. Learn.* 2.1 (2009), pp. 1–127. DOI: 10.1561/2200000006. URL: https://doi.org/10.1561/2200000006.

[68] Maurizio Capra et al. ‘Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead’. In: *IEEE Access* 8 (2020), pp. 225134–225180. DOI: 10.1109/ACCESS.2020.3039858. URL: https://doi.org/10.1109/ACCESS.2020.3039858.

[69] Lukás Sekanina. ‘Neural Architecture Search and Hardware Accelerator Co-Search: A Survey’. In: *IEEE Access* 9 (2021), pp. 151337–151362. DOI: 10.1109/ACCESS.2021.3126685. URL: https://doi.org/10.1109/ACCESS.2021.3126685.

[70] Almadena Yu. Chtchelkanova et al. ‘Parallel implementation of BLAS: general techniques for Level 3 BLAS’. In: *Concurr. Pract. Exp.* 9.9 (1997), pp. 837–857. DOI: 10.1002/(SICI)1096-9128(199709)9:9\<837::AID-CPE267\>3.0.CO;2-2. URL: https://doi.org/10.1002/(SICI)1096-9128(199709)9:9%5C%3C837::AID-CPE267%5C%3E3.0.CO;2-2.

[71] Lingran Zhao, Zhen Dong and Kurt Keutzer. ‘Analysis of Quantization on MLP-based Vision Models’. In: *CoRR* abs/2209.06383 (2022). DOI: 10.48550/arXiv.2209.06383. arXiv: 2209.06383. URL: https://doi.org/10.48550/arXiv.2209.06383.

[72] Chuanxin Tang et al. ‘Sparse MLP for Image Recognition: Is Self-Attention Really Necessary?’ In: *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*. AAAI Press, 2022, pp. 2344–2351. URL: https://ojs.aaai.org/index.php/AAAI/article/view/20133.

[73] George Cybenko. ‘Approximation by superpositions of a sigmoidal function’. In: *Math. Control. Signals Syst.* 2.4 (1989), pp. 303–314. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[74] R. D. Jones et al. ‘Function approximation and time series prediction with neural networks’. In: *1990 IJCNN International Joint Conference on Neural Networks* (1990), 649–665 vol.1.

[75] Hanxiao Liu et al. ‘Pay Attention to MLPs’. In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. Ed. by Marc’Aurelio Ranzato et al. 2021, pp. 9204–9215. URL: https://proceedings.neurips.cc/paper/2021/hash/4cc05b35c2f937c5bd9e7d41d3686fff-Abstract.html.

[76] Ilya O. Tolstikhin et al. ‘MLP-Mixer: An all-MLP Architecture for Vision’. In: *Annual Conference on Neural Information Processing Systems, NeurIPS*. 2021, pp. 24261–

24272. URL: https://proceedings.neurips.cc/paper/2021/file/cba0a4ee5ccd02fda0fe3f9a3e7b89fe-Paper.pdf.

[77] Fisher Yu and Vladlen Koltun. 'Multi-Scale Context Aggregation by Dilated Convolutions'. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1511.07122.

[78] Abdul Jabbar, Xi Li and Bourahla Omar. 'A Survey on Generative Adversarial Networks: Variants, Applications, and Training'. In: *ACM Comput. Surv.* 54.8 (2022), 157:1–157:49. DOI: 10.1145/3463475. URL: https://doi.org/10.1145/3463475.

[79] Junhai Zhai et al. 'Autoencoder and Its Various Variants'. In: *IEEE International Conference on Systems, Man, and Cybernetics, SMC 2018, Miyazaki, Japan, October 7-10, 2018*. IEEE, 2018, pp. 415–419. DOI: 10.1109/SMC.2018.00080. URL: https://doi.org/10.1109/SMC.2018.00080.

[80] Ademola Enitan Ilesanmi and Taiwo Ilesanmi. 'Methods for image denoising using convolutional neural network: a review'. In: *Complex & Intelligent Systems* (2021).

[81] Guansong Pang et al. 'Deep Learning for Anomaly Detection: A Review'. In: *ACM Comput. Surv.* 54.2 (2021), 38:1–38:38. DOI: 10.1145/3439950. URL: https://doi.org/10.1145/3439950.

[82] Shervin Minaee et al. 'Image Segmentation Using Deep Learning: A Survey'. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 44.7 (2022), pp. 3523–3542. DOI: 10.1109/TPAMI.2021.3059968. URL: https://doi.org/10.1109/TPAMI.2021.3059968.

[83] Theodoros Georgiou et al. 'A survey of traditional and deep learning-based feature descriptors for high dimensional data in computer vision'. In: *Int. J. Multim. Inf. Retr.* 9.3 (2020), pp. 135–170. DOI: 10.1007/s13735-019-00183-w. URL: https://doi.org/10.1007/s13735-019-00183-w.

[84] Anu Jagannath, Jithin Jagannath and Prem Sagar Pattanshetty Vasanth Kumar. 'A Comprehensive Survey on Radio Frequency (RF) Fingerprinting: Traditional Approaches, Deep Learning, and Open Challenges'. In: *CoRR* abs/2201.00680 (2022). arXiv: 2201.00680. URL: https://arxiv.org/abs/2201.00680.

[85] Florentin Bieder, Robin Sandkühler and Philippe C. Cattin. 'Comparison of Methods Generalizing Max- and Average-Pooling'. In: *CoRR* abs/2103.01746 (2021). arXiv: 2103.01746. URL: https://arxiv.org/abs/2103.01746.

[86] Lei Huang et al. 'Normalization Techniques in Training DNNs: Methodology, Analysis and Application'. In: *CoRR* abs/2009.12836 (2020). arXiv: 2009.12836. URL: https://arxiv.org/abs/2009.12836.

[87] Sergey Ioffe and Christian Szegedy. 'Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift'. In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.html.

[88] Tim Salimans and Diederik P. Kingma. 'Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks'. In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*. Ed. by Daniel D. Lee et al. 2016, p. 901. URL: https://proceedings.neurips.cc/paper/2016/hash/ed265bc903a5a097f61d3ec064d96d2e-Abstract.html.

[89] Lei Jimmy Ba, Jamie Ryan Kiros and Geoffrey E. Hinton. 'Layer Normalization'. In: *CoRR* abs/1607.06450 (2016). arXiv: 1607.06450. URL: http://arxiv.org/abs/1607.06450.

[90] Karen Simonyan and Andrew Zisserman. 'Very Deep Convolutional Networks for Large-Scale Image Recognition'. In: *CoRR* abs/1409.1556 (2014). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

[91] Kaiming He et al. 'Deep Residual Learning for Image Recognition'. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV,*

USA, June 27-30, 2016. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90. URL: https://doi.org/10.1109/CVPR.2016.90.

[92] Kaiyuan Guo et al. '[DL] A Survey of FPGA-based Neural Network Inference Accelerators'. In: *TRETS* 12.1 (2018), 2:1–2:26. URL: https://dl.acm.org/citation.cfm?id=3289185.

[93] Charles C. Tappert. 'Who Is the Father of Deep Learning?' In: *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. 2019, pp. 343–348. DOI: 10.1109/CSCI49370.2019.00067.

[94] J. Orbach. 'Principles of Neurodynamics. Perceptrons and the Theory of Brain Mechanisms.' In: *Archives of General Psychiatry* 7.3 (Sept. 1962), pp. 218–219. ISSN: 0003-990X. DOI: 10.1001/archpsyc.1962.01720030064010. eprint: https://jamanetwork.com/journals/jamapsychiatry/articlepdf/488205/archpsyc_7_3_010.pdf. URL: https://doi.org/10.1001/archpsyc.1962.01720030064010.

[95] Yann LeCun et al. 'Backpropagation Applied to Handwritten Zip Code Recognition'. In: *Neural Comput.* 1.4 (1989), pp. 541–551. DOI: 10.1162/neco.1989.1.4.541. URL: https://doi.org/10.1162/neco.1989.1.4.541.

[96] Yann LeCun et al. 'Handwritten digit recognition: applications of neural network chips and automatic learning'. In: *IEEE Commun. Mag.* 27.11 (1989), pp. 41–46. DOI: 10.1109/35.41400. URL: https://doi.org/10.1109/35.41400.

[97] Yann LeCun et al. 'Gradient-based learning applied to document recognition'. In: *Proc. IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791. URL: https://doi.org/10.1109/5.726791.

[98] Christian Szegedy et al. 'Going deeper with convolutions'. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594. URL: https://doi.org/10.1109/CVPR.2015.7298594.

[99] Jia Deng et al. 'ImageNet: A large-scale hierarchical image database'. In: *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society,

2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848. URL: https://doi.org/10.1109/CVPR.2009.5206848.

[100] Simone Bianco et al. 'Benchmark Analysis of Representative Deep Neural Network Architectures'. In: *IEEE Access* 6 (2018), pp. 64270–64277. DOI: 10.1109/ACCESS.2018.2877890. URL: https://doi.org/10.1109/ACCESS.2018.2877890.

[101] Hieu Pham et al. 'Meta Pseudo Labels'. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*. Computer Vision Foundation / IEEE, 2021, pp. 11557–11568. DOI: 10.1109/CVPR46437.2021.01139. URL: https://openaccess.thecvf.com/content/CVPR2021/html/Pham%5C_Meta%5C_Pseudo%5C_Labels%5C_CVPR%5C_2021%5C_paper.html.

[102] Hadjer Benmeziane et al. 'A Comprehensive Survey on Hardware-Aware Neural Architecture Search'. In: *CoRR* abs/2101.09336 (2021). arXiv: 2101.09336. URL: https://arxiv.org/abs/2101.09336.

[103] Pengzhen Ren et al. 'A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions'. In: *ACM Comput. Surv.* 54.4 (2022), 76:1–76:34. DOI: 10.1145/3447582. URL: https://doi.org/10.1145/3447582.

[104] Eriko Nurvitadhi et al. 'Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC'. In: *International Conference on Field-Programmable Technology, FPT 2016, Xi'an, China, December 7-9, 2016*. 2016, pp. 77–84. DOI: 10.1109/FPT.2016.7929192. URL: https://doi.org/10.1109/FPT.2016.7929192.

[105] Saining Xie et al. 'Aggregated Residual Transformations for Deep Neural Networks'. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2017, pp. 5987–5995. DOI: 10.1109/CVPR.2017.634. URL: https://doi.org/10.1109/CVPR.2017.634.

[106] Forrest N. Iandola et al. 'SqueezeBERT: What can computer vision teach NLP about efficient neural networks?' In: *Proceedings of SustaiNLP: Workshop on Simple and*

Efficient Natural Language Processing, SustaiNLP@EMNLP 2020, Online, November 20, 2020. Ed. by Nafise Sadat Moosavi et al. Association for Computational Linguistics, 2020, pp. 124–135. DOI: 10.18653/v1/2020.sustainlp-1.17. URL: https://doi.org/10.18653/v1/2020.sustainlp-1.17.

[107] Min Lin, Qiang Chen and Shuicheng Yan. *Network In Network*. 2013. DOI: 10.48550/ARXIV.1312.4400. URL: https://arxiv.org/abs/1312.4400.

[108] Mark Sandler et al. 'Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation'. In: *CoRR* abs/1801.04381 (2018). arXiv: 1801.04381. URL: http://arxiv.org/abs/1801.04381.

[109] Jonathan Huang et al. 'Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors'. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. 2017, pp. 3296–3297. DOI: 10.1109/CVPR.2017.351. URL: https://doi.org/10.1109/CVPR.2017.351.

[110] Caiwen Ding et al. 'CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices'. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*. 2017, pp. 395–408. DOI: 10.1145/3123939.3124552. URL: https://doi.org/10.1145/3123939.3124552.

[111] Shmuel Winograd. *Arithmetic complexity of computations*. Vol. 33. Siam, 1980.

[112] Qingcheng Xiao et al. 'Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs'. In: *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*. 2017, 62:1–62:6. DOI: 10.1145/3061639.3062244. URL: https://doi.org/10.1145/3061639.3062244.

[113] Barret Zoph et al. 'Learning Transferable Architectures for Scalable Image Recognition'. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. 2018, pp. 8697–8710. DOI: 10.1109/CVPR.2018.00907. URL: http://openaccess.thecvf.

com%20/content%5C_cvpr%5C_2018/html/Zoph%5C_Learning%5C_Transferable%5C_Architectures%5C_CVPR%5C_2018%5C_paper.html.

[114] Ningning Ma et al. 'ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design'. In: *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XIV*. 2018, pp. 122–138. DOI: 10.1007/978-3-030-01264-9_8. URL: https://doi.org/10.1007/978-3-030-01264-9_8.

[115] Amir Gholami et al. 'A Survey of Quantization Methods for Efficient Neural Network Inference'. In: *CoRR* abs/2103.13630 (2021). arXiv: 2103.13630. URL: https://arxiv.org/abs/2103.13630.

[116] Yijin Guan et al. 'FPGA-based accelerator for long short-term memory recurrent neural networks'. In: *22nd Asia and South Pacific Design Automation Conference, ASP-DAC 2017, Chiba, Japan, January 16-19, 2017*. IEEE, 2017, pp. 629–634. ISBN: 978-1-5090-1558-0. DOI: 10.1109/ASPDAC.2017.7858394. URL: https://doi.org/10.1109/ASPDAC.2017.7858394.

[117] Huimin Li et al. 'A high performance FPGA-based accelerator for large-scale convolutional neural networks'. In: *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*. 2016, pp. 1–9. DOI: 10.1109/FPL.2016.7577308. URL: https://doi.org/10.1109/FPL.2016.7577308.

[118] Chen Zhang et al. 'Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks'. In: *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016*. 2016, p. 12. DOI: 10.1145/2966986.2967011. URL: https://doi.org/10.1145/2966986.2967011.

[119] Thomas Elsken, Jan Hendrik Metzen and Frank Hutter. 'Neural Architecture Search: A Survey'. In: *J. Mach. Learn. Res.* 20 (2019), 55:1–55:21. URL: http://jmlr.org/papers/v20/18-598.html.

[120] Matthieu Courbariaux, Yoshua Bengio and Jean-Pierre David. *Training deep neural networks with low precision multiplications*. 2014. DOI: 10.48550/ARXIV.1412.7024. URL: https://arxiv.org/abs/1412.7024.

[121] Suyog Gupta et al. 'Deep Learning with Limited Numerical Precision'. In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 1737–1746. URL: http://proceedings.mlr.press/v37/gupta15.html.

[122] Paulius Micikevicius et al. 'Mixed Precision Training'. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=r1gs9JgRZ.

[123] *IEEE standard for binary floating-point arithmetic - IEEE standard 754-1985*. Beuth, 1985.

[124] Paulius Micikevicius et al. 'FP8 Formats for Deep Learning'. In: *CoRR* abs/2209.05433 (2022). DOI: 10.48550/arXiv.2209.05433. arXiv: 2209.05433. URL: https://doi.org/10.48550/arXiv.2209.05433.

[125] William Dally. *High-performance hardware for machine learning*. Tech. rep. NIPS Tutorial, 2015. URL: https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf.

[126] Mark Horowitz. '1.1 Computing's energy problem (and what we can do about it)'. In: *2014 IEEE International Conference on Solid-State Circuits Conference, ISSCC 2014, Digest of Technical Papers, San Francisco, CA, USA, February 9-13, 2014*. IEEE, 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323. URL: https://doi.org/10.1109/ISSCC.2014.6757323.

[127] Guandao Yang et al. 'SWALP : Stochastic Weight Averaging in Low Precision Training'. In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning

Research. PMLR, 2019, pp. 7015–7024. URL: http://proceedings.mlr.press/v97/yang19d.html.

[128] Xiao Sun et al. 'Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks'. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach et al. 2019, pp. 4901–4910. URL: https://proceedings.neurips.cc/paper/2019/hash/65fc9fb4897a89789352e211ca2d398f-Abstract.html.

[129] David Goldberg. 'What Every Computer Scientist Should Know About Floating-Point Arithmetic'. In: *ACM Comput. Surv.* 23.1 (1991), pp. 5–48. DOI: 10.1145/103162.103163. URL: https://doi.org/10.1145/103162.103163.

[130] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture, V1.0*. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[131] Naigang Wang et al. 'Training Deep Neural Networks with 8-bit Floating Point Numbers'. In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio et al. 2018, pp. 7686–7695. URL: https://proceedings.neurips.cc/paper/2018/hash/335d3d1cd7ef05ec77714a215134914c-Abstract.html.

[132] Léopold Cambier et al. 'Shifted and Squeezed 8-bit Floating Point format for Low-Precision Training of Deep Neural Networks'. In: *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL: https://openreview.net/forum?id=Bkxe2AVtPS.

[133] Nhut-Minh Ho and Weng-Fai Wong. 'Exploiting half precision arithmetic in Nvidia GPUs'. In: *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*. IEEE, 2017, pp. 1–7. DOI: 10.1109/HPEC.2017.8091072. URL: https://doi.org/10.1109/HPEC.2017.8091072.

[134] Intel. *BFLOAT16 – Hardware Numerics Definition (Revision 1.0)*. 2018. URL: https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf.

[135] Jack Choquette et al. 'NVIDIA A100 Tensor Core GPU: Performance and Innovation'. In: *IEEE Micro* 41.2 (2021), pp. 29–35. DOI: 10.1109/MM.2021.3061394. URL: https://doi.org/10.1109/MM.2021.3061394.

[136] Maolin Wang et al. 'NITI: Training Integer Neural Networks Using Integer-Only Arithmetic'. In: *IEEE Trans. Parallel Distributed Syst.* 33.11 (2022), pp. 3249–3261. DOI: 10.1109/TPDS.2022.3149787. URL: https://doi.org/10.1109/TPDS.2022.3149787.

[137] Ron Banner et al. 'Scalable methods for 8-bit training of neural networks'. In: *Annual Conference on Neural Information Processing Systems 2018, NeurIPS*. 2018, pp. 5151–5159. URL: https://proceedings.neurips.cc/paper/2018/hash/e82c4b19b8151ddc25d4d93baf7b908f-Abstract.html.

[138] Mario Drumond et al. 'Training DNNs with Hybrid Block Floating Point'. In: *Annual Conference on Neural Information Processing Systems 2018, NeurIPS*. 2018, pp. 451–461. URL: https://proceedings.neurips.cc/paper/2018/hash/6a9aeddfc689c1d0e3b9ccc3ab651bc5-Abstract.html.

[139] Kaiyuan Guo et al. 'Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA'. In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 37.1 (2018), pp. 35–47. DOI: 10.1109/TCAD.2017.2705069. URL: https://doi.org/10.1109/TCAD.2017.2705069.

[140] Benoit Jacob et al. 'Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference'. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Jacob%5C_Quantization%5C_and%5C_Training%5C_CVPR%5C_2018%5C_paper.html.

[141] Bichen Wu et al. ‘Mixed Precision Quantization of ConvNets via Differentiable Neural Architecture Search’. In: *CoRR* abs/1812.00090 (2018). arXiv: 1812.00090. URL: http://arxiv.org/abs/1812.00090.

[142] Ahmed T. Elthakeb et al. ‘ReLeQ: A Reinforcement Learning Approach for Deep Quantization of Neural Networks’. In: *CoRR* abs/1811.01704 (2018). arXiv: 1811.01704. URL: http://arxiv.org/abs/1811.01704.

[143] Kuan Wang et al. ‘HAQ: Hardware-Aware Automated Quantization With Mixed Precision’. In: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*. Computer Vision Foundation / IEEE, 2019, pp. 8612–8620. DOI: 10.1109/CVPR.2019.00881. URL: http://openaccess.thecvf.com/content%5C_CVPR%5C_2019/html/Wang%5C_HAQ%5C_Hardware-Aware%5C_Automated%5C_Quantization%5C_With%5C_Mixed%5C_Precision%5C_CVPR%5C_2019%5C_paper.html.

[144] Tianzhe Wang et al. ‘APQ: Joint Search for Network Architecture, Pruning and Quantization Policy’. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*. Computer Vision Foundation / IEEE, 2020, pp. 2075–2084. DOI: 10.1109/CVPR42600.2020.00215. URL: https://openaccess.thecvf.com/content%5C_CVPR%5C_2020/html/Wang%5C_APQ%5C_Joint%5C_Search%5C_for%5C_Network%5C_Architecture%5C_Pruning%5C_and%5C_Quantization%5C_Policy%5C_CVPR%5C_2020%5C_paper.html.

[145] Haoping Bai et al. ‘BatchQuant: Quantized-for-all Architecture Search with Robust Quantizer’. In: *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*. Ed. by Marc’Aurelio Ranzato et al. 2021, pp. 1074–1085. URL: https://proceedings.neurips.cc/paper/2021/hash/08aee6276db142f4b8ac98fb8ee0ed1b-Abstract.html.

[146] Shuchang Zhou et al. ‘Balanced Quantization: An Effective and Efficient Approach to Quantized Neural Networks’. In: *J. Comput. Sci. Technol.* 32.4 (2017), pp. 667–682.

DOI: 10.1007/s11390-017-1750-y. URL: https://doi.org/10.1007/s11390-017-1750-y.

[147] Nvidia. *Nvidia Tesla V100 GPU Architecture, WP-08608-001*. Version 1.1. 2017. URL: www.images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[148] Norman P. Jouppi et al. 'A Domain-specific Architecture for Deep Neural Networks'. In: *Commun. ACM* 61.9 (Aug. 2018), pp. 50–59. ISSN: 0001-0782. DOI: 10.1145/3154484. URL: http://doi.acm.org/10.1145/3154484.

[149] Yijin Guan et al. 'FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates'. In: *25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017, Napa, CA, USA, April 30 - May 2, 2017*. 2017, pp. 152–159. DOI: 10.1109/FCCM.2017.25. URL: https://doi.org/10.1109/FCCM.2017.25.

[150] Adrien Prost-Boucle et al. 'Scalable high-performance architecture for convolutional ternary neural networks on FPGA'. In: *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*. 2017, pp. 1–7. DOI: 10.23919/FPL.2017.8056850. URL: https://doi.org/10.23919/FPL.2017.8056850.

[151] Duncan J. M. Moss et al. 'High performance binary neural networks on the Xeon FPGA™ platform'. In: *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*. 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056823. URL: https://doi.org/10.23919/FPL.2017.8056823.

[152] Ritchie Zhao et al. 'Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs'. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017*. 2017, pp. 15–24. URL: http://dl.acm.org/citation.cfm?id=3021741.

[153] Hiroki Nakahara et al. 'A Batch Normalization Free Binarized Convolutional Deep Neural Network on an FPGA (Abstract Only)'. In: *Proceedings of the ACM/SIGDA*

International Symposium on Field-Programmable Gate Arrays, FPGA. 2017, p. 290. URL: http://dl.acm.org/citation.cfm?id=3021782.

[154] Hiroki Nakahara, Tomoya Fujii and Shimpei Sato. 'A fully connected layer elimination for a binarized convolutional neural network on an FPGA'. In: *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*. 2017, pp. 1–4. DOI: 10.23919/FPL.2017.8056771. URL: https://doi.org/10.23919/FPL.2017.8056771.

[155] Wenlin Chen et al. 'Compressing Neural Networks with the Hashing Trick'. In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. 2015, pp. 2285–2294. URL: http://jmlr.org/proceedings/papers/v37/chenc15.html.

[156] Fengfu Li and Bin Liu. 'Ternary Weight Networks'. In: *CoRR* abs/1605.04711 (2016). arXiv: 1605.04711. URL: http://arxiv.org/abs/1605.04711.

[157] Mingxing Tan and Quoc V. Le. 'EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks'. In: *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 6105–6114. URL: http://proceedings.mlr.press/v97/tan19a.html.

[158] NVIDIA. *WP-09183-001_v01 NVIDIA Turing GPU Architecture - Graphics Reinvented*. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.

[159] Norman P. Jouppi et al. 'A domain-specific supercomputer for training deep neural networks'. In: *Commun. ACM* 63.7 (2020), pp. 67–78. DOI: 10.1145/3360307. URL: https://doi.org/10.1145/3360307.

[160] Rehan Hameed et al. 'Understanding sources of inefficiency in general-purpose chips'. In: *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*. Ed. by André Seznec, Uri C. Weiser and Ronny Ronen.

ACM, 2010, pp. 37–47. DOI: 10.1145/1815961.1815968. URL: https://doi.org/10.1145/1815961.1815968.

[161] Deepak Ghimire, Dayoung Kil and Seong-heum Kim. ‘A Survey on Efficient Convolutional Neural Networks and Hardware Acceleration’. In: *Electronics* 11.6 (2022). ISSN: 2079-9292. DOI: 10.3390/electronics11060945. URL: https://www.mdpi.com/2079-9292/11/6/945.

[162] Yu-Hsin Chen et al. ‘Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices’. In: *IEEE J. Emerg. Sel. Topics Circuits Syst.* 9.2 (2019), pp. 292–308. DOI: 10.1109/JETCAS.2019.2910232. URL: https://doi.org/10.1109/JETCAS.2019.2910232.

[163] Yuhao Zhang et al. ‘A Practical Highly Paralleled ReRAM-Based DNN Accelerator by Reusing Weight Pattern Repetitions’. In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41.4 (2022), pp. 922–935. DOI: 10.1109/TCAD.2021.3071116. URL: https://doi.org/10.1109/TCAD.2021.3071116.

[164] Angshuman Parashar et al. ‘SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks’. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017.* ACM, 2017, pp. 27–40. DOI: 10.1145/3079856.3080254. URL: https://doi.org/10.1145/3079856.3080254.

[165] Lukas Cavigelli and Luca Benini. ‘Origami: A 803-GOp/s/W Convolutional Network Accelerator’. In: *IEEE Trans. Circuits Syst. Video Technol.* 27.11 (2017), pp. 2461–2475. DOI: 10.1109/TCSVT.2016.2592330. URL: https://doi.org/10.1109/TCSVT.2016.2592330.

[166] Shouyi Yin et al. ‘A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications’. In: *IEEE J. Solid State Circuits* 53.4 (2018), pp. 968–982. DOI: 10.1109/JSSC.2017.2778281. URL: https://doi.org/10.1109/JSSC.2017.2778281.

[167] Intel. *3rd Gen Intel® Xeon®Scalable processors*. 2022. URL: www.intel.ca/content/dam/www/central-libraries/us/en/documents/3rd-gen-intel-xeon-scalable-processors-product-brief.pdf.

[168] Nvidia. *Nvidia H100 Tensor Core GPU*. 2022. URL: www.resources.nvidia.com/en-us-gpu-resources/h100-datasheet-24306?lx=CPwSfP.

[169] Groq team. *Product Spec Sheet - Groq Chip Processor v1.5*. 2022. URL: groq.com/wp-content/uploads/2022/10/GroqChip%E2%84%A2-Processor-Product-Brief-v1.5.pdf.

[170] Xilinx Inc. *Versal® Architecture and Product Data Sheet: Overview (DS950)*. Tech. rep. Xilinx, 2022. URL: https://docs.xilinx.com/v/u/en-US/ds950-versal-overview.

[171] Sean Fox et al. 'Training Deep Neural Networks in Low-Precision with High Accuracy Using FPGAs'. In: *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*. IEEE, 2019, pp. 1–9. DOI: 10.1109/ICFPT47387.2019.00009. URL: https://doi.org/10.1109/ICFPT47387.2019.00009.

[172] Intel. *Intel Advanced Vector Extensions 512 (Intel AVX-512)*. 2022. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-solution-brief.html.

[173] John D. Owens et al. 'A Survey of General-Purpose Computation on Graphics Hardware'. In: *26th Annual Conference of the European Association for Computer Graphics, Eurographics 2005 - State of the Art Reports, Dublin, Ireland, August 29 - September 2, 2005*. Ed. by Yiorgos Chrysanthou and Marcus A. Magnor. Eurographics Association, 2005, pp. 21–51. DOI: 10.2312/egst.20051043. URL: https://doi.org/10.2312/egst.20051043.

[174] Eriko Nurvitadhi et al. 'Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?' In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA*. 2017, pp. 5–14. URL: http://dl.acm.org/citation.cfm?id=3021740.

[175] Stefano Markidis et al. 'NVIDIA Tensor Core Programmability, Performance & Precision'. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2018, Vancouver, BC, Canada, May 21-25, 2018*. IEEE

Computer Society, 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091. URL: https://doi.org/10.1109/IPDPSW.2018.00091.

[176] Zhe Jia et al. *Dissecting the Graphcore IPU Architecture via Microbenchmarking*. Tech. rep. High Performance Computing R&D Team, Citadel, 2019. URL: http://arxiv.org/abs/1912.03413.

[177] Adam Paszke et al. 'Automatic differentiation in PyTorch'. In: *NIPS-W*. 2017.

[178] Google. *Tensorflow source code*. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/rnn.py.

[179] Wenyan Lu et al. 'FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks'. In: *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*. 2017, pp. 553–564. DOI: 10.1109/HPCA.2017.29. URL: https://doi.org/10.1109/HPCA.2017.29.

[180] Hardik Sharma et al. 'Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network'. In: *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*. 2018, pp. 764–775. DOI: 10.1109/ISCA.2018.00069. URL: https://doi.org/10.1109/ISCA.2018.00069.

[181] Mathew Hall and Vaughn Betz. 'HPIPE: Heterogeneous Layer-Pipelined and Sparse-Aware CNN Inference for FPGAs'. In: *FPGA '20: The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, February 23-25, 2020*. Ed. by Stephen Neuendorffer and Lesley Shannon. ACM, 2020, p. 320. DOI: 10.1145/3373087.3375380. URL: https://doi.org/10.1145/3373087.3375380.

[182] B. Holdsworth and R. C. Woods. *Digital logic design*. 4th ed. Oxford ; Boston: Newnes, 2002. ISBN: 9780750645829.

[183] Stephen Dean Brown and Jonathan Rose. 'FPGA and CPLD Architectures: A Tutorial'. In: *IEEE Des. Test Comput.* 13.2 (1996), pp. 42–57. DOI: 10.1109/54.500200. URL: https://doi.org/10.1109/54.500200.

[184] Andrew Boutros and Vaughn Betz. ‘FPGA Architecture: Principles and Progression’. In: *IEEE Circuits and Systems Magazine* 21.2 (2021), pp. 4–29. DOI: 10.1109/MCAS.2021.3071607.

[185] Lesley Shannon et al. ‘Technology Scaling in FPGAs: Trends in Applications and Architectures’. In: *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*. 2015, pp. 1–8. DOI: 10.1109/FCCM.2015.11. URL: https://doi.org/10.1109/FCCM.2015.11.

[186] Vaughn Betz, Jonathan Rose and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAS*. Vol. 497. The Springer International Series in Engineering and Computer Science. Kluwer, 1999. ISBN: 978-1-4613-7342-1. DOI: 10.1007/978-1-4615-5145-4. URL: https://doi.org/10.1007/978-1-4615-5145-4.

[187] Vaughn Betz and Jonathan Rose. ‘How Much Logic Should Go in an FPGA Logic Block?’ In: *IEEE Des. Test Comput.* 15.1 (1998), pp. 10–15. DOI: 10.1109/54.655177. URL: https://doi.org/10.1109/54.655177.

[188] Elias Ahmed and Jonathan Rose. ‘The effect of LUT and cluster size on deep-submicron FPGA performance and density’. In: *IEEE Trans. Very Large Scale Integr. Syst.* 12.3 (2004), pp. 288–298. DOI: 10.1109/TVLSI.2004.824300. URL: https://doi.org/10.1109/TVLSI.2004.824300.

[189] David M. Lewis et al. ‘The Stratix II logic and routing architecture’. In: *Proceedings of the ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays, FPGA 2005, Monterey, California, USA, February 20-22, 2005*. Ed. by Herman Schmit and Steven J. E. Wilton. ACM, 2005, pp. 14–20. DOI: 10.1145/1046192.1046195. URL: https://doi.org/10.1145/1046192.1046195.

[190] Altera Corp. *Stratix II Device Handbook, Volume 1 (SII5V1-4.5)*. Tech. rep. Intel, 2007.

[191] Xilinx. *DS183 (v1.28) - Virtex-7 T and XT FPGAs Data Sheet:DC and AC Switching Characteristics*. https://www.xilinx.com/support/documentation/data_sheets/ds183_Virtex_7_Data_Sheet.pdf. 2019.

[192] Xilinx Inc. *UG574 UltraScale Architecture Configurable Logic Block*. Feb. 2017. URL: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf.

[193] Andrew Boutros et al. '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 2019, Seaside, CA, USA, February 24-26, 2019*. Ed. by Kia Bazargan and Stephen Neuendorffer. ACM, 2019, pp. 94–103. DOI: 10.1145/3289602.3293912. URL: https://doi.org/10.1145/3289602.3293912.

[194] Jin Hee Kim, Jongeun Lee and Jason Anderson. 'FPGA Architecture Enhancements for Efficient BNN Implementation'. In: *International Conference on Field-Programmable Technology, FPT*. 2018, pp. 214–221. DOI: 10.1109/FPT.2018.00039. URL: https://doi.org/10.1109/FPT.2018.00039.

[195] Hadi Parandeh-Afshar, Philip Brisk and Paolo Ienne. 'Efficient synthesis of compressor trees on FPGAs'. In: *Proceedings of the 13th Asia South Pacific Design Automation Conference, ASP-DAC 2008, Seoul, Korea, January 21-24, 2008*. Ed. by Chong-Min Kyung, Kiyoung Choi and Soonhoi Ha. IEEE, 2008, pp. 138–143. DOI: 10.1109/ASPDAC.2008.4483927. URL: https://doi.org/10.1109/ASPDAC.2008.4483927.

[196] Martin Kumm and Peter Zipf. 'Efficient High Speed Compression Trees on Xilinx FPGAs'. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2014, Böblingen, Germany*. Ed. by Jürgen Ruf, Dirk Allmendinger and Matteo Michel. Cuvillier, 2014, pp. 171–182.

[197] Martin Kumm and Peter Zipf. 'Pipelined compressor tree optimization using integer linear programming'. In: *24th International Conference on Field Programmable Logic and Applications, FPL*. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927468. URL: https://doi.org/10.1109/FPL.2014.6927468.

[198] Shuang Liang et al. 'FP-BNN: Binarized neural network on FPGA'. In: *Neurocomputing* 275 (2018), pp. 1072–1086.

[199] Mohamed Eldafrawy et al. ‘FPGA Logic Block Architectures for Efficient Deep Learning Inference’. In: *ACM Trans. Reconfigurable Technol. Syst.* 13.3 (June 2020). ISSN: 1936-7406. DOI: 10.1145/3393668. URL: https://doi.org/10.1145/3393668.

[200] Xilinx. *Virtex-II Platform FPGAs: Complete Data Sheet*. 2014. URL: https://docs.xilinx.com/v/u/en-US/ds031.

[201] Mountassar Maamoun et al. ‘Efficient FPGA based architecture for high-order FIR filtering using simultaneous DSP and LUT reduced utilization’. In: *IET Circuits Devices Syst.* 15.5 (2021), pp. 475–484. DOI: 10.1049/cds2.12043. URL: https://doi.org/10.1049/cds2.12043.

[202] Xilinx Inc. *Virtex-6 FPGA DSP48E1 Slice User Guide - UG369 (v1.3)*. Tech. rep. Xilinx, 2011. URL: https://docs.xilinx.com/v/u/en-US/ug369.

[203] Xilinx Inc. *UG579: UltraScale Architecture DSP Slice*. Tech. rep. Xilinx, 2018.

[204] Intel. *SV51001 Stratix V Device Overview*. www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-v/stx5_51001.pdf. 2020.

[205] Philip Colangelo et al. ‘Exploration of Low Numeric Precision Deep Learning Inference Using Intel® FPGAs’. In: *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*. 2018, pp. 73–80. DOI: 10.1109/FCCM.2018.00020. URL: https://doi.org/10.1109/FCCM.2018.00020.

[206] Jan Sommer et al. ‘DSP-Packing: Squeezing Low-precision Arithmetic into FPGA DSP Blocks’. In: *CoRR* abs/2203.11028 (2022). DOI: 10.48550/arXiv.2203.11028. arXiv: 2203.11028. URL: https://doi.org/10.48550/arXiv.2203.11028.

[207] Intel Corp. *Stratix-IV Device Handbook Volume 1*. Tech. rep. Intel, 2011.

[208] Hadi Parandeh-Afshar and Paolo Ienne. ‘Highly Versatile DSP Blocks for Improved FPGA Arithmetic Performance’. In: *18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2010, Charlotte, North*

Carolina, USA, 2-4 May 2010. 2010, pp. 229–236. DOI: 10.1109/FCCM.2010.42. URL: https://doi.org/10.1109/FCCM.2010.42.

[209] Andrew Boutros, Sadegh Yazdanshenas and Vaughn Betz. 'Embracing Diversity: Enhanced DSP Blocks for Low-Precision Deep Learning on FPGAs'. In: *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. 2018, pp. 35–42. DOI: 10.1109/FPL.2018.00014. URL: https://doi.org/10.1109/FPL.2018.00014.

[210] 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. 2019.

[211] Xilinx. *Versal ACAP DSP Engine, Architecture Manual, AM004 (v1.1.2)*. www.xilinx.com/support/documentation/architecture-manuals/am004-versal-dsp-engine.pdf. 2021.

[212] Brian Gaide et al. 'Xilinx Adaptive Compute Acceleration Platform: Versal^{TM} Architecture'. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*. 2019, pp. 84–93. DOI: 10.1145/3289602.3293906. URL: https://doi.org/10.1145/3289602.3293906.

[213] Eriko Nurvitadhi et al. 'In-Package Domain-Specific ASICs for Intel® Stratix® 10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC(Abstract Only)'. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, Monterey, CA, USA, February 25-27, 2018*. 2018, p. 287. DOI: 10.1145/3174243.3174966. URL: https://doi.org/10.1145/3174243.3174966.

[214] FlexLogix. *nnMAX Overview*. https://flex-logix.com/wp-content/uploads/2019/09/2019-09-nnMAX-4-page-Overview.pdf. 2019.

[215] Martin Langhammer et al. 'Stratix 10 NX Architecture'. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.4 (2022), 45:1–45:32. DOI: 10.1145/3520197. URL: https://doi.org/10.1145/3520197.

[216] Andrew Boutros et al. 'Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs'. In: *International Conference on Field-Programmable Technology, (IC)FPT 2020, Maui, HI, USA, December 9-11, 2020*. IEEE, 2020, pp. 10–19. DOI: 10.1109/ICFPT51103.2020.00011. URL: https://doi.org/10.1109/ICFPT51103.2020.00011.

[217] Achronix - Data Acceleration. *Speedster7t IP Component Library User Guide (UG086)*. www.achronix.com/sites/default/files/docs/Speedster7t_IP_Component_Library_User_Guide_UG086.pdf. 2019.

[218] Aman Aror, Zhigang Wei† and Lizy K. John. 'Hamamu: Specializing FPGAs for ML Applications by Adding Hard Matrix Multiplier Blocks'. In: *31th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP*. 2020. URL: http://lca.ece.utexas.edu/pubs/Hamamu___ASAP2020_Jun9.pdf.

[219] Aman Arora et al. 'Tensor Slices to the Rescue: Supercharging ML Acceleration on FPGAs'. In: *FPGA '21: The 2021 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, Virtual Event, USA, February 28 - March 2, 2021*. Ed. by Lesley Shannon and Michael Adler. ACM, 2021, pp. 23–33. DOI: 10.1145/3431920.3439282. URL: https://doi.org/10.1145/3431920.3439282.

[220] Milos D Ercegovac and Tomas Lang. *Digital arithmetic*. Elsevier, 2004.

[221] Angelo Raffaele Meo. 'Arithmetic Networks and their minimization using a line of elementary units'. In: *IEEE Transactions on Computers* 100.3 (1975), pp. 258–280.

[222] Thomas B. Preußer. 'Generic and universal parallel matrix summation with a flexible compression goal for Xilinx FPGAs'. In: *27th International Conference on Field Programmable Logic and Applications, FPL*. 2017, pp. 1–7. DOI: 10.23919/FPL.2017.8056834. URL: https://doi.org/10.23919/FPL.2017.8056834.

[223] Martin Kumm and Johannes Kappauf. 'Advanced compressor tree synthesis for FPGAs'. In: *IEEE Transactions on Computers* 67.8 (2018), pp. 1078–1091.

[224] Hadi Parandeh-Afshar, Philip Brisk and Paolo Ienne. 'Exploiting fast carry-chains of FPGAs for designing compressor trees'. In: *19th International Conference on Field*

Programmable Logic and Applications, FPL. 2009, pp. 242–249. DOI: 10.1109/FPL.2009.5272301. URL: https://doi.org/10.1109/FPL.2009.5272301.

[225] Hadi Parandeh-Afshar et al. ‘Compressor tree synthesis on commercial high performance FPGAs’. In: *TRETS* 4.4 (2011), 39:1–39:19. DOI: 10.1145/2068716.2068725. URL: https://doi.org/10.1145/2068716.2068725.

[226] J. G. Earle. ‘Latched carry-save adder’. In: *IBM Tech. Disc. Bull.* 7.10 (1965), pp. 909–910.

[227] Hadi Parandeh-Afshar et al. ‘Improving FPGA Performance for Carry-Save Arithmetic’. In: *IEEE Trans. Very Large Scale Integr. Syst.* 18.4 (2010), pp. 578–590. DOI: 10.1109/TVLSI.2009.2014380. URL: https://doi.org/10.1109/TVLSI.2009.2014380.

[228] Ajay K. Verma and Paolo Ienne. ‘Automatic synthesis of compressor trees: reevaluating large counters’. In: *2007 Design, Automation and Test in Europe Conference and Exposition, DATE 2007, Nice, France, April 16-20, 2007*. 2007, pp. 443–448. DOI: 10.1109/DATE.2007.364632. URL: https://doi.org/10.1109/DATE.2007.364632.

[229] Lei Shan et al. ‘A Dynamic Multi-precision Fixed-Point Data Quantization Strategy for Convolutional Neural Network’. In: *Computer Engineering and Technology - 20th CCF Conference, NCCET 2016, Xi’an, China, August 10-12, 2016, Revised Selected Papers*. 2016, pp. 102–111. DOI: 10.1007/978-981-10-3159-5_10. URL: https://doi.org/10.1007/978-981-10-3159-5_10.

[230] Yuan Dai et al. ‘APIR-DSP: An approximate PIR-DSP architecture for error-tolerant applications’. In: *International Conference on Field-Programmable Technology, ICFPT*. IEEE, 2021, pp. 1–8. DOI: 10.1109/ICFPT52863.2021.9609927. URL: https://doi.org/10.1109/ICFPT52863.2021.9609927.

[231] Magnus Själander and Per Larsson-Edefors. ‘Multiplication Acceleration Through Twin Precision’. In: *IEEE Trans. VLSI Syst.* 17.9 (2009), pp. 1233–1246. DOI: 10.1109/TVLSI.2008.2002107. URL: https://doi.org/10.1109/TVLSI.2008.2002107.

[232] Charles R. Baugh and Bruce A. Wooley. ‘A Two’s Complement Parallel Array Multiplication Algorithm’. In: *IEEE Trans. Computers* 22.12 (1973), pp. 1045–1047. DOI: 10.1109/T-C.1973.223648. URL: https://doi.org/10.1109/T-C.1973.223648.

[233] Xilinx. *DS202 (v5.5) Virtex-5 FPGA Data Sheet:DC and Switching Characteristics*. June 2016. URL: https://www.xilinx.com/support/documentation/data_sheets/ds202.pdf.

[234] Henry Wong, Vaughn Betz and Jonathan Rose. ‘Quantifying the Gap Between FPGA and Custom CMOS to Aid Microarchitectural Design’. In: *IEEE Trans. VLSI Syst.* 22.10 (2014), pp. 2067–2080. DOI: 10.1109/TVLSI.2013.2284281. URL: https://doi.org/10.1109/TVLSI.2013.2284281.

[235] S. Hsu et al. ‘An 8.8GHz 198mW 16x64b 1R/1W variation tolerant register file in 65nm CMOS’. In: *2006 IEEE International Solid State Circuits Conference - Digest of Technical Papers*. Feb. 2006, pp. 1785–1797. DOI: 10.1109/ISSCC.2006.1696235.

[236] Khawar Sarfraz and Mansun Chan. ‘A 65nm 3.2 GHz 44.2 mW Low-V t register file with robust low-capacitance dynamic local bitlines’. In: *European Solid-State Circuits Conference (ESSCIRC), ESSCIRC 2015-41st*. IEEE. 2015, pp. 331–334. DOI: 10.1109/ESSCIRC.2015.7313894.

[237] Pritam Bhattacharjee and Alak Majumder. ‘A Variation-Aware Robust Gated Flip-Flop for Power-Constrained FSM Application’. In: *J. Circuits Syst. Comput.* 28.7 (2019), 1950108:1–1950108:26. DOI: 10.1142/S0218126619501081. URL: https://doi.org/10.1142/S0218126619501081.

[238] Ji-Zhong Shen et al. ‘Low-power level converting flip-flop with a conditional clock technique in dual supply systems’. In: *Microelectronics Journal* 45.7 (2014), pp. 857–863. DOI: 10.1016/j.mejo.2014.04.035. URL: https://doi.org/10.1016/j.mejo.2014.04.035.

[239] Subhasis Das, Tor M. Aamodt and William J. Dally. ‘SLIP: reducing wire energy in the memory hierarchy’. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*. 2015, pp. 349–361.

DOI: 10.1145/2749469.2750398. URL: https://doi.org/10.1145/2749469.2750398.

[240] Aaron Stillmaker and Bevan M. Baas. 'Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm'. In: *Integration* 58 (2017), pp. 74–81. DOI: 10.1016/j.vlsi.2017.02.002. URL: https://doi.org/10.1016/j.vlsi.2017.02.002.

[241] Utku Aydonat et al. 'An OpenCL(TM) Deep Learning Accelerator on Arria 10'. In: *CoRR* abs/1701.03534 (2017). arXiv: 1701.03534. URL: http://arxiv.org/abs/1701.03534.

[242] Michaela Blott et al. 'FINN-*R*: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks'. In: *TRETS* 11.3 (2018), 16:1–16:23. DOI: 10.1145/3242897. URL: https://doi.org/10.1145/3242897.

[243] Vojin G. Oklobdzija, David Villeger and Simon S. Liu. 'A Method for Speed Optimized Partial Product Reduction and Generation of Fast Parallel Multipliers Using an Algorithmic Approach'. In: *IEEE Trans. Computers* 45.3 (1996), pp. 294–306. DOI: 10.1109/12.485568. URL: https://doi.org/10.1109/12.485568.

[244] Paul F. Stelling et al. 'Optimal Circuits for Parallel Multipliers'. In: *IEEE Trans. Computers* 47.3 (1998), pp. 273–285. DOI: 10.1109/12.660163. URL: https://doi.org/10.1109/12.660163.

[245] IBM ILOG CPLEX. 'V12. 1: User's Manual for CPLEX'. In: *International Business Machines Corporation* 46.53 (2009), p. 157.

[246] Stuart Mitchell, Stuart Mitchell Consulting and Iain Dunning. *PuLP: A Linear Programming Toolkit for Python*. 2011.

[247] Itay Hubara et al. 'Binarized Neural Networks'. In: *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems*. 2016, pp. 4107–4115. URL: http://papers.nips.cc/paper/6573-binarized-neural-networks.

[248] Xilinx. *UG474 7 Series FPGAs Configurable Logic Block*. 2016. URL: www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf.

[249] Zhourui Song, Zhenyu Liu and Dongsheng Wang. 'Computation Error Analysis of Block Floating Point Arithmetic Oriented Convolution Neural Network Accelerator Design'. In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by Sheila A. McIlraith and Kilian Q. Weinberger. AAAI Press, 2018, pp. 816–823. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16057.

[250] Itay Hubara et al. 'Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations'. In: *J. Mach. Learn. Res.* 18 (2017), 187:1–187:30. URL: http://jmlr.org/papers/v18/16-456.html.

[251] Nvidia. *DS-10184-001: NVIDIA Jetson Xavier NXSystem-on-Module*. Version 1.6. 2020. URL: https://img.iceasy.com/product/product/files/202107/8a8a8a1a7a81d57a017a9eb9bb204157.pdf.

[252] Xilinx. *XMP103: Product Selection Guide*. Version 2.0. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf. 2021.

[253] Esther Roorda et al. 'FPGA Architecture Exploration for DNN Acceleration'. In: *ACM Trans. Reconfigurable Technol. Syst.* 15.3 (2022), 33:1–33:37. DOI: 10.1145/3503465. URL: https://doi.org/10.1145/3503465.

[254] Aman Arora et al. 'Koios: A Deep Learning Benchmark Suite for FPGA Architecture and CAD Research'. In: *31st International Conference on Field-Programmable Logic and Applications, FPL 2021, Dresden, Germany, August 30 - Sept. 3, 2021*. IEEE, 2021, pp. 355–362. DOI: 10.1109/FPL53798.2021.00068. URL: https://doi.org/10.1109/FPL53798.2021.00068.

[255] Chen Zhang et al. 'Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks'. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170. ISBN: 9781450333153.

DOI: 10.1145/2684746.2689060. URL: https://doi.org/10.1145/2684746.2689060.

[256] Yewei Shi et al. 'Image Recognition Based on Multi-Scale Dilated Lightweight Network Model'. In: *Proceedings of the 5th International Conference on Multimedia and Image Processing*. ICMIP '20. Nanjing, China: Association for Computing Machinery, 2020, pp. 43–48. ISBN: 9781450376648. DOI: 10.1145/3381271.3381300. URL: https://doi.org/10.1145/3381271.3381300.

[257] Wei Sun et al. 'A Lightweight Neural Network Combining Dilated Convolution and Depthwise Separable Convolution'. In: *Cloud Computing, Smart Grid and Innovative Frontiers in Telecommunications*. Ed. by Xuyun Zhang et al. Cham: Springer International Publishing, 2020, pp. 210–220. ISBN: 978-3-030-48513-9.

[258] Marco Carreras et al. 'Optimizing Temporal Convolutional Network inference on FPGA-based accelerators'. In: *CoRR* abs/2005.03775 (2020). arXiv: 2005.03775. URL: https://arxiv.org/abs/2005.03775.

[259] Baidu. *DeepBench*. https://github.com/baidu-research/DeepBench. 2016.

[260] Martin Langhammer and Bogdan Pasca. 'Floating-Point DSP Block Architecture for FPGAs'. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015*. Ed. by George A. Constantinides and Deming Chen. ACM, 2015, pp. 117–125. DOI: 10.1145/2684746.2689071. URL: https://doi.org/10.1145/2684746.2689071.

[261] Avnet. *Ultra96-V2 Single Board Computer Hardware User's, Guide Version 1.3*. 2021. URL: https://www.avnet.com/wps/wcm/connect/onesite/b85b9556-0b2a-42b3-ad6a-8dcf3eac1ff9/Ultra96-V2-HW-User-Guide-v1_3.pdf?MOD=AJPERES&CACHEID=ROOTWORKSPACE.Z18_NA5A1I41L0ICD0ABNDMDDG0000-b85b9556-0b2a-42b3-ad6a-8dcf3eac1ff9-nDNP5R3.

[262] Xilinx. *ZCU111 Evaluation Board, User Guide, UG1271 (v1.2)*. https://www.xilinx.com/support/documentation/boards_and_kits/zcu111/ug1271-zcu111-eval-bd.pdf. 2018.