Algebraic Enhancements for Systolic Arrays

ALGEBRAIC ENHANCEMENTS FOR SYSTOLIC ARRAYS

BY

TREVOR E. POGUE, B.Eng., M.A.Sc.

A THESIS

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Doctor of Philosophy (2024) McMaster University

(Electrical & Computer Engineering) Hamilton, Ontario, Canada

TITLE:	Algebraic Enhancements for Systolic Arrays
AUTHOR:	Trevor E. Pogue B.Eng. (Electrical Engineering), M.A.Sc. (Electrical & Computer Engineering), McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. Nicola Nicolici
NUMBER OF PAGES:	xvii, 168

Abstract

The field of deep learning has seen increasing breakthroughs and commercial adoption in recent years for enabling a wide range of applications including image and speech recognition, multimedia generation, information summarization, and human-like chatbots. This has led to a growing need for hardware that can quickly and efficiently perform deep learning inference, which increasingly requires massive amounts of computational power.

To address this need, recent years have seen many works for optimizing deep learning inference in hardware. Systolic arrays are an efficient class of hardware designs to use as a starting point for this application. However, after hardware-oriented deep learning model optimizations reach their limits, after the known parallelism for executing their compute patterns in hardware is exhausted, and after technology scaling slows to a halt, there is an accelerator wall that limits further improvement on the implementation side.

In this thesis, we contribute to this field through an under-explored direction by presenting new efficient matrix multiplication algorithms and/or their systolic-array hardware architectures that increase performance-per-area by reducing the workload at the algebraic level, and thus by computing the same result from a re-arranged compute pattern requiring fewer or cheaper operations to be performed in hardware. We evaluate our architectures in an end-to-end deep learning accelerator, demonstrating their ability to increase the performance-per-area of hardware accelerators beyond their normal theoretical limits.

Acknowledgements

I would like to thank and acknowledge all those who contributed to the completion of this thesis. I had a lot of fun producing this work, but it is time to move on now. First of all, I want to express my gratitude to my supervisor, Dr. Nicola Nicolici, who has always been a great mentor both professionally and personally. He provided me with a creative environment and space to think freely, which allowed me to thrive and reach my full potential. I also would like to thank my supervisory committee members, Dr. Sorina Dumitrescu and Dr. Mohamed Hassan, as well as my external examiner Dr. Zeljko Zilic, for their time reviewing my thesis and their advice which improved my work. Despite our in-person time getting cut short from the pandemic, I thank all of the colleagues I have overlapped with during my time at the Computer-Aided Design and Test Research Group at McMaster University, both former and present members. I would also like to thank my family for supporting me along this chapter of my life. Finally, I thank my wife, Michelle Pogue, for her constant support which has played a major role in my success.

Abbreviations

ML: Machine Learning.

DNN: Deep Neural Network.

CNN: Convolutional Neural Network.

GEMM: General Matrix Multiplication.

CPU: Central Processing Unit.

GPU: Graphics Processing Unit.

FPGA: Field Programmable Gate Arrays.

ASIC: Application-Specific Integrated Circuit.

TPU: Tensor Processing Unit.

MXU: Matrix Multiplication Unit.

GOPS: Giga Operations per Second.

PE: Processing Element.

DSP: Digital Signal Processing.

MAC: Multiply-Accumulate.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent years have seen increasing breakthroughs and commercial adoption of deep learning, which has enabled ground-breaking applications ranging from human-like chatbots like ChatGPT, to generating realistic image and video content from prompts, self-driving cars [1], detecting cancer [2], and beating human champions at the complex game of Go [3]. However, deep learning inference increasingly requires massive amounts of computational power to perform, making it ever-more difficult to execute quickly and efficiently. This can be mitigated by building special-purpose computer hardware that can perform deep learning inference more efficiently than general-purpose hardware like conventional central processing units (CPU)s. In this thesis, we join this effort by identifying and advancing an under-explored area in the field of deep learning hardware design.

The rest of this chapter provides more background on deep learning in Section 1.1, and Section 1.2 goes over a high-level description of hardware acceleration and why it is necessary to meet the computational demands of deep learning. Section 1.3 then identifies under-explored areas in prior work on deep learning acceleration that form the basis for our contributions. Finally, our contributions and thesis organization are outlined in Section 1.4.

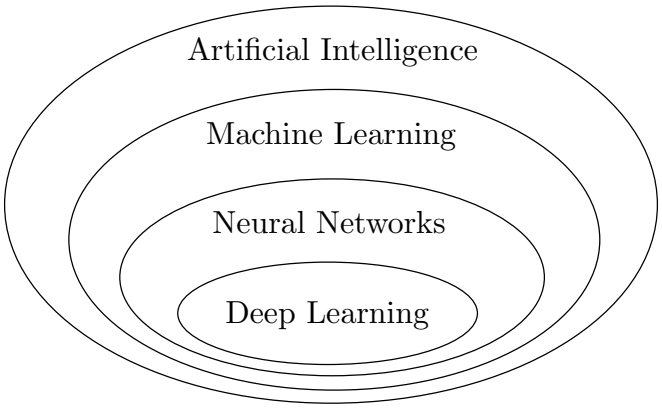

Figure 1.1: Categorization of the subfield of deep learning within the broader field of artificial intelligence.

1.1 Deep Learning

Increases in computational power and available training data in recent decades have helped initiate an awakening for the field of deep learning, allowing models and theories to be proven and leveraged in real-world applications that previously were infeasible to test. This has led to a positive feedback loop between increasing adoption of deep learning into commercial applications and further funding and research interest leading to scientific advancements.

Deep learning (DL) is a category of machine learning (ML), which is subset of the broad field of artificial intelligence (AI). AI was founded as an academic discipline in the 1950s and, at a high level, is a field of research with the goal of creating artificial systems that exhibit intelligent human-like behaviour. Machine learning is a strategy to achieve AI through systems that train and learn to achieve tasks on their own or without explicit instructions. This saves humans much of the laborious efforts of working out all of the problem details and doing all the manual programming of the system, passing some of that effort onto the system itself.

Artificial Neural networks (ANN)s, also referred to as just neural networks (NN)s were first proposed in the 1940s and are a method within the field of ML inspired by the human brain, which is currently and originally the best known general intelligence system for learning and solving problems. NNs are built around the concept of a neuron, which is the primitive computational element in the human brain, and there are approximately 86 billion neurons in the average human brain [4]. Deep learning, first proposed in the 1960s, is the study of large neural networks, also called deep neural networks, and is the primary technology behind the applications currently labelled as using ML or AI today. In Section 2.1, we provide more details on DNNs and the computational patterns they map to.

1.2 Hardware Acceleration

Due to the increasing adoption of deep learning, there is high demand for ways to efficiently perform deep learning inference on a range of computational platforms ranging from smart phones to warehouse-scale distributed computing data centers alike; all of which aim to maximize execution speed and minimize power usage. However, the number of operations required to perform deep learning inference is commonly in the billions and is only increasing in newer deep learning models, making these performance goals ever-more difficult to reach. Fortunately, the deep learning inference commonly decomposes down to performing the same subclass of compute patterns which mainly consist of performing a large number of multiplications and additions that can be performed in parallel with one another. This makes it a great fit for hardware acceleration, in which custom hardware is designed to execute a specialized subset of compute patterns in parallel more efficiently than what is possible to be performed in general-purpose hardware like CPUs that can perform a wider range of operations but operate more sequentially and slowly.

Conventional CPUs are fit for any general or unknown computational patterns, making them the central part of a general-purpose computer. The intention of more specialized hardware, however, is to be more efficient at executing a smaller range of computational patterns. Specialized hardware typically achieves this by containing only the logic necessary for its specific subset of tasks, and by performing many subtasks in parallel.

GPUs are specialized at performing simple arithmetic operations such as multiplications and additions in parallel, making GPUs an initially natural fit for deep learning acceleration. However, one limiting factor of GPUs is that it can be difficult to load all of the inputs to the multiplications and additions and return their outputs quickly enough, causing this data movement to become the performance bottleneck. Specialized hardware such as the Google Tensor Processing Unit (TPU) [3], [5], [6] allows for many multiplications and additions to be performed in parallel while also requiring less intermediate data movement, leading to faster and more efficient deep learning acceleration compared to a GPU.

Specialized hardware can be implemented using two main types of technologies, application specific integrated circuits (ASIC) or field-programmable gate-arrays (FPGA). ASICs are specialized hardware designed strictly for a specific purpose where the functionality is permanent after the hardware is shipped. In contrast, an FPGA is a device that contains programmable circuitry that can be updated to implement a specified logic circuit at any time which makes design errors more forgivable, provides future-proofing by allowing hardware designs to be updated, and it is much cheaper to purchase a small number of FPGAs than manufacturing an ASIC for mass production. On the other hand, while the functionality of ASICs is permanent, they are more optimized and efficient at performing that specific functionality than an FPGA programmed for that same functionality. In Section 2.2, we discuss more background and prior works on custom deep learning hardware.

1.3 Efficient Algebraic Algorithms

To address the increasing need for efficiently performing deep learning inference, a variety of methods for hardware designs and optimizing the deep learning inference in hardware have been explored in recent years. However, as all deep learning model and hardware implementation optimizations are becoming explored to their limits, optimizations for reducing deep learning workloads at the algebraic level remain a less travelled route for continuing progress. This involves rearranging the computation for carrying out a deep learning model's algebra such that it produces the same output but from fewer or cheaper operations performed in hardware.

For example, multiplications can require more circuitry to execute in hardware than additions. So if a deep neural network's algebra can be rearranged to produce the same output while trading half of the multiplications for additions, then the same result could be computed from a smaller and less power-consuming hardware circuit. Or the circuitry could be scaled up to execute the deep neural network faster while consuming the same amount hardware area and power as a circuit using conventional algebra. Additionally, there are under-explored efficient algebraic algorithms that can produce the same result from both fewer additions and multiplications being performed which have not been sufficiently studied for exploitation in custom hardware circuits.

This leaves opportunities to derive new contributions showing how to translate new or prior works on efficient algebraic algorithms into improvements in hardware accelerator architectures for deep learning, which is the focus of this thesis. In Section 2.3 we provide more background on relevant efficient algebraic algorithms and their hardware architectures.

1.4 Contributions and Thesis Organization

In this thesis, we study the identified under-explored area of algebraic enhancements for matrix multiplication algorithms and hardware architectures with application to deep learning acceleration. We propose several advancements to efficient algebraic algorithms and/or their systolic-array hardware architectures in Chapters 4 - 6.

Chapter 2 provides more detailed background on deep learning and the computational patterns it maps to, prior approaches for deep learning acceleration, and prior work on efficient algebraic algorithms and their implementation in hardware, which builds the foundation for presenting our contributions.

Chapter 3 outlines the deep learning accelerator system used for evaluating the architectures proposed in Chapters 4 - 6. To evaluate each of our contributions, different proposed matrix multiplication architectures are swapped for the baseline matrix multiplication unit (MXU) in the system design, while the remaining components in the accelerator system remain largely unchanged for each method.

Chapter 4 presents an algorithm called the free-pipeline inner product (FFIP) and general hardware architecture that improve Winograd's under-explored inner-product algorithm called the fast inner product (FIP) [7] that can be seamlessly incorporated into deep learning accelerators to significantly increase the accelerator's performance-per-area. We implement and evaluate FIP for the first time in a deep learning accelerator system described in Chapter 3. We then identify a weakness of FIP and propose the new FFIP algorithm and generalized hardware architecture that inherently address that weakness. We provide deep learning-specific optimizations for the FIP and FFIP algorithms and hardware architectures. We derive how the (F)FIP architectures increase the theoretical compute efficiency and performance limits in the general case. The contributions from this chapter

have been published in [8].

Chapter 5 proposes an algorithm and its hardware architectures that extend the Karatsuba algorithm [9] to matrix multiplication. While the Karatsuba algorithm reduces the complexity of large integer multiplication, the extra additions required minimize its benefits for smaller integers of more commonly-used bitwidths. In this chapter, we propose the extension of the scalar Karatsuba multiplication algorithm to matrix multiplication, showing how this maintains the reduction in multiplication complexity of the original Karatsuba algorithm while reducing the complexity of the extra additions. Furthermore, we propose new matrix multiplication hardware architectures for efficiently exploiting this extension of the Karatsuba algorithm in custom hardware. We show that the proposed algorithm and hardware architectures can provide real area or execution time improvements for integer matrix multiplication compared to scalar Karatsuba or conventional matrix multiplication algorithms, while also supporting implementation through proven systolic array and conventional multiplier architectures at the core. We provide a complexity analysis of the algorithm and architectures and evaluate the proposed designs both in isolation and in an end-to-end deep learning accelerator system described in Chapter 3 compared to baseline designs and prior state-of-the-art works implemented on the same type of compute platform, demonstrating their ability to increase the performance-per-area of matrix multiplication hardware.

Chapter 6 explores hardware architectures for exploiting Strassen's fast matrix multiplication algorithm. While Strassen's matrix multiplication algorithm reduces the complexity of naive matrix multiplication, general-purpose hardware is not suitable for achieving the algorithm's promised theoretical speedups, leaving the question of if it could be better exploited in custom hardware architectures designed specifically for executing the algorithm.

However, there is limited prior work on this and it is not immediately clear how to derive such architectures or if they can ultimately lead to real improvements. We bridge this gap, presenting and evaluating new systolic-array architectures that efficiently translate the theoretical complexity reductions of Strassen's algorithm directly into hardware resource savings. Furthermore, the architectures are multi-systolic-array designs that can multiply smaller matrices with higher utilization than single-systolic-array designs. The proposed design implemented on FPGA for multiplying matrix sizes down to 24×24 at 2 levels of Strassen recursion uses approximately 10% fewer soft logic resources and 1.3× fewer DSP units than a conventional multi-systolic-array design. We evaluate the proposed Strassen systolic arrays in isolation as well as in an end-to-end deep learning accelerator system described in Chapter 3 compared to baseline designs and prior works implemented on the same type of compute platform, demonstrating their ability to increase compute efficiency and achieve state-of-the-art performance. Finally, Chapter 7 provides a summary of the contributions and results, possible directions for future work, and concluding remarks.

Chapter 2

Background and Prior Work

In this chapter, we provide more background on deep learning and how it maps to matrix multiplication in Section 2.1, followed by a literature review of prior work and approaches for deep learning hardware acceleration in Section 2.2. Section 2.3 then provides background on an under-explored avenue for continuing progress in the field of deep learning hardware acceleration, which is advancement and application of efficient matrix multiplication algorithms to deep learning hardware architectures.

2.1 Deep Learning and Matrix Multiplication

Neural networks are built around the concept of a neuron, which is the primitive computational element in the human brain. Neural networks contain interconnected layers of neurons, where each neuron outputs some function of its inputs like shown in Fig. 2.1, and passes its result called an *activation* as input to neurons in other layers. The functions that the neurons perform on their inputs contain constants or parameters called *weights* that determine properties such as the scaling factor to scale each of its input values by.

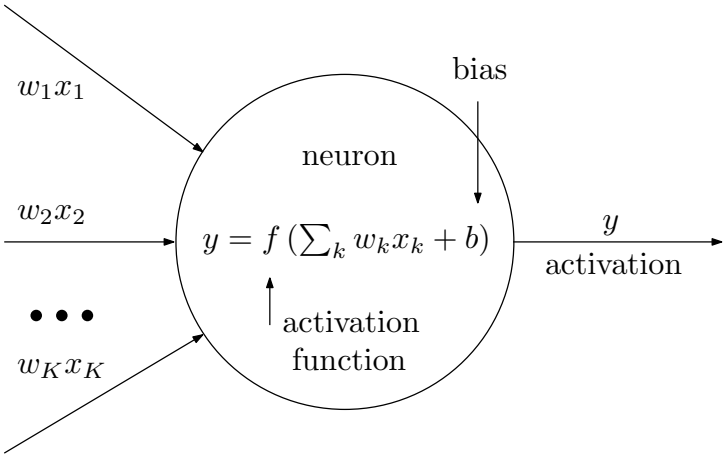

Figure 2.1: A neural network neuron in a fully-connected layer.

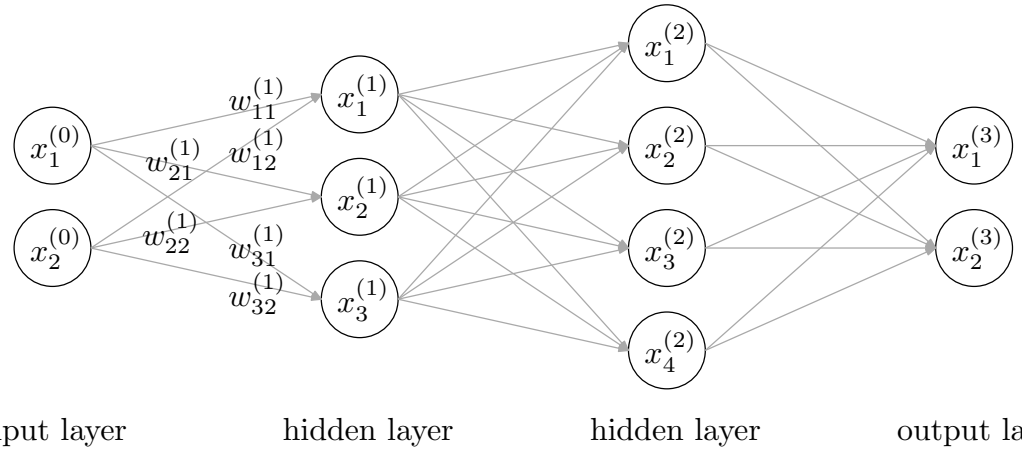

Figure 2.2: A deep neural network with an input layer, two hidden layers, and output layer.

A deep neural network (DNN) is a neural network consisting of four or more layers such that it contains two or more middle layers (called *hidden layers*) between the input and output layers such as the example shown in Fig. 2.2. While there are many variations of deep neural network model architectures, the computationally dominant portion of many common DNN models used today are based on several types of neural network layers discussed next that can all be mainly decomposed to matrix multiplication.

2.1.1 Fully-Connected Layers

Fully-connected layers are a common building block included in many types of DNN models including convolutional neural networks (CNN)s and Transformer models that are discussed next. The hidden layers in Fig. 2.2 are an example of a fully-connected layer. Each output activation in a fully-connected layer is a weighted sum of all output activations from the previous layer, followed by the addition of a *bias* value, and finally the application of a non-linear activation function such as a sigmoid, hyperbolic tangent, or rectified linear unit (ReLU) function [10].

This computation translates to a matrix-vector multiplication-based computation between the weights and the previous layer's activations. For illustration, the leftmost hidden layer in Fig. 2.2 is a fully-connected layer that performs the following operation:

$$\begin{bmatrix} y_1^{(1)} \\ y_2^{(1)} \\ y_3^{(1)} \end{bmatrix} = \begin{bmatrix} b^{(1)} & w_{11}^{(1)} & w_{12}^{(1)} \\ b^{(1)} & w_{21}^{(1)} & w_{22}^{(1)} \\ b^{(1)} & w_{31}^{(1)} & w_{32}^{(1)} \end{bmatrix} \begin{bmatrix} 1 \\ x_1^{(0)} \\ x_2^{(0)} \end{bmatrix}, \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \\ x_3^{(1)} \end{bmatrix} = \begin{bmatrix} f(y_1^{(1)}) \\ f(y_2^{(1)}) \\ f(y_3^{(1)}) \end{bmatrix}. \tag{2.1}$$

Multiple inferences can also be performed consecutively in batches when passing through neural network layers. Performing inference on a batch of inputs like this then translates

to a matrix-matrix multiplication-based computation. For example, this would translate to the following operation in the fully-connected layer contained in the leftmost hidden layer in Fig. 2.2:

$$\begin{bmatrix} y_{11}^{(1)} & y_{12}^{(1)} & y_{13}^{(1)} \\ y_{21}^{(1)} & y_{22}^{(1)} & y_{23}^{(1)} \\ y_{31}^{(1)} & y_{32}^{(1)} & y_{33}^{(1)} \end{bmatrix} = \begin{bmatrix} b^{(1)} & w_{11}^{(1)} & w_{12}^{(1)} \\ b^{(1)} & w_{21}^{(1)} & w_{22}^{(1)} \\ b^{(1)} & w_{31}^{(1)} & w_{32}^{(1)} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ x_{11}^{(0)} & x_{12}^{(0)} & x_{13}^{(0)} \\ x_{21}^{(0)} & x_{22}^{(0)} & x_{23}^{(0)} \end{bmatrix} \tag{2.2}$$

$$\begin{bmatrix} x_{11}^{(1)} & x_{12}^{(1)} & x_{13}^{(1)} \\ x_{21}^{(1)} & x_{22}^{(1)} & x_{23}^{(1)} \\ x_{31}^{(1)} & x_{32}^{(1)} & x_{33}^{(1)} \end{bmatrix} = \begin{bmatrix} f(y_{11}^{(1)}) & f(y_{12}^{(1)}) & f(y_{13}^{(1)}) \\ f(y_{21}^{(1)}) & f(y_{22}^{(1)}) & f(y_{23}^{(1)}) \\ f(y_{31}^{(1)}) & f(y_{32}^{(1)}) & f(y_{33}^{(1)}) \end{bmatrix} . \tag{2.3}$$

This can then be written as:

$$\mathbf{X^{(1)}} = f(\mathbf{W^{(1)}X^{(0)}} + b^{(1)}) . \tag{2.4}$$

Letting K and M respectively represent the number of layer inputs and outputs and N be the number of inferences being performed in a batch (the *batch size*), $\mathbf{W^{(1)}}$ is then a matrix of size $M \times K$ and $\mathbf{X}$ is a matrix of size $K \times N$. The majority of operations in a fully-connected layer performing inference in batches is then contained in the matrix-matrix multiplication $\mathbf{W^{(1)}X}$ above, as it requires $\mathcal{O}(MNK)$ operations, which dominates the complexity of the bias addition and activation function which require $\mathcal{O}(MN)$ operations.

In this thesis, we instead refer to the matrix multiplications $\mathbf{W^{(d)}X^{(d-1)}}$ in these layers in the following format:

$$\mathbf{C} = \mathbf{AB} , \tag{2.5}$$

where $\mathbf{C} = (\mathbf{W^{(d)}X^{(d-1)}})^T$, $\mathbf{A} = \mathbf{X^{(d-1)}}^T$, and $\mathbf{B} = \mathbf{W^{(d)}}^T$.

2.1.2 Convolutional Layers

Convolutional neural networks (CNN)s are a common type of DNN that played a major role in the re-awakening of the field of deep learning in the early 2010's when AlexNet [11] enabled a sudden boost in the best achievable error rates for image classification tasks. They contain convolutional layers that typically perform 2-D convolutions between activations and weights.

The computationally dominant portion of CNNs are the convolutional layers. The input and output activations of a convolutional layer are 2-D and are referred to as *feature maps* or *channels*. For example, the input image to a CNN will contain 3 channels/feature maps to represent the red, green, and blue values of each pixel, and each feature map will have a height/width corresponding to the pixel height/width of the image. Each layer's set of input/output feature maps can be thought of as a single 3-D input/output.

Each 2-D input feature map is convolved with a different 2-D convolution filter called a *kernel*, and the convolution outputs for all input feature maps are summed together to produce an output feature map. The set of kernels that are used for each of the different input feature maps can then be thought of as a single 3-D kernel. This computation is then repeated with multiple distinct 3-D kernels to produce multiple output feature maps. Due to this, each layer's kernel can be thought of as a 4-D tensor. Finally, multiple inferences can be performed consecutively on a batch of multiple sets of features maps.

Each layer input can then be thought of as a 4-D tensor, letting C_{in}/C_{out} represent a layer's number of input/output channels or feature maps, H/W represent the height/width of a layer's output feature maps, N is the batch size, and n is the current inference being

performed in the batch. Each layer's kernel is then a 4-D tensor with dimensions of size C_{out}, C_{in}, H_k, W_k, where H_k/W_k are the height/width of the kernel. A dimension's stride, which is explained below, is represented by the dimension with a $_s$ subscript (e.g. W_s is the stride for the W dimension).

A convolutional layer's calculation for each output element is then described by the following, where $\mathbf{X^{(d)}}$ is the layer output, $\mathbf{X^{(d-1)}}$ is the layer input, and $\mathbf{W^{(d)}}$ is the layer's weight/kernel:

$$\mathbf{X^{(d)}}[n][c_{out}][h][w] = \sum_{c_{in}=0}^{C_{in}-1} \sum_{w_k=0}^{W_k-1} \sum_{h_k=0}^{H_k-1} \mathbf{X^{(d-1)}}[n][c_{in}][hH_s + h_k][wW_s + w_k] \; \mathbf{W^{(d)}}[c_{out}][c_{in}][h_k][w_k] \, . \tag{2.6}$$

For illustration, Fig. 2.3 shows a visual representation of an example convolutional layer performing (2.6).

To understand how convolutional layers' computation in (2.6) can be mapped to matrix multiplication, consider how a 1-D convolution can be mapped to a vector-matrix multiplication as follows:

$$\begin{bmatrix} a_0 & a_1 & a_2 \end{bmatrix} * \begin{bmatrix} b_0 & b_1 & b_2 \end{bmatrix} = \begin{bmatrix} b_0 & 0 & 0 \\ b_1 & b_0 & 0 \\ b_2 & b_1 & b_0 \\ 0 & b_2 & b_1 \\ 0 & 0 & b_2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} b_0 a_0 \\ b_1 a_0 + b_0 a_1 \\ b_2 a_0 + b_1 a_1 + b_0 a_2 \\ b_2 a_1 + b_1 a_2 \\ b_2 a_2 \end{bmatrix} . \tag{2.7}$$

Similarly, a convolutional layer's computation in (2.6) can be mapped to a matrix-matrix multiplication $\boldsymbol{C} = \boldsymbol{AB}$ [12], where each row of the $\boldsymbol{A}$ matrix is formed from

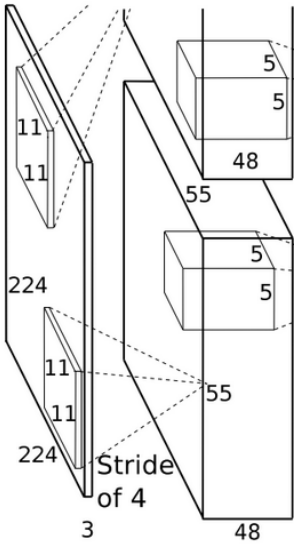

Figure 2.3: Example of a convolutional layer in Alexnet [11]. The bottom portion of the figure shows a convolutional layer where $N = 1$, $C_{in} = 3$, $C_{out} = 48$, $H = W = 55$, $H_k = W_k = 11$, and $H_s = W_s = 4$. The input feature maps have a width of 224 (before padding), and the next convolutional layer that is partially shown has a kernel size of $H_k = W_k = 11$.

the $\mathbf{X^{(d-1)}}$ elements by iterating through the indices in (2.6) as described below:

$$\mathbf{A}[i][j] = \mathbf{X^{(d-1)}}[N][c_{in}][hH_s + h_k][wW_s + w_k]\,, \tag{2.8}$$

where:

$$i = nC_{in}HW + hW + w \tag{2.9}$$

$$j = c_{in}H_kW_k + h_kW_k + w_k\,. \tag{2.10}$$

Each column j of the $\mathbf{B}$ matrix is formed from $\mathbf{W^{(d)}}$ by collapsing the C_{in}, H_k, W_k

dimensions of $W[c_{out}]$ as described below:

$$\mathbf{B}[i][j] = \mathbf{W^{(d)}}[c_{out}][c_{in}][h_k][w_k]\,. \tag{2.11}$$

where:

$$i = c_{in}H_kW_k + h_kW_k + w_k \tag{2.12}$$

$$j = c_{out}\,. \tag{2.13}$$

2.1.3 Transformers and Attention Layers

Transformer models, introduced in 2017 [13], are a more recent deep learning model that has since been shown to be superior in quality in many popular benchmarks compared to prior deep learning models like CNNs, and it is the base model used in popular works such as BERT [14] and GPT [15] models. The computationally intensive portion of transformer models are based around the following operations, which mainly consist of a sequence of large matrix multiplications as shown below:

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}(\frac{\mathbf{QK^T}}{\sqrt{d_k}})\mathbf{V} \tag{2.14}$$

$$\mathbf{H_i} = \text{Attention}(\mathbf{QW_i^Q},\mathbf{KW_i^K},\mathbf{VW_i^V}) \tag{2.15}$$

$$\text{MultiHead}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{Concat}(\mathbf{H_1},...,\mathbf{H_h})\mathbf{W^O}\,, \tag{2.16}$$

where all bolded variables are matrices, and all matrices and variables are either derived from layer inputs or are a form of pretrained weight values or constants.

While the methods presented in our work may be well suited for accelerating these

types of deep learning models, further descriptions of transformer models and exploration of their acceleration are considered out of the scope of this thesis and left as a future work.

2.1.4 Commonality

As shown in this section, the computationally most intensive portions of many common DNN models used today are based around the few types of neural network layers discussed above which can all mainly reduce to matrix multiplication. Therefore, increasing the efficiency of general matrix multiplication (GEMM) is a key area of interest to focus on for advancing the field of deep learning hardware acceleration.

2.2 Deep Learning Hardware Acceleration

This section provides a literature review of a range of prior works and common strategies used to improve the acceleration of deep learning workloads in custom hardware. It is impossible to evaluate all possible approaches and works due to the popular and fast-paced nature of this field. Nonetheless, we review a representative range of strategies and works relevant to the contributions in this thesis.

2.2.1 Systolic Arrays

Systolic arrays, which will also be referred to as matrix multiplication units (MXU)s for convenience, are an effective choice for use in GEMM accelerators as they significantly reduce the required memory traffic and can reach high clock frequencies due to their short and regular interconnects. Systolic-array architectures have been used in state-of-the-art GEMM and deep learning accelerators such as the Tensor Processing Unit (TPU) [3], [5],

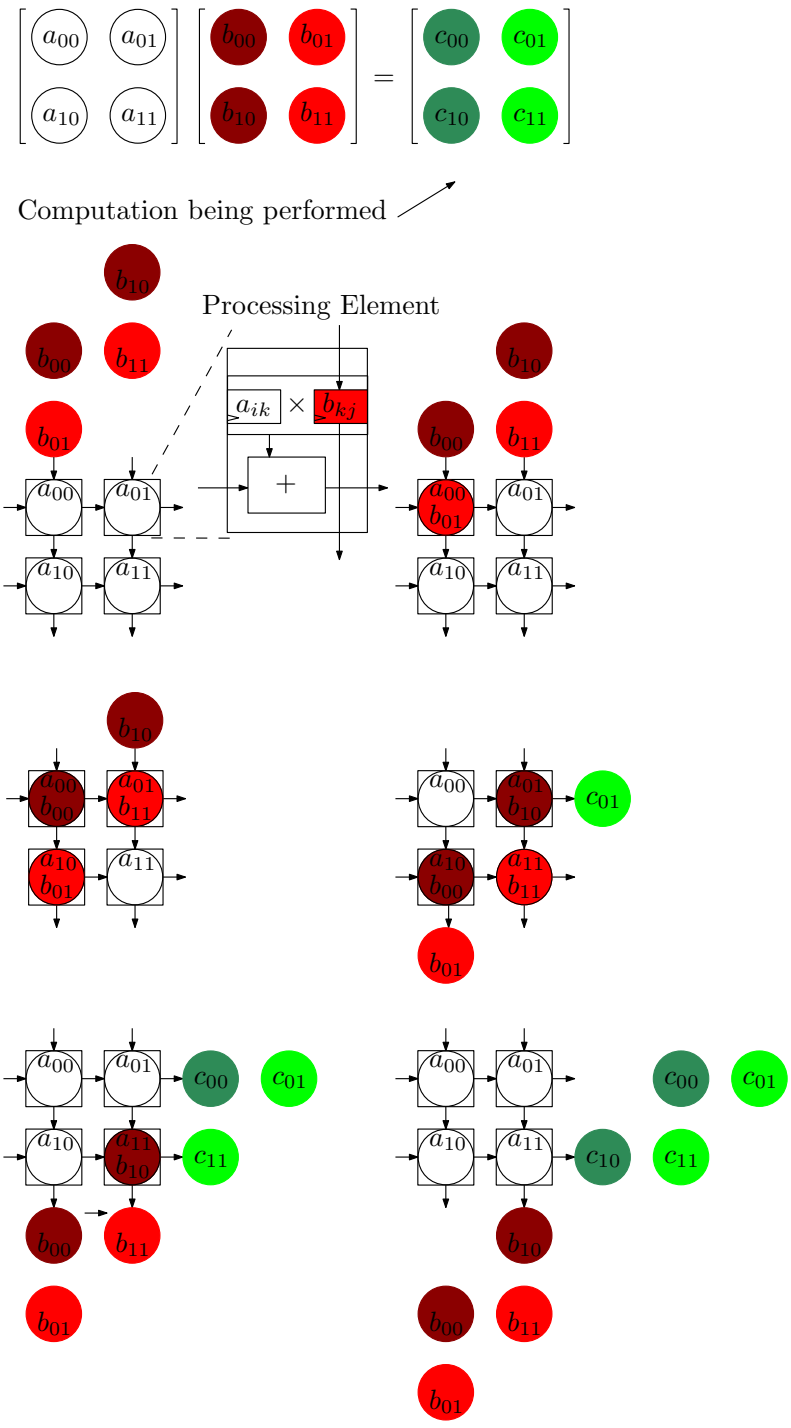

Figure 2.4: Demonstrating how matrix multiplication is performed on a systolic array of processing elements.

[6], among others [8], [16].

A systolic array consists of a 2-D array of simple connected processors. Each processor typically consists of only a multiplier and adder unit, and the processors pass data to adjacent processors in a consistent pattern, demonstrated in Figure 2.4. This makes the wire connections between processors very short and regular and increases the clock frequency of the circuit. In contrast, the processor arrays in GPUs can contain multiple longer and more complex physical connections between the various components and a central control unit.

This makes systolic arrays very efficient for deep learning acceleration. For example, while an 8-core CPU might be able to perform up to 8 multiplications or additions per clock cycle, systolic array accelerators such as the Google TPU will perform up to 64K multiplication and additions per clock cycle.

Multi-Systolic-Array Systems

However, a systolic array can only be fully utilized when the input matrix sizes at minimum match the dimensions of the systolic array or are larger, and real workloads have limits to the matrix sizes being multiplied. For example, the average matrix size that the computations reduce to in ResNet [17] DNN models are in the range of approximately 300×2000. There is then a limit to how fast the workload can be accelerated on a single systolic array design because, even if more compute resources are instantiated to scale up the size of the systolic array, the systolic array will begin to be underutilized after its size surpasses the workload's matrix sizes, and the workload will not be able to execute any faster.

This is particularly true in modern workloads such as DNN acceleration, where the matrix sizes that the workloads break down to are smaller than the maximum systolic array

size that could be instantiated in an accelerator. To combat this, multiple smaller systolic arrays can be used in parallel [5], [6], [18], [19], which allows for the total compute power in the systolic-array system to increase while the minimum supported matrix sizes remain the same.

2.2.2 Quantization

The weight and layer inputs to a DNN can be scaled and given an offset such that they are represented on integer values [20]. The bitwidth of the representation in practice can range from 16 bits [21] to an extreme of 1 bit [22] [23] [24]. Following this quantization, GEMM can be performed directly with the integer values, and a re-scaling and new offset are given between each layer.

The benefit of using small-width integers instead of the conventional 32-bit floating-point (FP32) representation is very significant for hardware acceleration. The values are represented by fewer bits, which frees up space in memory and also reduces the memory bandwidth required to read/write the values from/to memory. Additionally, small-width integer arithmetic units consume far fewer hardware resources than their FP32 counterparts [25]. Table 3 from the work by Guo et al. [25] shows that an 8-bit adder or multiplier consumes roughly 10x fewer resources than a FP32 multiplier or adder on a Xilinx FPGA.

For 1-bit quantization, the XNOR operation is performed on the data instead of multiplication, followed by a bit count operation. These operations can be efficiently implemented using look-up tables (LUT)s which are one of the base logic elements used in FPGAs to allow for their re-programmable logic. Taking this concept further, networks with LUT-based neurons [26] have been explored. Modern FPGAs usually contain LUTS that can be programmed to perform any desired function which takes 6-7 binary inputs and produces 1-2

binary outputs. On the other hand, an XNOR operation takes 2 binary inputs and produces 1 binary output. This means that the number of XNOR operations that can be implemented per LUT will be constrained by the number of LUT outputs, and each LUT can then only implement at most 2 XNOR operations, which would consume at most 4 inputs, leaving 2-3 inputs unused. Therefore, a LUT-based network could use more accurate larger-than-1-bit inputs to make use of these unused LUT inputs, while consuming the same number of LUTs as a binary network using 1-bit inputs and XNOR operations.

The study from Nurvitadhi et al. [27] tested FPGA accelerators designed to support 8-bit and binary (1-bit) data and compared it to an accelerator designed for FP32 data. The 8-bit and binary designs had roughly an 8x and 75x increase in peak performance per watt, respectively, compared to the FP32 design. It has been shown that negligible error is introduced for as low as 8-bit representations [25]. For bitwidths below this, however, noticeable increases in error start to be introduced.

2.2.3 Precision-Scalable Architectures

Precision-scalable architectures allow for a way to efficiently execute workloads across multiple input precisions for applications where the input bitwidths are expected to vary [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41]. Deep learning acceleration is a great use-case for precision-scalable hardware architectures, where neural networks can perform the majority of their inference on reduced-bitwidth operations with little to no loss in accuracy but the bitwidths required to provide sufficient accuracy vary across different deep neural network models, applications, and between individual layers within the same neural network model [28]. For example, some neural network models can be executed with high accuracy even when performing the majority of the operations

on small bitwidths, however, a smaller portion of the layers still need to be computed on larger bitwidths to preserve accuracy [28]. Therefore, a fixed-bitwidth accelerator must make a trade-off between either supporting only lower bitwidths while reducing the model's accuracy, or supporting larger bitwidths for higher accuracy but under-utilizing the MAC units during majority of computation as most stages require only lower-bit inputs.

Precision-scalable architectures address this trade-off by providing structures that can more efficiently support execution of varying input bitwidths. These designs are less optimized for accelerating inputs of a fixed bitwidth compared to a design made specifically for that certain bitwidth, however, they have the flexibility to provide more efficient acceleration when executing on other bitwidths. One approach is to use MAC units consisting of multiple smaller-bitwidth multipliers [29] which can either be individually used to multiply/accumulate multiple smaller-bitwidth products, or they can be reconfigured to collectively multiply/accumulate fewer larger-bitwidth products per clock cycle. Another approach referred to as bit-serial architectures [30], is to have MAC arrays which repeatedly perform fixed-size smaller-bitwidth vector operations on different bit slices of the vectors, summing up the separate vector products to get the final full-bitwidth result.

2.2.4 Pruning and Sparsity

Pruning is a technique in which a neural network is intentionally trained in such a way that many of its weights become zero-valued. For example, it has been shown that AlexNet can be pruned so that 85% of the weights in the convolutional layers are removed with less than 1% degradation in accuracy [42]. This provides potential speedup opportunities for accelerators as the calculations then decompose to sparse GEMM, and all of the multiplications with zero values can be skipped.

In the works by Parashar et al. [43] and Zhang et al. [44], compression schemes for weights and intermediate data were developed such that they are stored in a compressed form and require fewer data transfers and fewer storage requirements throughout computation. In the study from Nurvitadhi et al. [27], an FPGA accelerator design for sparse networks gave a 4x speedup on FP32 data compared to their accelerator for dense FP32 models.

Criticisms of this technique are that creating a hardware design capable of scheduling zero skipping prevents the design from reaching the same high clock frequency as a systolic array architecture suitable only for dense GEMM [27]. For example, the sparse GEMM accelerator design from Nurvitadhi et al. [27] ran at a frequency of 300 MHz vs 440 MHz for the dense GEMM systolic array accelerator. Furthermore, quantizing pruned networks to integer widths under 12 bits can introduce large accuracy degradation [45], preventing the sparse GEMM engines from taking advantage of the many benefits of using small-width integer arithmetic 2.2.2.

2.2.5 Memory Optimizations

While GEMM requires a large amount of computational power, it also requires large amounts of data to be moved and accessed by the computational logic [46]. Additionally, off-chip memory accesses can have higher energy costs than other operations [47]. Due to this, memory bandwidth, especially for off-chip memory access, can be a performance bottleneck or energy concern. To address this, prior works have proposed solutions to mitigate these issues [46] [47] by increasing data reuse and using efficient data tiling and caching techniques. Additionally, quantization [20] and sparse neural networks [42] are used to reduce memory bandwidth requirements as discussed in Sections 2.2.2 and 2.2.4. The

algebraic optimizations and hardware architectures proposed in this thesis are generally orthogonal to memory bandwidth optimizations and both techniques can be used in combination with each other.

2.2.6 Hardware Architecture Design Automation

It can be a difficult engineering feat to design a single deep learning accelerator that supports execution of a wide range of DNN models like the Google TPU [3], [5], [6]. Furthermore, it may be possible to make certain design choices that are more resource-efficient when optimizing a hardware design for one specific DNN model. Prior works have explored automated design flows which map a neural network described in a deep learning software framework like Caffe [21] or Pytorch [48], and automatically generate an FPGA accelerator design optimized for that DNN model based on the target FPGA platform [49] [50] [51] [52] [53] [54] [55].

These design space exploration methods can automatically derive architectural features such as efficient off-chip memory layouts and efficient systolic array dimensions based on the DNN layer sizes and an FPGA platform's resource limitations. Pseudocode with for loops is derived to model a layer's computational schedule and represent the order in which data is fed into the systolic array, and the loop tile sizes can be used to model the dimensions of the systolic array. In some cases, automated loop transformations are applied to the layer's pseudocode in order to find transformations for increasing data reuse, reducing off-chip bandwidth, and increasing on-chip parallelization in the automatically generated designs [53] [16].

In the work by Wang et al. [56], a flow is presented for generating an FPGA design for a CNN model based on the dimensions and weights of a model obtained from Caffe

[21]. The method chooses architecture elements obtained from a library of hand-coded Verilog to maintain more low-level optimizations than other techniques. Additionally, unlike other methods, the analysis works for highly irregular network topologies such as in GoogleLeNet [57] and ResNet [17] DNN models.

The loop transformation methods mentioned above use more trivial applications of an otherwise more robust compiler optimization theory called polyhedral compilation [58] [59]. Polyhedral compilation has been included in many compiler libraries including GCC [60] and LLVM [61]. In polyhedral compilation, all iterations of a loop nest are represented as different integer points in a space, all of which form a polytope that can be transformed as a whole, rather than manipulating individual iterations. The dependencies between different iterations are also modeled in this manner. Affine transformations on the polytopes can be found that produce execution schedules that reduce the average dependency distance between iterations to improve data reuse/locality. These techniques have also been used in DNN accelerator design automation [16] [55].

The techniques discussed in this subsection provide the opportunity for an additional level of optimization in an accelerator design by producing hardware designs optimized for specific DNN models and FPGA platforms. On the other hand, this prevents the designs from being suitable for ASIC implementation, which is inherently more optimized than FPGA designs. It also complicates usage scenarios where more than one single DNN model may be required to run on the accelerator, which may be a more practical real-life usage scenario. Furthermore, it has been shown by works such as Google's TPU design [3], [5], [6] that it is possible to make one hardware design that is highly efficient but still supports the execution of a wide range of DNN models.

2.2.7 Hardware-Oriented DNN Model Design Automation

The architecture of a DNN model can be designed from the ground up in such a way that it can be executed more effectively on a specific hardware platform. The DNN model can be designed to support a given throughput and latency constraint based on given hardware budget or available hardware resources on a given hardware platform such as the on/off-chip memory size, total registers, and/or total DSP units on the device. The design space dimensions can include the types of layers to use, size of the layers, as well as the pruning severity and quantization widths.

In the work from Jiang et al. [62], a method is presented in which the design space exploration steps first choose a DNN model and generate an FPGA design to accelerate it that meets certain throughput specifications. Other model designs are then explored in search of ones that reduce the implied hardware accelerator resource requirements. Next, the selected model architecture is trained and fine-tuned such that results providing both higher accuracy and hardware efficiency are rewarded. This has also been done specifically for mapping execution of binarized neural networks to a hardware platform [23]. DNN model architecture/hardware accelerator architecture co-design has also been explored in the work by Abdelfattah et al. [63], where a DNN model was produced that has 1.3% better accuracy than ResNet [17] on certain metrics, while at the same time, the resulting hardware accelerator architecture increased performance per area by 41%.

These methods have the benefit of making a DNN model meet given latency and throughput constraints on a wider range of devices. On the other hand, it restricts the usage of more well-known DNN models that might be already pre-trained on large data sets and have more well-analyzed success/error rates.

2.2.8 Fast Convolution Algorithms

In 1980, Winograd presented a minimal-filtering algorithm [64] showing that the operations in a convolution can be re-arranged such that fewer multiplications and additions are performed for the same result compared to conventional convolution algorithms. The work from Lavin et al. [65] shows how this algorithm can be substituted into the compute pattern for convolutional layers of CNNs, resulting in a speedup of $2\times$ or more in throughput for these layers. Furthermore, the algorithm can be used to speed up the computation in both custom hardware architectures as well as in software running on CPUs or GPUs. This algorithm has since been exploited in numerous CNN accelerator works [66] [67] [68].

This method is the most similar approach to the methods explored in this thesis of advancing and applying efficient algebraic algorithms to deep learning hardware architectures. While the work from Lavin et al. [65] provided a good initiative in this direction, the hardware research community has since focused primarily on application of this one efficient algebraic technique which is only beneficial for speeding up convolutional layers. However, as revealed in the work from this thesis, there are multiple other under-explored efficient algebraic algorithms that can be extended and applied to deep learning hardware architectures. Furthermore, the methods from this thesis are based around speeding up matrix multiplication and are therefore applicable to speeding up a wider range of common DNN model layers including fully-connected, convolutional, recurrent, and attention/transformer layers. Finally, the Winograd convolution technique [65] still results in matrix multiplication, which may therefore still be complementary to our methods and applicable in addition to the techniques presented in this thesis.

2.3 Efficient Algebraic Algorithms

As shown in Section 2.2, recent years have seen many works on hardware-oriented DNN model optimizations and system-level improvements for deep learning hardware architectures. At a certain point, however, after hardware-oriented DNN model optimizations reach their limit, after the known parallelism and system-level optimizations for executing their compute patterns are exploited, and after technology scaling slows to a halt, there is an accelerator wall which causes limited improvement on the implementation side [69]. A less-explored avenue to continue advancement after this point is to reduce the workload at the algebraic level, by calculating the same deep learning model algebra, nevertheless using a re-arranged compute pattern which produces the same output from fewer or cheaper operations performed in hardware.

Consider the matrix multiplication $\boldsymbol{C} = \boldsymbol{AB}$ for $\boldsymbol{A}$ of size $M \times K$ and $\boldsymbol{B}$ of size $K \times N$. Using the conventional inner product, $\boldsymbol{C}$ is calculated from MNK multiplications and $MN(K-1)$ additions, where each element $c_{i,j}$ of $\boldsymbol{C}$ is calculated as follows:

$$c_{i,j} = \sum_{k=1}^{K} a_{i,k} b_{k,j} \, . \tag{2.17}$$

As discussed in Section 2.1, the majority of the computational workload in deep learning models can commonly be mapped to the matrix multiplication shown in (2.17), and as can be seen, the operations in this equation are a series of multiply-accumulate operations. For all deep learning accelerators, unless additional algebraic innovations are used, the throughput is ultimately limited by the maximum number of multiply-accumulate operations from (2.17) that can be performed per clock cycle. Due to this, deep learning accelerators contain a large number of MAC units, causing multipliers and MAC units to commonly be one

of the area-dominant resources in GEMM and deep learning accelerators [70], [3], [5], and an accelerator's throughput can be directly limited by how many multipliers its hardware budget can afford.

As a result, surpassing this theoretical performance per multiplier limit should be a key area of interest for advancing the field of deep learning hardware acceleration. As discussed in Section 2.2.8, this approach has been touched upon with Winograd's minimal filtering algorithms applied to convolutional neural networks (CNN)s [65], [64], [66], [70]. However, this algorithm is applicable only to CNN deep learning models, and there are numerous other efficient algebraic algorithms that are also applicable to GEMM and therefore a broader range of DNN models.

In this thesis, we continue in this under-explored direction and provide further algebraic enhancements for matrix multiplication algorithms and their custom hardware implementations for the application of deep learning acceleration. This section provides the necessary background on prior efficient algebraic algorithms and their hardware implementations outside the application of deep learning to build the foundation for our contributions.

2.3.1 Fast Inner Product (FIP)

In 1968, Winograd introduced an alternative inner-product algorithm [7] that we refer to as the Fast Inner Product (FIP). Compared to the traditional inner product in (2.17), FIP allows matrix multiplication to be performed with approximately half of the multiplication and accumulation operations traded for low-bitwidth additions. In FIP [7], each element $c_{i,j}$ of $\boldsymbol{C}$ is calculated as follows:

$$c_{i,j} = \sum_{k=1}^{K/2} (a_{i,2k-1} + b_{2k,j})(a_{i,2k} + b_{2k-1,j}) - \alpha_i - \beta_j \,, \tag{2.18}$$

where:

$$\alpha_i = \sum_{j=1}^{K/2} a_{i,2j-1} \cdot a_{i,2j} \tag{2.19}$$

$$\beta_j = \sum_{i=1}^{K/2} b_{2i-1,j} \cdot b_{2i,j} \,. \tag{2.20}$$

For even K, calculating $\boldsymbol{C}$ then requires the following number of multiplications:

$$\frac{MNK + MK + NK}{2} \,, \tag{2.21}$$

and the following number of additions:

$$\frac{3 \cdot MNK + MK + NK}{2} - MN - M - N \,. \tag{2.22}$$

This means that an accelerator performing this equation instead of (2.17) can trade nearly half of its multipliers for low-bitwidth adders while achieving the same throughput. Since the hardware footprint of fixed-point multipliers dominates that of adders [71], [25], [72], FIP can theoretically significantly improve the throughput per compute area roof of an accelerator, where *roof* refers to the upper bound of that metric and *compute area* refers to the area of computational logic used to perform arithmetic operations. Despite the high impact potential of these benefits, the FIP algorithm [7] has never before been implemented in a deep learning hardware accelerator.

Custom FIP Hardware Architectures

The FIP algorithm [7] has been explored by Gustafsson et al. for application to finite impulse response (FIR) filtering [73] in a non-systolic-array architecture proposal. Three prior works have also been proposed for systolic-array architectures for exploiting FIP [74] [75] [76].

However, none of the prior works on FIP architectures are focused on application to deep learning, which uses a certain range of matrix sizes and data representations, making it deserving of a stand-alone study. For all prior works related to deep learning acceleration which cite FIP [7], upon closer inspection, it becomes evident that the work either does not evaluate FIP, or the context of the citation is, in fact, referring to Winograd's more-popular minimal filtering algorithms for convolutional neural networks [65], [64] **rather than** Winograd's inner-product algorithm [7], on which our work presented in Chapter 4 is based. Furthermore, the work from Bravo et al. [76] reports no clear improvements from their FIP implementation, and all three other prior works on FIP architectures [73] [74] [75] are purely design proposals and do not provide empirical insight into its trade-offs.

The benefits of FIP in hardware rely on the premise that the hardware footprint of adders are cheaper than that of multipliers. Since the hardware complexity of fixed-point multipliers typically scales quadratically with the input bitwidth compared to linearly for adders, this premise has been shown to hold true for fixed-point data types [71], [25], [72]. However, the exact benefits in performance/resources/clock frequency when applying FIP to deep learning acceleration are still not immediately clear, and finding this out requires a comparison of hardware implementations that use multipliers/accumulators of the specific data sizes/types commonly used in deep learning accelerators.

In Chapter 4, we address the discussed gaps in prior research by implementing and evaluating FIP for the first time in a deep learning accelerator, we then identify a weakness of FIP in hardware and propose a new algorithm and its hardware architecture that inherently address that weakness, and we provide deep learning-specific optimizations for the FIP and FFIP algorithms and hardware architectures.

2.3.2 Karatsuba Scalar Multiplication (KSM)

The Karatsuba algorithm [9], introduced in 1962, was one of the first proposed multiplication algorithms asymptotically faster than the traditional approach. To explain the algorithm, we first begin by reviewing the conventional method for multi-digit scalar multiplication. Fig. 2.5 (a) shows the conventional method for performing 2-digit scalar multiplication where a w-bit multiplication is split into four smaller-bit scalar multiplications before being summed to form the final product. Algorithm 1 shows the generalization of this, where n-digit multiplication is performed by carrying out the same steps recursively for each smaller-bit multiplication.

Fig. 2.5 (b) shows the Karatsuba algorithm [9] for 2-digit scalar multiplication where a w-bit multiplication is split instead into *three* smaller-bit scalar multiplications before being summed to form the final product. Algorithm 2 shows the generalization of this, where n-digit multiplication is performed by carrying out the same steps recursively for each smaller-bit multiplication.

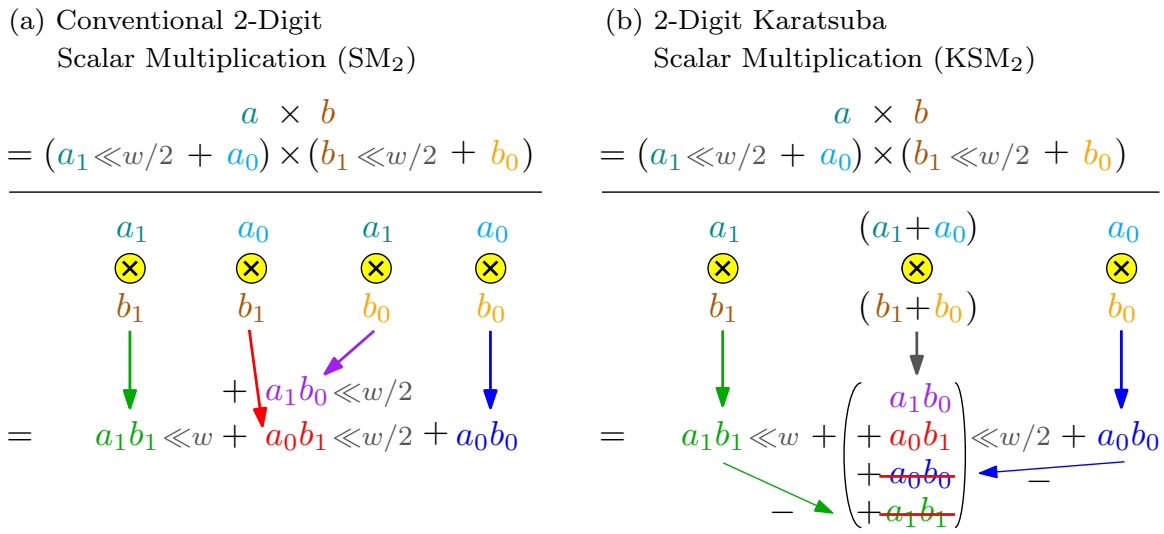

Figure 2.5: SM_2 algorithm illustration on left, KSM_2 algorithm illustration on right. Compared to SM_2, KSM_2 requires only 3 single-digit multiplications, however, it requires 3 more additions, increasing the overall operation count.

Algorithm 1 Conventional n-Digit Scalar Multiplication.

function $SM_n^{[w]}(a, b)$
  if $(n > 1)$ **then**
    $a_1 = a^{[w-1:\lceil w/2 \rceil]}$
    $a_0 = a^{[\lceil w/2 \rceil - 1:0]}$
    $b_1 = b^{[w-1:\lceil w/2 \rceil]}$
    $b_0 = b^{[\lceil w/2 \rceil - 1:0]}$
    $c_1 = SM_{n/2}^{[\lfloor w/2 \rfloor]}(a_1, b_1)$
    $c_{10} = SM_{n/2}^{[\lceil w/2 \rceil]}(a_1, b_0)$
    $c_{01} = SM_{n/2}^{[\lceil w/2 \rceil]}(a_0, b_1)$
    $c_0 = SM_{n/2}^{[\lceil w/2 \rceil]}(a_0, b_0)$
    $c = c_1 \ll w$
    $c \mathrel{+}= (c_{01} + c_{10}) \ll \lceil w/2 \rceil$
    $c \mathrel{+}= c_0$
  else
    $c = a \times b$
  end if
  return c
end function

Algorithm 2 n-Digit Karatsuba Scalar Multiplication.

function $\text{KSM}_n^{[w]}(a, b)$
  if $(n > 1)$ **then**
    $a_1 = a^{[w-1:\lceil w/2 \rceil]}$
    $a_0 = a^{[\lceil w/2 \rceil - 1:0]}$
    $b_1 = b^{[w-1:\lceil w/2 \rceil]}$
    $b_0 = b^{[\lceil w/2 \rceil - 1:0]}$
    $a_s = a_1 + a_0$
    $b_s = b_1 + b_0$
    $c_1 = \text{KSM}_{n/2}^{[\lfloor w/2 \rfloor]}(a_1, b_1)$
    $c_s = \text{KSM}_{n/2}^{[\lceil w/2 \rceil + 1]}(a_s, b_s)$
    $c_0 = \text{KSM}_{n/2}^{[\lceil w/2 \rceil]}(a_0, b_0)$
    $c = c_1 \ll w$
    $c \mathrel{+}= (c_s - c_1 - c_0) \ll \lceil w/2 \rceil$
    $c \mathrel{+}= c_0$
  else
    $c = a \times b$
  end if
  return c
end function

Custom Karatsuba Hardware Architectures

Prior works on KSM-based low-bitwidth accurate integer multiplier circuits have shown some area benefits for input bitwidths in the range of 64 bits or less, with minimal area improvements in the smallest ranges of 16 bits [77], [78]. This affirms that, while KSM theoretically reduces the complexity of large-bitwidth integer multiplication, the extra addition operations it introduces limit its area reduction in multiplier circuits for smaller integers of more commonly-used bitwidths.

To address this, Chapter 5 shows how KSM can be extended to matrix multiplication to reduce the impact of these extra additions similarly to how the algorithm in Fig. 2.5 (a) can be extended to matrix multiplication as illustrated in Fig. 2.6. In Fig. 2.6, four separate partial-product matrix multiplications are performed between matrices each containing bit

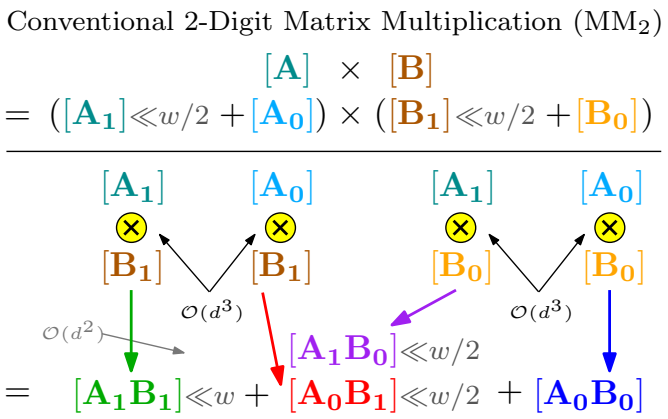

Figure 2.6: MM_2 algorithm illustration. The 4 single-digit matrix multiplications of complexity $\mathcal{O}(d^3)$ dominate the $\mathcal{O}(d^2)$ complexity of the matrix additions.

slices of every element, and they are later summed together to form the final matrix product. Algorithm 3 shows the generalization of this, where n-digit matrix multiplication is performed by carrying out the same steps recursively for each smaller-bit matrix multiplication. The elements in matrices $\mathbf{A_0}$ and $\mathbf{B_0}$ contain the lower bits (bits $\lceil w/2 \rceil - 1$ to 0) of every element in the $\mathbf{A}$ and $\mathbf{B}$ matrices, while $\mathbf{A_1}$ and $\mathbf{B_1}$ contain the upper bits (bits $w - 1$ to $\lceil w/2 \rceil$) of every element in matrices $\mathbf{A}$ and $\mathbf{B}$. This allows for w-bit matrix multiplication using smaller m-bit multipliers. The MM_1 algorithm on line 15 of Algorithm 3 is a conventional matrix multiplication algorithm such as that in (2.17).

Chapter 5 then shows how Karatsuba scalar multiplication extended to matrix multiplication can be exploited in hardware to increase the performance-per-area limits of matrix multiplication hardware. It is also shown there how this can be used to reduce the minimum possible execution times of precision-scalable hardware architectures such that they scale less than quadratically with the input bitwidths w. The hardware algorithms used in prior works on precision-scalable hardware architectures use variations of the SM and MM algorithms shown in Algorithms 1 and 3 to combine partial products and compute

Algorithm 3 Conventional n-Digit Matrix Multiplication.

function $\text{MM}_n^{[w]}(\mathbf{A}, \mathbf{B})$
  if $(n > 1)$ **then**
    $\mathbf{A_1} = \begin{bmatrix} a_{1,1}^{[w-1:\lceil w/2 \rceil]}, & \dots & a_{1,K}^{[w-1:\lceil w/2 \rceil]} \\ \dots & \dots & \dots \\ a_{M,1}^{[w-1:\lceil w/2 \rceil]}, & \dots & a_{M,K}^{[w-1:\lceil w/2 \rceil]} \end{bmatrix}$
    $\mathbf{A_0} = \begin{bmatrix} a_{1,1}^{[\lceil w/2 \rceil-1:0]}, & \dots & a_{1,K}^{[\lceil w/2 \rceil-1:0]} \\ \dots & \dots & \dots \\ a_{M,1}^{[\lceil w/2 \rceil-1:0]}, & \dots & a_{M,K}^{[\lceil w/2 \rceil-1:0]} \end{bmatrix}$
    $\mathbf{B_1} = \begin{bmatrix} b_{1,1}^{[w-1:\lceil w/2 \rceil]}, & \dots & b_{1,N}^{[w-1:\lceil w/2 \rceil]} \\ \dots & \dots & \dots \\ b_{K,1}^{[w-1:\lceil w/2 \rceil]}, & \dots & b_{K,N}^{[w-1:\lceil w/2 \rceil]} \end{bmatrix}$
    $\mathbf{B_0} = \begin{bmatrix} b_{1,1}^{[\lceil w/2 \rceil-1:0]}, & \dots & b_{1,N}^{[\lceil w/2 \rceil-1:0]} \\ \dots & \dots & \dots \\ b_{K,1}^{[\lceil w/2 \rceil-1:0]}, & \dots & b_{K,N}^{[\lceil w/2 \rceil-1:0]} \end{bmatrix}$
    $\mathbf{C_1} = \text{MM}_{n/2}^{[\lfloor w/2 \rfloor]}(\mathbf{A_1}, \mathbf{B_1})$
    $\mathbf{C_{10}} = \text{MM}_{n/2}^{[\lceil w/2 \rceil]}(\mathbf{A_1}, \mathbf{B_0})$
    $\mathbf{C_{01}} = \text{MM}_{n/2}^{[\lceil w/2 \rceil]}(\mathbf{A_0}, \mathbf{B_1})$
    $\mathbf{C_0} = \text{MM}_{n/2}^{[\lceil w/2 \rceil]}(\mathbf{A_0}, \mathbf{B_0})$
    $\mathbf{C} = \mathbf{C_1} \ll w$
    $\mathbf{C} \mathrel{+}= (\mathbf{C_{10}} + \mathbf{C_{01}}) \ll \lceil w/2 \rceil$
    $\mathbf{C} \mathrel{+}= \mathbf{C}_0$
  else
    $\mathbf{C} = \text{MM}_1^{[w]}(\mathbf{A}, \mathbf{B})$
  end if
  return $\mathbf{C}$
end function

variable-bitwidth w-bit matrix products using smaller m-bit multipliers, where the number of m-bit multiplications and minimum possible execution time if fully utilizing the m-bit multipliers scales quadratically with the input bitwidths w. In contrast, the minimum possible execution time of the precision-scalable hardware architectures proposed in Chapter 5 scale less than quadratically with the input bitwidths w.

2.3.3 Strassen Matrix Multiplication (SMM)

The Strassen algorithm [79], proposed in 1969, was one of the first matrix multiplication algorithms showing that the n^3 complexity of the traditional approach is not optimal. To understand the algorithm, first consider how the matrix multiplication $\mathbf{C} = \mathbf{AB}$ below can be computed by dividing $\mathbf{A}$ and $\mathbf{B}$ into 4 matrix blocks, where $\mathbf{C}$ is then computed by carrying out 8 matrix block multiplications and 4 matrix block additions between the $\mathbf{A}$ and $\mathbf{B}$ blocks as follows:

$$\begin{bmatrix} \mathbf{C_{11}} & \mathbf{C_{12}} \\ \mathbf{C_{21}} & \mathbf{C_{22}} \end{bmatrix} = \begin{bmatrix} \mathbf{A_{11}B_{11}} + \mathbf{A_{12}B_{21}} & \mathbf{A_{11}B_{12}} + \mathbf{A_{12}B_{22}} \\ \mathbf{A_{21}B_{11}} + \mathbf{A_{22}B_{21}} & \mathbf{A_{21}B_{12}} + \mathbf{A_{22}B_{22}} \end{bmatrix}. \tag{2.23}$$

This process can then be carried out recursively again on each matrix block product by splitting the matrix blocks again into smaller blocks and repeating the same process again.

The Strassen algorithm [79], shown in (2.24)-(2.25), provides a way to carry out (2.17)

instead using 7 matrix block multiplications and 18 matrix block additions as follows:

$$\begin{aligned}
\mathbf{T_1} &= \mathbf{A_{11}} + \mathbf{A_{22}} & \mathbf{S_1} &= \mathbf{B_{11}} + \mathbf{B_{22}} \\
\mathbf{T_2} &= \mathbf{A_{21}} + \mathbf{A_{22}} & \mathbf{S_2} &= \mathbf{B_{11}} \\
\mathbf{T_3} &= \mathbf{A_{11}} & \mathbf{S_3} &= \mathbf{B_{12}} - \mathbf{B_{22}} \\
\mathbf{T_4} &= \mathbf{A_{22}} & \mathbf{S_4} &= \mathbf{B_{21}} - \mathbf{B_{11}} \\
\mathbf{T_5} &= \mathbf{A_{11}} + \mathbf{A_{12}} & \mathbf{S_5} &= \mathbf{B_{22}} \\
\mathbf{T_6} &= \mathbf{A_{21}} - \mathbf{A_{11}} & \mathbf{S_6} &= \mathbf{B_{11}} + \mathbf{B_{12}} \\
\mathbf{T_7} &= \mathbf{A_{12}} - \mathbf{A_{22}} & \mathbf{S_7} &= \mathbf{B_{21}} + \mathbf{B_{22}}
\end{aligned} \tag{2.24}$$

$$\begin{aligned}
\mathbf{Q_1} &= \mathbf{T_1} \cdot \mathbf{S_1} \\
\mathbf{Q_2} &= \mathbf{T_2} \cdot \mathbf{S_2} \\
\mathbf{Q_3} &= \mathbf{T_3} \cdot \mathbf{S_3} \\
\mathbf{Q_4} &= \mathbf{T_4} \cdot \mathbf{S_4} \\
\mathbf{Q_5} &= \mathbf{T_5} \cdot \mathbf{S_5} \\
\mathbf{Q_6} &= \mathbf{T_6} \cdot \mathbf{S_6} \\
\mathbf{Q_7} &= \mathbf{T_7} \cdot \mathbf{S_7}
\end{aligned} \qquad \begin{aligned}
\mathbf{C_{11}} &= \mathbf{Q_1} + \mathbf{Q_4} - \mathbf{Q_5} + \mathbf{Q_7} \\
\mathbf{C_{12}} &= \mathbf{Q_3} + \mathbf{Q_5} \\
\mathbf{C_{21}} &= \mathbf{Q_2} + \mathbf{Q_4} \\
\mathbf{C_{22}} &= \mathbf{Q_1} - \mathbf{Q_2} + \mathbf{Q_3} + \mathbf{Q_6}
\end{aligned} \; . \tag{2.25}$$

Winograd Form

The Winograd form of the Strassen algorithm [80] has the same asymptotic complexity but requires 15 matrix block additions at each level of recursion rather than 18. However, for fixed-point data types, this form increases the multiplier input datapath bitwidth by up to 2 bits at each recursion level rather than 1 bit. Due to this, we focus on the original form of

the Strassen algorithm from (2.24)-(2.25) in our work instead.

Executing Strassen on GPUs and CPUs

Strassen's algorithm has been well explored in prior work for execution on general-purpose CPUs and GPUs [81], [82], [83], [84], [85], [86], [87], [88]. However, its execution on CPUs and GPUs in these prior works does not achieve the promised theoretical speedups unless the widths/heights of the matrices being multiplied are in the range of at least 1024 elements or even much larger. This limits the applicability of Strassen's algorithm on CPUs and GPUs for modern workloads such as deep learning that do not always decompose to such large matrix multiplications.

Strassen's algorithm cannot achieve the promised speedups in CPUs and GPUs because irregularities introduced in the algorithm such as the extra data accesses required for reading/computing/storing additional intermediate matrices before/after the matrix multiplication steps all add to the overall execution time beyond what would be expected purely from a theoretical analysis of only the number of required arithmetic operations [83], [87].

Custom Strassen Hardware Architectures

While software implementations of the Strassen algorithm on CPUs and GPUs has been well explored in prior work, custom hardware designs for efficiently exploiting the Strassen algorithm in hardware remain under-explored. A systolic-array design concept for implementing Strassen's algorithm for one level of recursion on 2×2 matrices has been proposed in the work by Elfimova et al. [74] without evaluation of an implementation. Another hardware design for implementing Strassen's algorithm for one level of recursion on 2×2 matrices has also been proposed in the work by León-Vega et al. [89], where the Strassen

architecture reduced the DSP usage by up to 12.5% at the expense of 25-40% increase in LUT resources to implement the additional adders.

In Chapter 6, we address this gap by presenting and evaluating a new hardware architecture that efficiently translates the theoretical Strassen complexity reductions directly into hardware resource savings beyond what has been presented in prior work. Unlike the only two prior works on custom hardware designs for executing the Strassen algorithm, we propose architectures in this work that allow for Strassen's algorithm to be implemented on matrices larger than 2×2, which is essential for minimizing the complexity penalty of the additional adders, and the architectures are capable of implementing multiple levels of Strassen recursion to achieve greater hardware resource savings. Furthermore, the proposed architectures allow proven traditional systolic arrays to be still used at the core, or they allow Strassen's algorithm to be used in combination with other hardware designs that can efficiently perform further algebraic optimizations on matrices after the Strassen portion is carried out, such as techniques proposed in Chapter 4 and Chapter 5. Finally, the proposed Strassen architectures are multi-systolic-array designs, meaning they can multiply smaller matrices with higher utilization than single-systolic-array designs with the same computational strength.

Furthermore, the Strassen architecture proposed in Chapter 6 is a multi-systolic-array architecture, meaning it can multiply smaller matrices with higher utilization than single-systolic-array designs. Prior multi-systolic-array designs achieve this property by implementing variations of (2.23) in hardware by dividing larger matrices into smaller tiles/blocks, executing the smaller matrix tile multiplications on multiple smaller systolic arrays, and later summing the tile products to form the final larger matrix multiplication product. Chapter 6 shows how to efficiently implement (2.24)-(2.25) in hardware to require less hardware

area to achieve this same goal as in prior works of efficiently supporting multiplication of smaller matrices.

2.4 Summary

The field of deep learning has seen increasing breakthroughs and commercial adoption in recent years for enabling a wide range of applications including image/speech recognition and human-like chatbots. This has led to a growing need for computer hardware that can quickly and efficiently execute these deep learning applications, which increasingly require massive amounts of computational power to execute.

To understand how to further improve deep learning hardware to address these needs, this chapter first discussed in Section 2.1 how the computationally dominant portions of deep learning workloads can commonly be decomposed to matrix multiplication, highlighting the importance of focusing on speeding up matrix multiplication in order to continue advancement in the field deep learning acceleration. Section 2.2 then discussed relevant prior approaches for improving deep learning execution in hardware, and it was identified that advancement and application of efficient matrix multiplication algorithms in deep learning hardware architectures is an under-explored field in this area with potential for new exploration. Section 2.3 then presented relevant background and prior works on efficient algebraic algorithms and their hardware implementations outside the application of deep learning to provide the foundation for presenting our contributions in Chapters 4 - 6 of deriving and applying new or under-explored efficient matrix multiplication algorithms in custom deep learning hardware architectures.

Chapter 3

Deep Learning Accelerator System Architecture

The key contributions of this thesis are the proposed algebraic algorithms and/or custom hardware architectures for exploiting them and their application to deep learning acceleration. However, in order to ensure that there are no unseen side-effects, it is important to integrate the hardware architectures into a full-system implementation to fully assess the end benefits at the application level. Furthermore, our goal is to validate the benefits of the hardware architectures when overlaid on top of systems based on the most efficient systolic array accelerators used in practice. Therefore, the high-level organization of our system design shown in Fig. 3.1 *intentionally* shares similarities with the TPUv1 [3] and some similarities with the TPUv2-TPUv4 as well [5] [6]. Based on this, we provide here the system-level implementation we used for the sake of completeness and for better understanding of the context in which our main contributions have been validated.

3.1 Overview

The accelerator system, shown in Fig. 3.1, consists of a GEMM Unit for multiplying and accumulating matrix products, a Post-GEMM Unit for performing DNN-specific functions after matrix multiplication, a Memory Unit for storing and accessing data, an Instruction Unit for decoding instructions sent from the host, and an RxTx Unit for communicating with the host through PCIe. The design is specialized for performing inference of non-sparse DNN models on fixed-point inputs consisting of convolutional layers, fully-connected layers, and pooling layers. However, the system is also a highly-optimized GEMM accelerator in general. Based on this, the contributions from this thesis are also applicable to matrix multiplication acceleration in general, and therefore can be extended to aiding all deep learning model layers that can mainly decompose to matrix multiplication, including fully-connected, convolutional, recurrent, and attention/transformer layers, with most of our contributions being relevant for computation of fixed-point data types.

The system design allows for matrix multiplication or neural network inference to be carried out in a highly deterministic/time-predictable manner where data is passed through at high throughput rates without intermediate stores/loads and operation scheduling to be required between the GEMM and post-GEMM execution stages. The data flows directly from matrix multiplication to accumulation to post-GEMM operations. This allows the accelerator to achieve very high utilizations that approach its theoretical throughput roof. This was a difficult property to achieve, but a highly efficient system was necessary as a starting point for demonstrating the full benefits of the proposed MXU architectures which allow accelerators to surpass their normal throughput limits when overlayed onto efficient systems already reaching close to their normal theoretical limits.

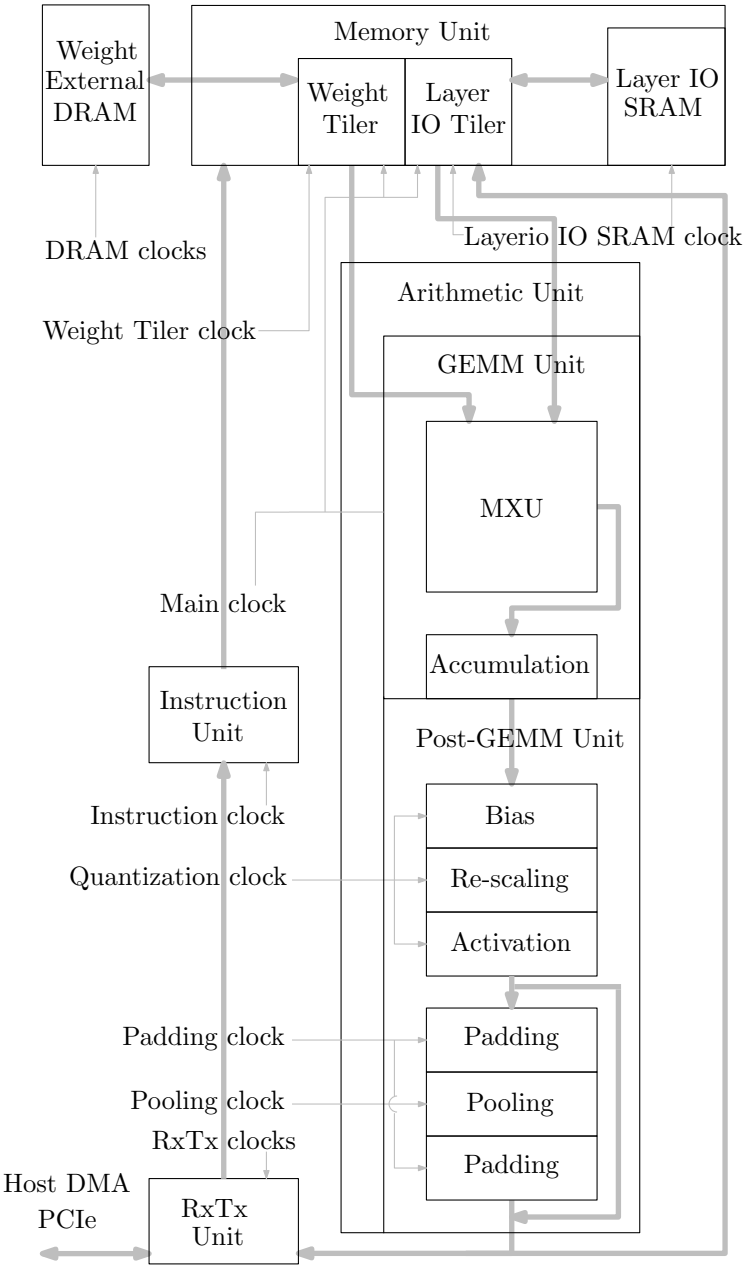

Figure 3.1: The example accelerator system design used to host and evaluate the baseline and proposed MXUs when overlaid on top of systems based on the most efficient systolic array accelerators used in practice, e.g., the TPU [3], [5], [6].

Algorithm 4 Layer IO memory access pattern implemented in the counters from Fig. 3.2 for in-place mapping of 2-D convolution to GEMM, provided for the sake of completeness.

for n_t = 0; $n_t < N_t \cdot N_t s$; n_t += $N_t s$
  for h_t = 0; $h_t < H_t \cdot H_t s$; h_t += $H_t s$
    for kh = 0; $kh < KH \cdot KHs$; kh += KHs
      for kw = 0; $kw < KW \cdot KWs$; kw += KWs
        for cin_t = 0; $cin_t < Cin_t \cdot Cin_t s$; Cin_t += $Cin_t s$
          for h = 0; $h < H \cdot Hs$; h += Hs
            for w = 0; $w < W \cdot Ws$; w += Ws
              $k_{\text{offset}} = kh + kw + cin_t$
              $m_{\text{offset}} = h_t + h + w$
              address = $m_{\text{offset}} + k_{\text{offset}}$
            end for
          end for
        end for
      end for
    end for
  end for
end for

3.2 Memory Subsystem

The memory accesses are controlled by multi-digit counters (also referred to as tilers) shown in Fig. 3.2 containing programmable digit sizes and strides which calculate GEMM read/write patterns, as well as map two-dimensional convolution to matrix multiplication to perform Algorithm 4 for any matrix or convolution window sizes without requiring a stand-alone memory remapping stage. The size and strides for each layer are calculated offline once per neural network, and then can be re-used for all inferences of the same neural network after that where they are updated in the memory tilers in real time between each layer. The tilers allow the Memory Unit and external dynamic random-access memory (DRAM) to be interfaced from the Arithmetic Unit using simple first-in first-out (FIFO) interfaces.

The convolutional layer inputs consist of three dimensions, depth Cin, height H, and

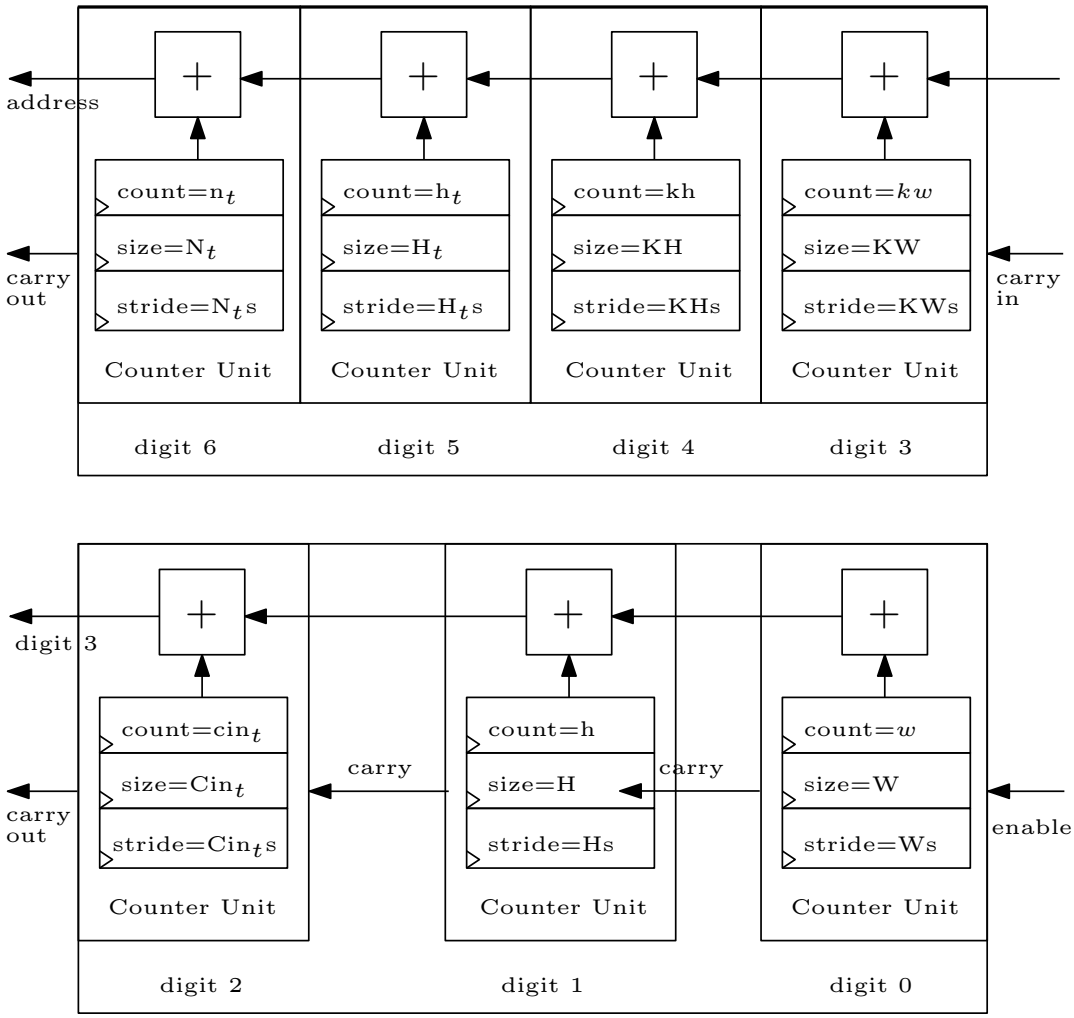

Figure 3.2: Layer IO memory access counters for performing in-place mapping of 2-D convolution to GEMM.

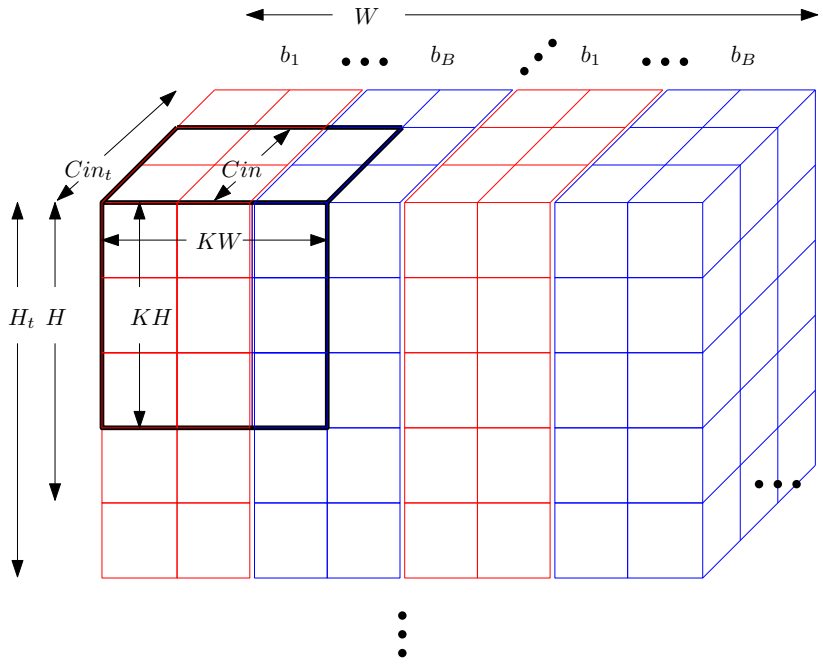

Figure 3.3: Layer IO memory access and blocking scheme to partition the memory and perform in-place mapping of two-dimensional convolution to GEMM, provided for context.

width W. The Cin and H dimensions are each split into two separate dimensions to divide them into tiles, and a dimension with the $_t$ subscript denotes the number of tiles in that dimension (e.g. the H dimension is split into H_t tiles each of a new smaller size H). Consider that KH and KW stand for the height and width of the kernel size, a dimension suffixed with s represents the dimension's stride (e.g. Ws is the stride for the W dimension), and N_t stands for the number of tile columns in the B/weight matrix. The convolution is then mapped in-place to GEMM as shown in Algorithm 4 where all of the layer input dimensions are mapped to tiled matrices by mapping each dimension to either a K or M dimension. Similarly, the weight dimensions are tiled and mapped to either the K or N dimensions for GEMM. Each memory location contains X elements along the Cin dimension, where X is the MXU width.

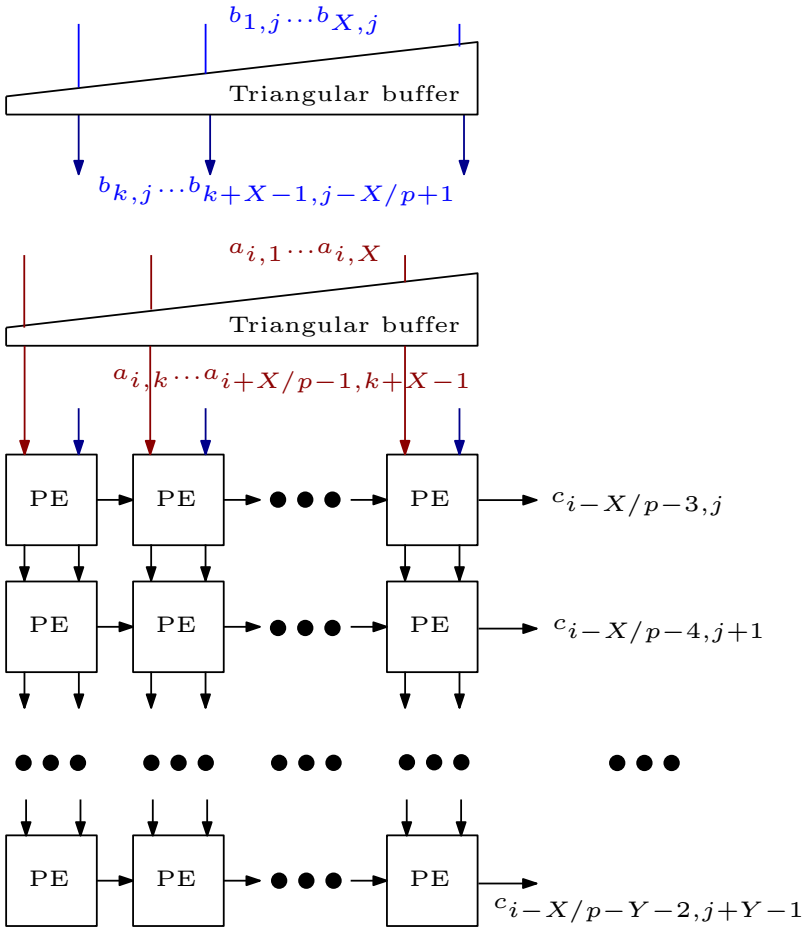

Figure 3.4: The baseline MXU architecture.

3.3 GEMM Unit

The GEMM unit contains the MXU as well as SRAM and addition logic for accumulating matrix tiles. Fig. 3.4 shows how the PEs are laid out into an MXU architecture. The MXU input buffers shown are triangular-shaped register arrays containing X shift registers of varying depths, with each shift register SR_k with depth k loading one $a_{i,k}$ or $b_{k,j}$ value per clock cycle.

Within the GEMM Unit, an $\mathbf{A}$ matrix represents the inputs of a DNN layer after being

rearranged by the memory subsystem described in Section 3.2 for converting the convolutional layer operation to GEMM, a $\mathbf{B}$ matrix represents the layer weights after being rearranged for GEMM, and a C matrix represents the GEMM output that corresponds to the layer's output values. In order to perform GEMM on an MXU, the input matrices are divided into tiles and fed to the MXU one-by-one. Following each tile multiplication, the partial tile products are accumulated outside of the MXU to generate each final matrix product tile. Prior to each tile multiplication, a $\mathbf{B}$ tile is loaded into the MXU. It then remains in place as the $\mathbf{A}$ tile flows through the MXU producing the tile product, during which a new $\mathbf{A}_{i,:}$ vector is fed into the MXU each clock cycle. Additionally, to hide the latency of loading $\mathbf{B}$ tiles, the MXU PEs each contain one extra b buffer to load the next $\mathbf{B}$ tile into the MXU as the current tile is being multiplied.

3.4 Post-GEMM Unit

The post-GEMM unit contains neural network-specific functions to be performed on the matrix multiplication outputs. This includes adding the bias values, inter-layer rescaling for quantization [20], activation, padding, and pooling.

The re-scaling unit in Fig. 3.1 carries out inter-layer rescaling that is necessary for readjusting the zero points and scaling factors of the quantized data which may be different in each layer. This is executed to carry out the method defined by Jacob et al. [20], where each 2-dimensional data vector going through the post-GEMM unit passes through a series of vector operation units that perform the a variable shift and multiplication on each of the elements in the data vector. Prior to this, an addition is performed on each element to add the neural network layer's bias. Following this, a ReLU activation is performed on each element by passing zero if the element is less than zero, otherwise passing the same value.

3.5 Host Access and Accelerator Instructions

The accelerator has direct memory access (DMA) to the system memory in the host through a Peripheral Component Interconnect Express (PCIe) 3.0 connection. We use a PCIe DMA controller and host driver provided by Xillybus [90]. From the hardware end, the PCIe controller is interfaced through FIFOs. From the software end, the driver is interfaced through shell scripts which we interface with through Python to provide an end-to-end framework for running deep learning applications on the accelerator.

The host sends variable-length instruction packets to the accelerator each containing a 32-bit header consisting of an 8-bit opcode and a 24-bit field determining the number of bytes remaining in the packet after the header. Each instruction packet can contain the following information:

- Input data: This contains the input data to the neural network such as images for image classification.
- Weight data: The neural network weights.
- Layer memory tiler count/size parameters: The size and stride of each neural network layer.
- Quantization parameters: Contains inter-layer rescaling parameters for quantization.
- Other layer parameters: This includes other information about each layer such as layer input bitwidths, layer dimensions, pooling kernel dimensions.
- Top-level instruction: this includes instructions for the accelerator such as initiating inference and passing the result back to the host, and reading performance counters.

The weights and model parameters for a specific neural network are pre-loaded onto the accelerator once each time that a new neural network model needs to be accelerated.

3.6 Timing Optimizations

3.6.1 Memory Subsystem Timing Optimizations

It was initially found that the maximum frequency of the memory access control counters was lower than the GEMM Unit clock frequency, which would lower the throughput of the entire accelerator and restrict the full frequency advantages of the FFIP method described in Chapter 4 from being properly evaluated. To resolve this, we partitioned the Layer IO memory into submemories each containing a block of the total memory and separate memory tiler controllers. This allowed each memory block to be accessed concurrently at a slower clock speed and for their read data to then be accessed in an interleaved manner by the main clock.

To do this, we derived a memory partitioning scheme for dividing the convolutional layer inputs into blocks in a way that still supports in-place remapping of two-dimensional convolution to GEMM as described above. The partitioning scheme supports dividing the memory into B blocks for any B that is a power of 2, allowing the memory tiler controllers to run at a frequency of $1/B$ times the main clock frequency. In our results, we use $B = 2$. The W dimension is divided into different blocks as show in Fig. 3.3, where each W slice is Ws elements wide.

One remaining issue, however, is that when the KW dimension increments to a certain range, the submemory may need to access elements from a block that it does not contain. For example, consider the case in Fig. 3.3 where $kh = kw = 3$, $Hs = Ws = 2$, and

$B = 2$. When $kw \in \{1, 2\}$ then element 1 in block 1 will be accessed by the main clock for $w = 1$ followed by element 1 in block 2 for $w = 2$. However, when $kw = 3$ then the block 1 submemory should require access starting with an element from a block that it does not contain. To resolve this, when $kw = 3$ then block 2 will be accessed first by the main memory for $w = 1$, followed by the next block 1 slice for $w = 2$. In other words, the next memory elements are instead taken from the adjacent submemory, the kw and w incrementer digits are adjusted in each submemory, and the interleaving order in which the submemories are accessed from the main clock is modified.

We also run the weight memory control logic at a fraction of the main clock speed by accessing the memory in bursts and thus requiring memory control to be communicated to the DRAM controller at an infrequent rate relative to the main clock speed. The external DRAM memory is used only for storing the weights, and the layer inputs/outputs always stay in on-chip memory. This allows the device's external memory bandwidth to rarely be a bottleneck for the implementations evaluated in our experimental setup.

3.6.2 MXU Timing Optimizations

As discussed in Section 3.3, prior to each tile multiplication, a b/y tile is loaded into the MXU. It then remains in place as the a/g tile flows through the MXU to produce the tile product. As show in Fig. 3.5, our initial implementation for the mechanism which shifts b/y tiles into the MXU was to implement a basic one-dimensional shift register vector to shift each column of weights into the MXU, and then have an enable control signal that connects to each element in the vector to stop the shifting all at once when the weight column is fully loaded. However, this required enable control signals to be connected non-locally to every element's shift register enable inputs without being able to locally buffer the enable control

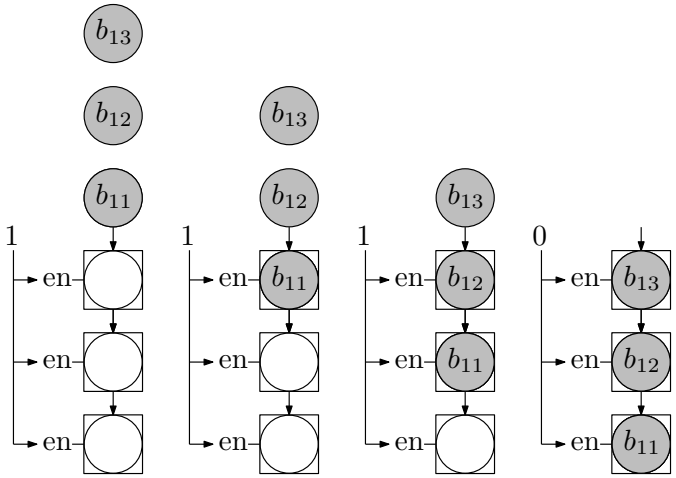

Figure 3.5: MXU weight column shift register logic requiring control signal to be connected to each element in the column.

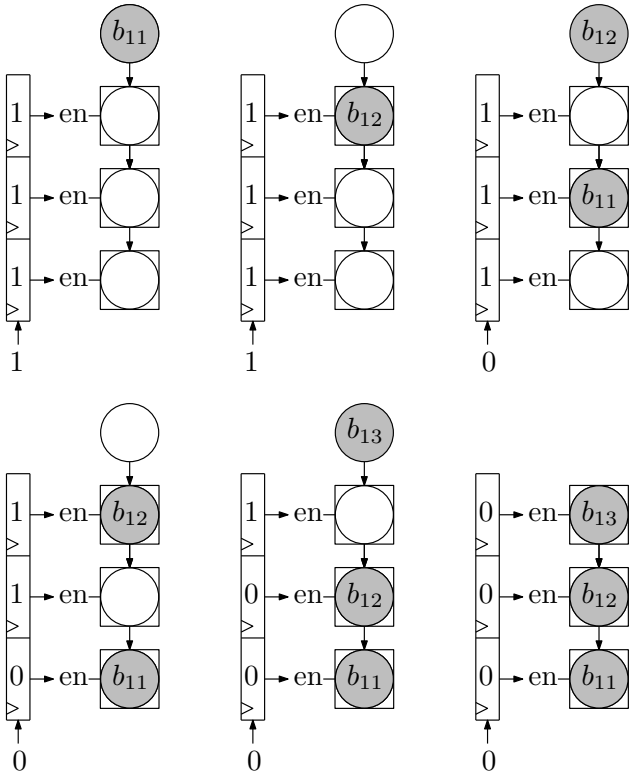

Figure 3.6: MXU weight column shift register logic with control connections fully localized to adjacent elements.

signals close to their destinations.

In order to mitigate this, we instead designed an improved shift-register-based mechanism shown in Fig. 3.6 in which the control signal connections are fully localized by being buffered directly before their destination in each element's shift register enable input to improve the frequency potential of the MXU. In this design, the control signal has its own shift register mechanism separate from the datapath shift register which is pre-loaded with 1's. To allow this mechanism, the weights are shifted in every-other clock cycle instead of every clock cycle. Shifting in the weight vectors more slowly like this does not affect the throughput so long as the layer input M_t tile size can usually be at least twice as large as the N_t tile size used for the weights, which was found to be true for the models we used for evaluation.

3.7 Summary

The key contributions of this work are the proposed algebraic algorithm and/or their custom hardware architectures and their evaluation in a deep learning accelerator system. Furthermore, our goal is to evaluate the hardware architectures when overlaid on top of a system based on the most efficient systolic array accelerators used in practice. Therefore, this section expanded upon our system design shown in Fig. 3.1 that *intentionally* shares similarities with the TPUv1 [3] and some similarities with the TPUv2-TPUv4 as well [5], [6], and provides the context in which our main contributions have been validated to better understand their overall impact.

The source code for our accelerator implementation is available on GitHub [91] and is hand-coded in SystemVerilog and implemented to be highly configurable. The MXU height/width can be parameterized to be any value that is a multiple of 4. The bitwidth of

the a and b values are also fully parameterizable, as well as whether they are on signed or unsigned representations.

Chapter 4

Fast Inner-Product Algorithms and Hardware Architectures

In 1968, Winograd introduced an alternative inner-product algorithm [7] shown in (2.18) that we refer to as the Fast Inner Product (FIP) that can be used to reduce the complexity of matrix multiplication. Prior research on algebraic enhancements applied to deep learning acceleration has been focused primarily on using Winograd's minimal filtering algorithms applied to convolutional neural networks (CNN)s [65], [64]. However, other fast algebraic algorithms such as the unrelated FIP algorithm also proposed by Winograd on which our work in this chapter is based remain under-explored for application to deep learning hardware designs.

Compared to the traditional inner product (referred to as baseline), FIP allows matrix multiplication to be performed with approximately half of the multiply-accumulate (MAC) operations traded for low-bitwidth additions. Since MAC units are commonly the area-dominant computational resource in DNN accelerators [70], [3], [5], cutting the number of required MAC units in half could nearly double the performance per compute area.

Despite the high impact potential of these benefits, the FIP algorithm [7] has never before been implemented in a deep learning hardware accelerator. The first step in our work in this chapter is to address this oversight and carry out that investigation. We show that, while the number of required MAC units is nearly halved as expected, there is also a reduction in clock frequency and, as a consequence, throughput.

In order to address this weakness of FIP, we introduce a new inner-product algorithm referred to as the Free-pipeline Fast Inner Product (FFIP) that achieves the same near 2× reduction in required MAC units, however, it inherently improves the clock frequency and, as a consequence, overall throughput for a similar hardware cost compared to FIP. Additionally, as expected from our theoretical analysis, the effective size of the largest systolic array that could be fit in our experimental compute platform was increased from 56×56 processing elements (PE)s when using a baseline systolic array to 80×80 PEs when using (F)FIP systolic arrays, an increase of over 2× in effective number of PEs.

Our work shows that the FFIP architecture is functionally equivalent to traditional systolic-array architectures, and that it can be seamlessly incorporated into any deep learning accelerator system that uses traditional fixed-point systolic arrays for the arithmetic to double the throughput per MAC unit and significantly increase its performance per compute area for inference of all deep learning models that will execute on the systolic array. This is because any such accelerator could substitute its traditional systolic-array PEs for just nearly half the number of proposed FFIP PEs without fundamentally altering the accelerator's functionality or internal interfaces in any way. Furthermore, systolic-array accelerators have been proven effective for accelerating a wide range of modern deep learning models [3], [5], [6], [55], [16]. Our results indicate that FFIP, when overlaid on top of the most efficient systolic-array accelerator systems used in practice, can further improve

compute efficiency and increase the theoretical performance limits across a wide range of devices, system implementations, and deep learning models.

In summary, our key contributions from this chapter, which have been published in [8], are the following:

- We implement FIP for the first time in a deep learning accelerator and validate our generalized analysis of its ability to increase an accelerator's theoretical throughput and compute efficiency limits for the data types commonly used in deep learning acceleration. As part of this undertaking, we also identify a weakness in FIP of a reduction in clock frequency. To address this, we introduce a new inner-product algorithm (FFIP) in Section 4.2.2 to inherently bypass this trade-off. Additionally, we provide deep learning-specific optimizations for both the FIP and FFIP algorithms in Section 4.1.2.

- We propose a generalized PE architecture in Section 4.2.2 for performing FFIP in hardware which inherently improves the clock frequency and, as a consequence, overall throughput for a similar hardware cost compared to FIP. Additionally, we provide deep learning-specific hardware optimizations for the systolic-array architectures housing FIP or FFIP PEs in Section 4.2.4. We demonstrate in Section 4.3 how an example implementation of the FFIP architecture achieves higher throughput and compute efficiency across different deep learning models than the best-in-class prior solutions implemented on the same type of compute platform. More importantly, the results indicate that when FFIP is overlaid on top of the most efficient systolic-array systems used in practice, it can further increase compute efficiency in the general case across a wide range of devices, system implementations, and deep learning models.

4.1 Free-pipeline Fast Inner Product (FFIP)

This section introduces a new inner-product algorithm referred to as the Free-pipeline Fast Inner Product (FFIP) that, in hardware, achieves the same near $2\times$ reduction in required MAC units as FIP [7] while also addressing a key weakness of FIP by inherently improving the clock frequency and, as a consequence, overall throughput of an accelerator for a similar hardware cost compared to FIP.

In FFIP, each element $c_{i,j}$ of $\boldsymbol{C}$ is calculated as follows:

$$c_{i,j} = \sum_{k=1}^{K/2} g^{(j)}_{i,2k-1} \cdot g^{(j)}_{i,2k} - \alpha_i - \beta_j \,, \tag{4.1}$$

where α and β are the same as in Eqs. (2.19) and (2.20), and:

$$g^{(j)}_{i,2k-1} = a_{i,2k} + y_{2k-1,j} \qquad \text{for } j = 1 \tag{4.2a}$$

$$g^{(j)}_{i,2k} = a_{i,2k-1} + y_{2k,j} \qquad \text{for } j = 1 \tag{4.2b}$$

$$g^{(j)}_{i,k} = g^{(j-1)}_{i,k} + y_{k,j} \qquad \text{for } j > 1 \tag{4.2c}$$

$$y_{i,j} = \begin{cases} b_{ij} & \text{for } j = 1 \\ b_{ij} - b_{i,j-1} & \text{for } j > 1 \,. \end{cases} \tag{4.3}$$

The extra subtractions in (4.3) require $\mathcal{O}(NK)$ operations to calculate, which is negligible since the complexity is dominated by the overall $\mathcal{O}(MNK)$ from (4.1). Similarly to (2.18), (4.1) also results in $(MNK + MK + NK)/2$ multiplications for even K and the same number of additions as in (2.22) as well.

The resulting terms being multiplied in (4.1) are identical to those being multiplied in (2.18). The difference between Eqs. (2.18) and (4.1) lies in the terms being fed to the addition in (4.2c). This allows the addition output $g_{i,k}^{(j)}$ in (4.2c) to be passed directly into the adjacent adder for $g_{i,k}^{(j+1)}$, which we later show improves the inner-product's hardware frequency at essentially no cost. Additionally, although each $g_{i,k}^{(j-1)}$ takes $j-1$ iterations to form before being passed to the next $g_{i,k}^{(j)}$, Section 4.2 shows how these iterations for all g terms in the systolic array are each being performed in parallel in a pipeline every clock cycle. Therefore, after an initial latency that is negligible relative to the entire execution time, every matrix product will thereafter be produced in a pipeline at a consistent throughput.

4.1.1 Proof

For $j = 1$, one can substitute the $g_{i,k}^{(j)}$ and $y_{i,j}$ terms for their values specified in Eqs. (4.2a), (4.2b), and (4.3), and observe the same terms as in (2.18). For $j > 1$, (4.1) is proven by first showing that (2.18) can be rewritten as:

$$\begin{aligned} c_{i,j} &= \sum_{k=1}^{K/2} \left([a_{i,2k} + b_{2k-1,j-1}] + [b_{2k-1,j} - b_{2k-1,j-1}]\right) \\ &\qquad \left([a_{i,2k-1} + b_{2k,j-1}] + [b_{2k,j} - b_{2k,j-1}]\right) - \alpha_i - \beta_j \\ &= \sum_{k=1}^{K/2} (h_{i,2k-1}^{(j-1)} + y_{2k-1,j})(h_{i,2k}^{(j-1)} + y_{2k,j}) - \alpha_i - \beta_j \,, \end{aligned} \tag{4.4}$$

where:

$$h_{i,2k-1}^{(j-1)} = a_{i,2k} + b_{2k-1,j-1} \tag{4.5a}$$

$$h_{i,2k}^{(j-1)} = a_{i,2k-1} + b_{2k,j-1} \,, \tag{4.5b}$$

and therefore:

$$h^{(j)}_{i,2k-1} = a_{i,2k} + b_{2k-1,j} \tag{4.6a}$$

$$h^{(j)}_{i,2k} = a_{i,2k-1} + b_{2k,j}\,. \tag{4.6b}$$

Now, $h^{(j)}_{i,2k-1}$ and $h^{(j)}_{i,2k}$ can also be rewritten as:

$$\begin{aligned} h^{(j)}_{i,2k-1} &= (a_{i,2k} + b_{2k-1,j-1}) + (b_{2k-1,j} - b_{2k-1,j-1}) \\ &= h^{(j-1)}_{i,2k-1} + y_{2k-1,j} \end{aligned} \tag{4.7a}$$

$$\begin{aligned} h^{(j)}_{i,2k} &= (a_{i,2k-1} + b_{2k,j-1}) + (b_{2k,j} - b_{2k,j-1}) \\ &= h^{(j-1)}_{i,2k} + y_{2k,j}\,. \end{aligned} \tag{4.7b}$$

Therefore, $h^{(j)}_{i,k} = h^{(j-1)}_{i,k} + y_{k,j}$ and holds the equivalent property as (4.2c), showing that $h^{(j)}_{i,k} = g^{(j)}_{i,k}$ and that (4.4), which is equivalent to (2.18), is equivalent to:

$$\begin{aligned} c_{i,j} &= \sum_{k=1}^{K/2} (g^{(j-1)}_{i,2k-1} + y_{2k-1,j})(g^{(j-1)}_{i,2k} + y_{2k,j}) - \alpha_i - \beta_j \\ &= \sum_{k=1}^{K/2} g^{(j)}_{i,2k-1} \cdot g^{(j)}_{i,2k} - \alpha_i - \beta_j\,, \end{aligned} \tag{4.8}$$

which is equivalent to (4.1).

4.1.2 Deep Learning-Specific Optimizations

For the application of FIP and FFIP to deep learning, a values represent the layer inputs of a DNN layer, b values represent the layer weights, and c values correspond to the layer's output values. The β term in the FIP and FFIP algorithms is a function of the weights. Hence, it remains constant over each inference and can be pre-computed after training. The β terms can also be incorporated into the biases, allowing the β subtraction in (2.18) to be performed during the bias addition stage at no extra cost. Each β_j term is then subtracted from $bias_j$ as follows:

$$bias_j = bias_j - \beta_j \,, \tag{4.9}$$

where each j represents a layer output channel. This allows the computation from (4.1) to be reduced to:

$$c'_{i,j} = \sum_{k=1}^{K/2} g^{(j)}_{i,2k-1} \cdot g^{(j)}_{i,2k} - \alpha_i \,, \tag{4.10}$$

where c' is a layer output before biasing and activation.

Finally, the $y_{i,j}$ values in FFIP from (4.3) are also a function of the layer weights, and can also be pre-computed after training.

4.2 Fast Inner-Product Architectures

4.2.1 Definitions

- w: The bitwidth that the weight and activations are quantized to.

- d: A bitwidth increase applied to certain locations in the FIP and FFIP PE datapaths, where $d = 1$ if a and b are both signed or both unsigned, and $d = 2$ if either a or b is signed while the other is unsigned.

- X, Y: The width and height of an MXU, respectively, in effective number of MAC units. For a baseline MXU, this refers to the the actual width and height in number of MAC units. For FIP and FFIP MXUs, X and Y refer to the MAC width and height required to achieve the same computational power if implemented on a baseline MXU. However, for FIP and FFIP, the number of actual instantiated MAC columns is $X/2$, and the number of actual instantiated MAC rows is $Y + 1$ (where the extra row is for calculating the α terms as explained in Section 4.2.3).

4.2.2 Processing Element (PE) Architectures

Fig. 4.1a shows two traditional PEs which implement the baseline inner-product algorithm as in existing state-of-the-art systolic-array deep learning accelerator architectures [3], [5], [6], [55], [16]. Figs. 4.1b and 4.1c show PE architectures for implementing the FIP and FFIP algorithms in an MXU hardware architecture, respectively. The FIP and FFIP PEs shown in Figs. 4.1b and 4.1c each individually provide the same effective computational power as the combined computational power of the two baseline PEs shown in Fig. 4.1a which implement the baseline inner product.

As can be seen, compared to the baseline PEs, the FIP and FFIP PEs therefore provide the same computational power with half the number of multipliers, where the removed multipliers are instead traded for cheap additions on w or $w + d$ bits. Furthermore, half of the accumulations on $2w + clog_2(X)$ bits are also traded for lower-bitwidth additions on w or $w + d$ bits. Finally, one last benefit is that the MXUs using FIP and FFIP PEs have a

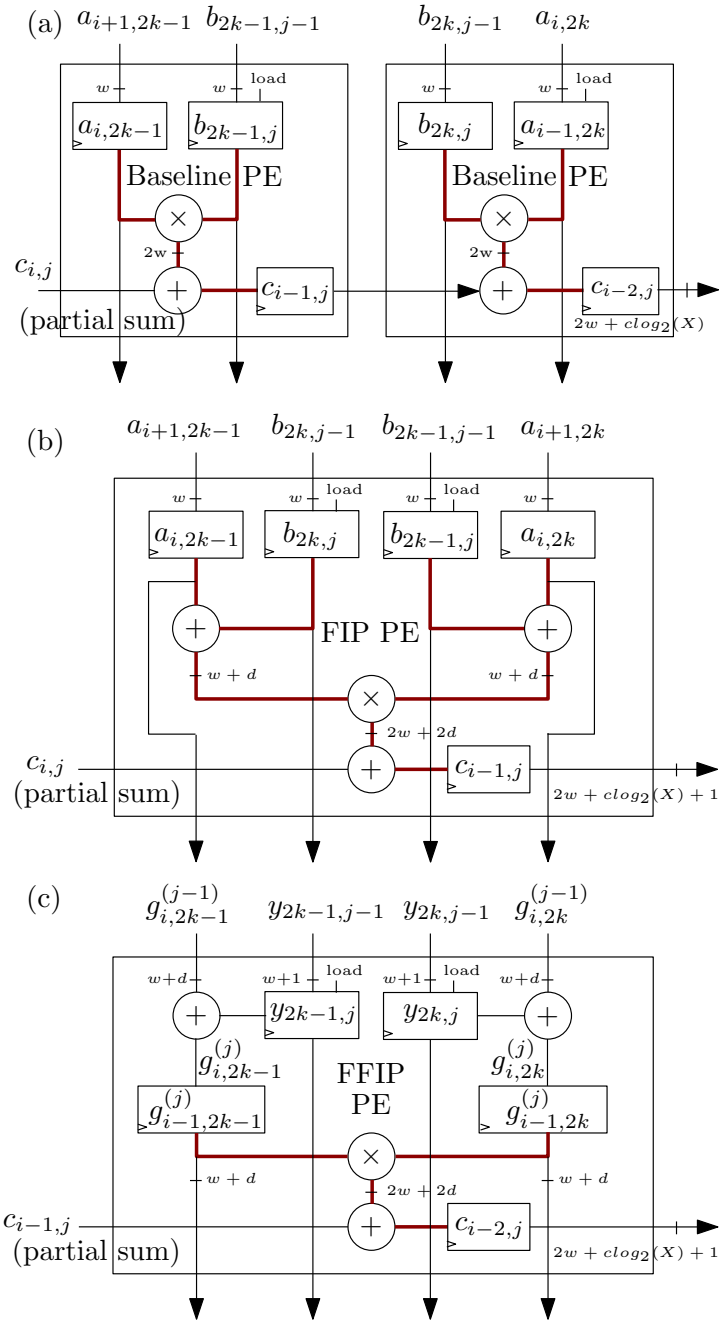

Figure 4.1: The PE architectures for implementing the (a) baseline, (b) FIP, and (c) FFIP inner-product algorithms in hardware. The FIP and FFIP PEs shown in (b) and (c) each individually provide the same effective computational power as the two baseline PEs shown in (a) combined which implement the baseline inner product as in existing systolic-array deep learning accelerators. Critical paths are highlighted.

latency that is $X/2$ fewer clock cycles than a baseline MXU.

The resulting terms being multiplied in Fig. 4.1c are the same as in Fig. 4.1b, however, the addition outputs $g_{i,k}^{(j)}$ (the elements $g_{i-1,2k-1}^{(j)}$ and $g_{i-1,2k}^{(j)}$ in Fig. 4.1c) can now be passed directly into the adder inputs of the next adjacent PE below. This is beneficial because the addition outputs $g_{i,k}^{(j)}$ stored in registers now serve the dual purpose of:

1. Acting as pipelining registers for the addition outputs in (4.2), allowing the $g_{i,k}^{(j)}$ terms to be buffered directly before the multiplication, and

2. Serving as systolic buffers to store the $g_{i,k}^{(j)}$ inputs for the adjacent PE below.

In contrast, the FIP algorithm from (2.18) requires the inputs (a and b in this case) to be stored in systolic buffers prior to addition before passing them to the adjacent PEs below, but doing so does not serve the dual purpose of also acting as pipelining registers before the multiplication. Therefore, the FFIP algorithm inherently results in a higher maximum frequency for a similar hardware cost.

However, in order to facilitate the usage of the FFIP PEs and gain their benefits, it is mandatory to fundamentally change the systolic-array data flow of the original FIP algorithm in a non-obvious way requiring a high-level mathematical understanding of the algebra being performed, and is something that goes beyond the capabilities of today's CAD tools. Our FFIP algorithm defined in Section 4.2.2 provides these required high-level mathematical changes.

FFIP PE versus FIP PE with Additional Registers

After investigating and implementing the FIP algorithm in hardware, a key downside we found was that the FIP implementation results in a reduction in clock frequency and, as

a consequence, throughput in the resulting architecture. As illustrated in Fig. 4.1b, this reduction in clock frequency comes from the fact that the longest path between registers that the logic in the FIP PE has to traverse through is across two adders and one multiplier rather than just one adder and multiplier. While additional registers could be placed at the FIP PE multiplier inputs to match the critical path of the FFIP PE, this adds a significant cost in registers in the PE array compared to using FFIP. The register requirements for each FIP PE would then increase from (4.11) to (4.12):

$$4w + (2w + clog_2(X) + 1) = 6w + clog_2(X) + 1 \tag{4.11}$$

$$2(w + d) + (6w + clog_2(X) + 1) \tag{4.12a}$$

$$= 8w + 2d + clog_2(X) + 1\,, \tag{4.12b}$$

where (4.12) is derived by adding to (4.11) the register sizes that would be required to buffer the multiplier inputs. In contrast, the register requirements for each FFIP PE is:

$$2(w + d) + 2(w + 1) + (2w + clog_2(X) + 1) \tag{4.13a}$$

$$= 6w + 2d + clog_2(X) + 3\,. \tag{4.13b}$$

Fig. 4.2 shows that the FFIP register overhead defined in (4.13) starts to increase more rapidly for bitwidths below 4, which may make it less desirable to use below that width. However, outside of this range, FFIP register requirements are significantly less than an FIP PE with extra registers added to match the critical path/frequency of an FFIP PE.

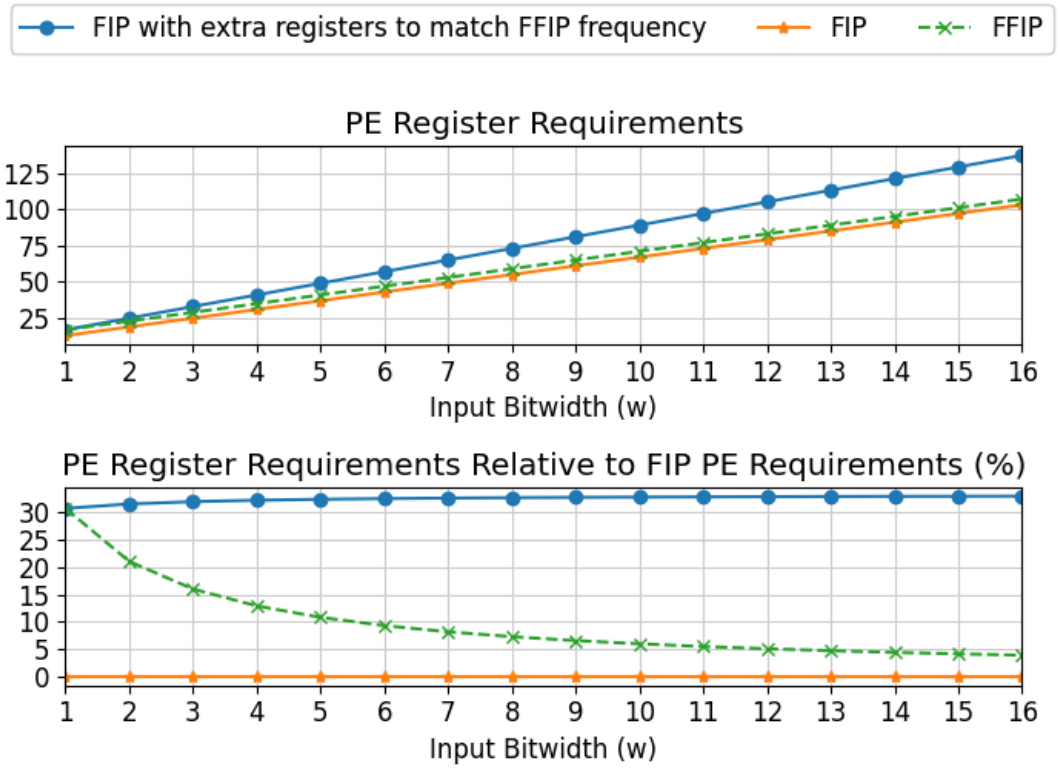

Figure 4.2: PE register requirements at different w bitwidths for the FIP PE, FFIP PE, and FIP PE with extra registers before the multiplier to match the frequency of the FFIP PEs. Values are calculated using Eqs. (4.11) - (4.13) for $X = 64$ and $d = 1$.

Finally, while there will also be other registers to consider in an accelerator outside of the PEs in the datapath and control logic, these register utilizations are highly implementation-specific and will vary depending on the details of the system implementation used to house the systolic array. Furthermore, as the systolic array size increases, the number of PEs increase quadratically and their total register contributions determined by Eqs. (4.11), (4.12), or (4.13) will dominate other registers outside of the systolic array.

4.2.3 Matrix Multiplication Unit (MXU) Architectures

Fig. 4.3 shows how the PEs are laid out into an MXU architecture. The suggested MXU input buffers shown are triangular-shaped register arrays containing X shift registers of varying depths, with each shift register SR_k loading one $a_{i,k}$ or $b_{k,j}/y_{k,j}$ value per clock cycle. The depth of each SR_k is $\lceil k/2 \rceil$ for the FIP and FFIP MXUs, and k for the baseline MXU. In order to perform general matrix multiplication (GEMM) on a MXU, the input matrices are divided into tiles fed to the MXU one-by-one. Following each tile multiplication, the partial tile products are accumulated outside of the MXU to generate each final matrix product tile. Prior to each tile multiplication, a b/y tile is loaded into the MXU. It then remains in place as the a/g tile flows through the MXU producing the tile product, during which a new a_i vector is fed into the MXU each clock cycle. Additionally, to hide the latency of loading b/y tiles, we suggest the MXU contains an extra b/y tile buffer which loads the next tile as the current tile is being multiplied.

4.2.4 Deep Learning-Specific Optimizations

In quantized deep learning inference, the weights and activations can each be quantized to signed or unsigned integers by choosing different zero points to quantize the values with

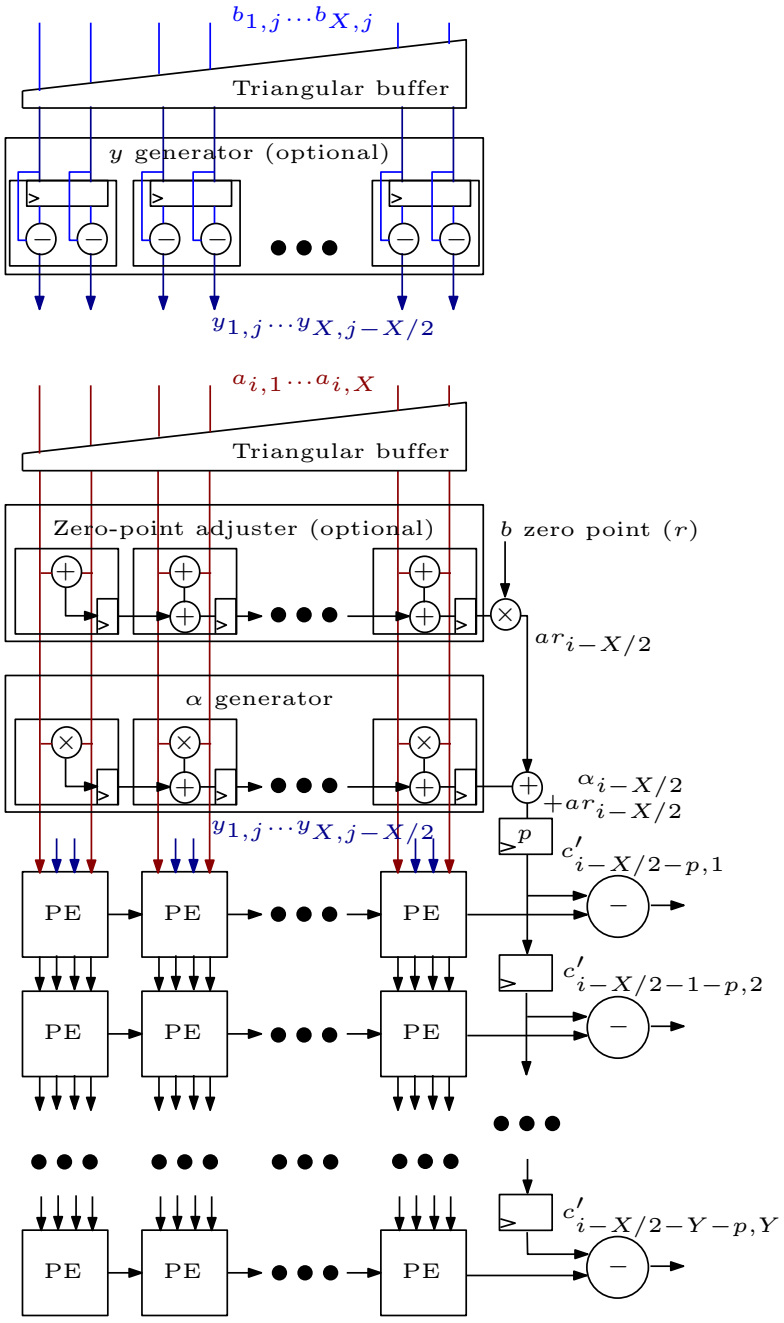

Figure 4.3: The FFIP MXU architecture. The FIP MXU is the same except that FIP PEs are used instead and the y generator block is not present, and b inputs are passed in instead of y inputs. The α terms are calculated and subtracted as shown by first passing the a inputs through an additional row of MAC units before they enter the rest of the MXU.

[20]. However, for the FIP and FFIP architectures it is beneficial to quantize the weights and activations such that they are *both* represented on either signed or unsigned integers. Otherwise, representing one on signed and the other on unsigned adds a penalty in the hardware footprint of the FIP and FFIP PEs because then, due to the possible range of the result, d must equal 2 instead of 1, and the sum of a and b elements must then be represented on $w+2$ bits rather than the $w+1$ bits needed if they were both signed or both unsigned. This would lead to extra register requirements in the FFIP PE and cause multiplication on $w+2$-bit inputs rather than $w+1$ bits for both the FIP and FFIP PEs.

Furthermore, if the employed quantization scheme requires adding a constant offset to the weights to adjust their zero point, the contributions of the zero-point offset in the GEMM product must then be separately calculated and subtracted from the result to eliminate their contribution (for any architecture, not just FIP and FFIP) [20]. We provide a solution to mitigate this penalty in the FIP and FFIP architectures when layer-wise zero-point offsets are used on the weights. We show how to pass some of these extra required computations into the pre-existing α generator logic as shown in Fig. 4.3. To help illustrate how this works, consider that the weight zero-point offsets can be represented by a constant matrix $\boldsymbol{R}$ being added to the weights where each element has a constant value. This results in the MXU performing the following computation:

$$\boldsymbol{A}\left(\boldsymbol{B}+\boldsymbol{R}\right)=\boldsymbol{A}\boldsymbol{B}+\boldsymbol{A}\boldsymbol{R}\,. \tag{4.14}$$

Therefore, to eliminate the contribution of the constant matrix $\boldsymbol{R}$ in the matrix product, the $\boldsymbol{AR}$ product must be subtracted from the MXU output. In order to do this and combine its subtraction with the existing α generator logic, we provide the *zero-point adjuster* block shown in Fig. 4.3, which calculates the $\boldsymbol{AR}$ elements using only one multiplier, and show

how to efficiently combine its $\boldsymbol{AR}$ output elements with the α elements. This way, both the $\boldsymbol{AR}$ and α values are subtracted from the MXU output vectors at the same time and an additional hardware subtraction vector dedicated to subtracting only the $\boldsymbol{AR}$ elements is not required on the MXU output.

Finally, as discussed in Section 4.1.2, the β terms do not need to be calculated as they do for α since they can be pre-computed and added to the biases. Furthermore, the y values can either be calculated in real time with the y generator as shown in Fig. 4.3, or they can be pre-computed at the cost of storing them in 1 extra bit in memory.

4.2.5 Multiplier Compute Efficiency

In this subsection, we define a performance-per-area metric called the multiplier compute efficiency in (4.18c) which we use to compare the FFIP architecture against baseline models and prior works. The metric is used to compare the amount of computational work that can be performed per compute area regardless of the clock frequency.

The hardware complexity of fixed-point multipliers typically scale quadratically with the input bitwidth compared to linearly for adders and registers [71], [25], [72], causing the hardware footprint of multipliers to dominate that of adders and registers. Due to this, multipliers and MAC units are commonly the area-dominant computational resources in deep learning and GEMM-based accelerators [70], [3], [5]. Therefore, we derive a performance-per-area metric defined below for quantifying how much the algebraic optimizations exploited in an architecture reduce the computational complexity of the area-dominant operations (multiplications) and measure how effectively an architecture can utilize these resources relative to a conventional design using no algebraic optimizations.

The multiplier compute efficiency is defined as follows:

$$\frac{\text{mults/multiplier}}{\text{clock cycle}} = \frac{(\text{mults/s})/\#\text{multipliers}}{f}\,, \tag{4.15}$$

where mults/s above is measured by taking the number of multiplications required to carry out an execution using conventional algebra and dividing it by the measured execution time, #multipliers is the number of instantiated multipliers in the design, and f is the clock frequency that the hardware design is operating at.

Now, the metric from (4.15) has the following limit when executing the conventional MM algorithm in hardware:

$$\text{MM}\ \frac{\dfrac{\text{mults}}{\text{multiplier}}}{\text{clock cycle}}\ \text{roof} = 1\,. \tag{4.16}$$

In contrast, the FIP and FFIP algorithms requires half the number of multiplications as MM for the same throughput. Therefore, the multiplier compute efficiency can reach the following limit in (F)FIP architectures:

$$\text{(F)FIP}\ \frac{\dfrac{\text{mults}}{\text{multiplier}}}{\text{clock cycle}}\ \text{roof} = 2\,, \tag{4.17}$$

showing how (F)FIP can increase the throughput roof of accelerators without increasing the area.

Based on the above definitions, we use the following three performance metrics for

judging prior state-of-the-art solutions against FFIP:

$$\text{throughput} \quad (\text{GOPS}) \quad = \quad \text{op/s} \cdot 10^{-9} \tag{4.18a}$$

$$\frac{\text{throughput}}{\text{compute area}} \left(\frac{\text{mults}}{\text{multiplier}} \right) = \frac{\text{mults/s} \cdot 10^{-9}}{\text{\#multipliers}} \tag{4.18b}$$

$$\frac{\text{throughput}}{\dfrac{\text{compute area}}{\text{clock cycle}}} \left(\frac{\dfrac{\text{mults}}{\text{multiplier}}}{\text{clock cycle}} \right) = \frac{\dfrac{\text{mults/s}}{\text{\#multipliers}}}{f} . \tag{4.18c}$$

The *throughput* metric defined in (4.18a) measures the raw performance that a solution is able to achieve on a device and is the foremost commonly used metric for comparison in the field. We use the *throughput per compute area* and *throughput per compute area per clock cycle* metrics defined in Eqs. (4.18b) - (4.18c) to help compare the scaling potential of a design regardless of how many computational resources are available on the device it is scaled onto, or so that prejudice is not given towards implementations that could theoretically achieve higher throughputs but did not scale up their design to take full advantage of all the compute resources available on the device. The *throughput per compute area per clock cycle* metric serves a similar purpose as *throughput per compute area*, however, it also normalizes for clock frequency to further abstract the performance criteria from the implementation platforms and their technology nodes, which will have varying timing potentials, and this metric will also remove any doubt regarding FFIP's performance improvements coming solely from increase in clock frequency compared to the prior works. However, we still include the *throughput per compute area* metric because a key benefit of FFIP is that it *does* inherently allow for a higher clock frequency for a similar hardware cost compared to FIP.

4.3 Results

This section demonstrates the generality of the FIP-based approaches by evaluating example implementations of our (F)FIP architectures for DNN inference as integrated into the the deep learning accelerator system described in Chapter 3. The theoretical benefits of the FIP architecture are validated for the first time in a deep learning accelerator. Additionally, we validate the theoretical benefits of the proposed FFIP architecture against the baseline and FIP architectures, and against state-of-the-art prior works. Although the theoretical concepts presented in this work are general and applicable to both custom integrated circuits and field-programmable gate array (FPGA) implementations, our FFIP algorithm and architecture were validated on FPGA, and we therefore confirm our theoretical insights by comparing the benefits of the proposed FFIP architecture against the best-in-class of prior state-of-the-art deep learning accelerator solutions that are also evaluated on FPGA. We compare designs for non-sparse neural network acceleration and input sizes quantized to 8 - 16 bits, which are bitwidths commonly used in practice as they provide a balanced trade-off between accuracy versus hardware efficiency [20].

Full system-level validation of the (F)FIP MXUs as integrated into the experimental accelerator system from Section 3 has been done on an Arria 10 SoC Developement Kit [92] containing the Arria 10 SX 660 device by measuring model throughput in real-time. However, this device contains fewer soft logic resources than the Arria 10 GX 1150 used in the prior works we compare against, and we generate compilation results in Section 4.3.2 for our design on the same Arria 10 GX 1150 device used in prior works for a more fair and consistent comparison. Throughput values of our designs on the Arria 10 GX 1150 device are then calculated using an accurate throughput estimation model based on our highly deterministic and time-predictable system implementation, which accurately

predicts actual throughputs measured on the Arria 10 SX 660 device available to us. The models used for evaluation are AlexNet [11] and ResNet [17].

4.3.1 FFIP Compared to Baseline and FIP

The baseline, FIP, and FFIP MXUs from Section 4.2.3 have been instantiated inside an example accelerator system used for validation that was discussed in Section 3. Different accelerator designs were compiled where the system architecture remained the same, and only the MXU size and type changed. The MXU height/width was incremented by multiples of 8 from sizes 32 to 80, or until the device no longer contained enough digital signal processing (DSP) units to instantiate the design. For each MXU size, we compiled a baseline, FIP, and FFIP MXU, except for the baseline MXU designs above size 56×56 which no longer fit on the device due to the DSP resources reaching their limit.

The memory resource deviation seen in the 72×72 FFIP MXU in Fig. 4.4 is due to the FPGA compiler placing some of the datapath buffers outside of the MXU into memory resources. However, due to the heuristic nature of FPGA compilers, where even bit-equivalent designs will produce small variations in hardware utilizations for different random seeds, this deviation is interpreted as a random outlier and not something inherent to FFIP.

The FIP architecture uses up to 15-20% more ALMs and registers than the baseline to implement the pre-adders that nearly half the multipliers and accumulators are traded for as discussed in Section 4.2.2. However, as expected from our theoretical analysis, the FIP architecture provides the much more significant **near 2× reduction in DSP units**, which are hard logic resources used to implement MAC units in FPGAs since that operation cannot be mapped efficiently onto the soft logic resources [25]. The clock frequency of the

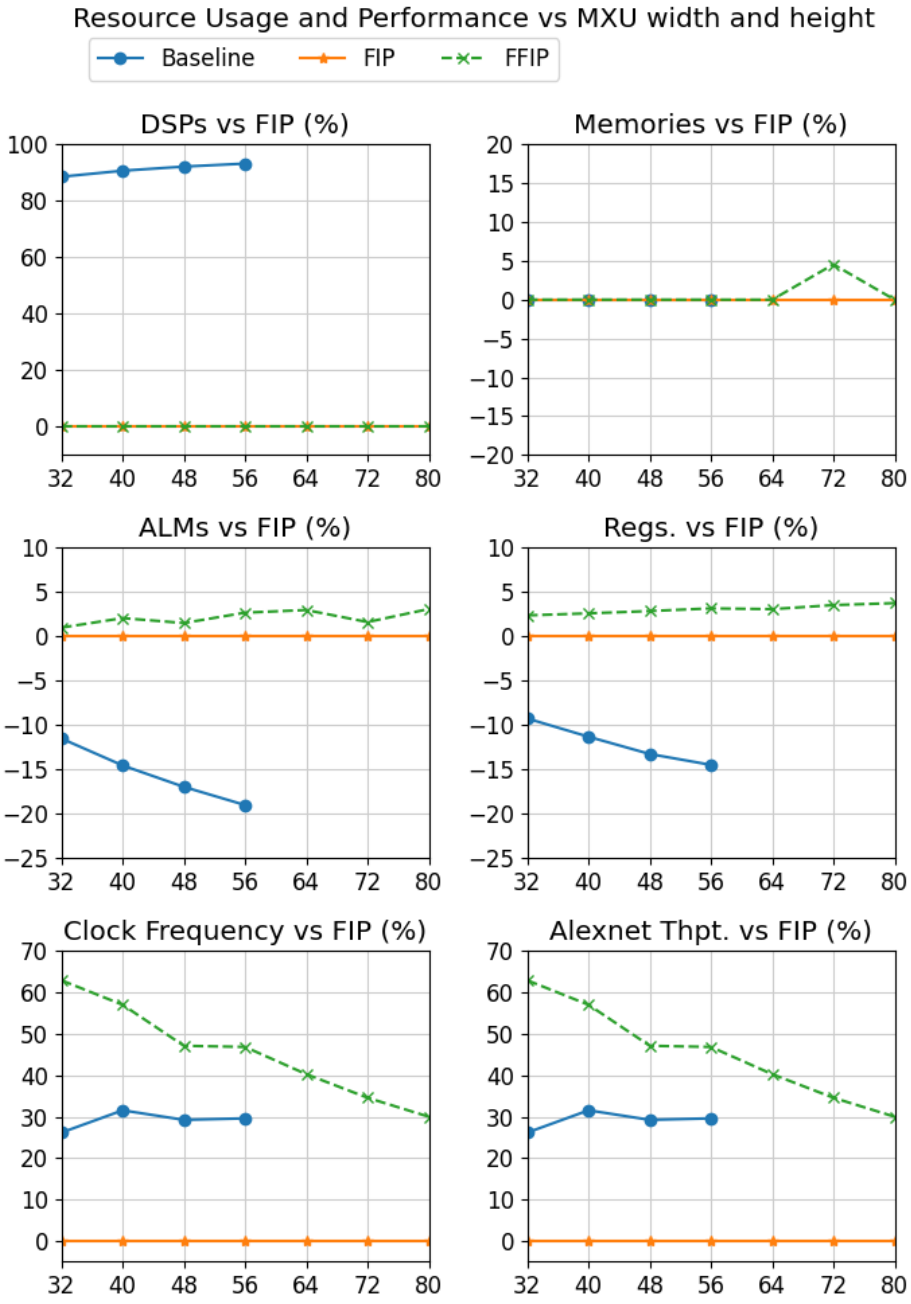

Figure 4.4: Evaluating the baseline, FIP, and FFIP MXUs at different sizes instantiated into an example deep learning accelerator system design used for validation, with 8-bit fixed-point inputs on an Arria 10 SX 660 FPGA.

FIP designs, however, is approximately 30% lower than the clock frequency of the baseline designs, which consequently reduces the throughput by the same ratio. As anticipated from our theoretical analysis, the FFIP architecture inherently addresses this weakness of the FIP architecture for a similar hardware cost by maintaining the same near 2× reduction in the number of required DSPs, however, **FFIP improves the clock frequency by over 30%**, and as a consequence, the overall throughput of the accelerator is improved by the same amount compared to FIP. Additionally, the effective size of the largest systolic array that can be fit onto the device before maxing out the DSPs is increased from 56×56 PEs to 80×80 when using FIP and FFIP, a **2× increase in effective number of PEs**.

Finally, while additional registers could be placed at the FIP PE multiplier inputs to match the critical path of the FFIP PEs, this would come at a significant extra cost in additional registers compared to using FFIP. A detailed analysis for this is provided in Section 4.2.2.

4.3.2 FFIP Compared to the State-of-the-Art on FPGA

Although the theoretical concepts presented in this work are general and applicable to both custom integrated circuits and FPGA implementations, our FFIP algorithm and architecture were validated on FPGA, and we therefore confirm our theoretical insights by comparing the benefits of the proposed FFIP architecture against the best-in-class of prior deep learning accelerator solutions that are also evaluated on FPGA.

For the results in Section 4.3.2, #multipliers is calculated as $\#DSPs$ for AMD/Xilinx FPGAs where each DSP instantiates one 18×27-bit multiplier [93], and $\#DSPs \times 2$ for Intel/Altera FPGAs where each DSP instantiates two 18×19-bit multipliers [94]. The only exception to this is for the works using Intel/Altera FPGAs from Liu et al. [95] and Fan et

al. [96], where two 8-bit multiplications are packed onto each 18×19-bit multiplier in the DSPs and additional ALMs as explained in Section 4.3.2, and therefore $\#multipliers = \#DSP \times 4$.

We also note that, when evaluating solutions on FPGA, it is more common in the field to measure the performance per compute area using *throughput/DSP* rather than *throughput/multiplier*. However, we use the latter in order to make the evaluation more generalized and to detach it from the specific DSP architectures currently being used in modern FPGAs. For example, the DSPs on most AMD/Xilinx FPGAs [93] contain one fixed-point multiplier, whereas the DSPs on most Intel/Altera FPGAs [94] contain two fixed-point multipliers, which would make GOPS/DSP comparisons unfair when comparing designs on Intel versus AMD FPGAs, and would decrease the generality of the comparison.

As derived in Section 4.2.5, (F)FIP increases the theoretical roof of the performance metrics defined in Eqs. (4.18a) - (4.18c) in the general case, regardless of the implementation technology. To empirically validate this, we compare our results in this section against the prior state-of-the-art, and since our FFIP algorithm and generalized architecture were validated on FPGA, we confirm our theoretical insights by comparing the benefits of the proposed FFIP architecture against the best-in-class prior solutions also evaluated on FPGA. We demonstrate that FFIP is extremely competitive and surpasses the prior works in the evaluated performance metrics.

Due to the clear advantages, increasing the theoretical limits of the performance metrics defined in Eqs. (4.18a) - (4.18c) from Section 4.2.5 has been focused on recently in other works as well. The works by Yepez et al. [66] and Jiang et al. [99] do this by exploiting Winograd's minimal filtering algorithms applied to convolutional neural networks as

Table 4.1: Comparison with state-of-the-art 8-bit-input accelerators for different models on the same FPGA family.

	TNNLS '22 [95]		TCAD '22 [96]		Entropy '22 [97]		Ours (FFIP 64×64)		
FPGA	Arria 10 GX 1150		Arria 10 GX 1150		Arria 10 GX 1150		Arria 10 GX 1150		
Data type	8-bit fixed		8-bit fixed		8-bit fixed		8-bit fixed		
ALMs	304K		304K		303K		118K		
Registers	889K		890K		-		311K		
Memories	2334		2334		1953		1782		
DSPs	1473		1473		1503		1072		
Frequency (MHz)	200		220		172		388		
Model	ResNet-50	VGG 16	Bayes ResNet-18	Bayes VGG 11	R-CNN (ResNet-50)	R-CNN (VGG 16)	ResNet-50	ResNet-101	ResNet-152
GOPS [1]	1519	1295	1590	534	719	865	2529	2752	2838
$\frac{\text{mults}}{\text{multiplier}}$ [2]	0.129	0.110	0.135	0.045	0.120	0.144	0.590	0.642	0.662
$\frac{\frac{\text{mults}}{\text{multiplier}}}{\text{clock cycle}}$ [3]	0.645	0.549	0.613	0.206	0.695	0.837	1.521	1.655	1.707

[1] Throughput, defined in (4.18a), motivation and explanation provided in Section 4.2.5.
[2] Throughput per compute area, defined in (4.18b), motivation and explanation provided in Section 4.2.5.
[3] Throughput per compute area per clock cycle, defined in (4.18c), motivation and explanation provided in Section 4.2.5.

Table 4.2: Comparison with state-of-the-art 16b-bit-input accelerators for different models on the same FPGA family.

	TCAD '20 [98]			TVLSI '20 [66]		TCAS-II '22 [99]	TCAS-I '23 [100]	Ours (FFIP 64×64)		
FPGA	Arria 10 GX 1150			Arria 10		Arria 10 GX 1150	Arria 10 SoC	Arria 10 GX 1150		
Data type	16-bit fixed			16-bit fixed		8/16-bit fixed [4]	16-bit fixed	16-bit fixed		
ALMs	286K	335K	208K	181K	180K	-	189K	199K		
Registers	-	-	-	-	-	-	-	530K		
Memories	2356	2692	2319	1310	1310	1565	-	2713		
DSPs	1518	1518	1518	1344	1344	1161	1536	1072		
Frequency (MHz)	240	240	240	250	250	163	200	346		
Model	ResNet-50	ResNet-152	VGG 16	VGG16	Modified VGG16	CTPN(VGG +Bi LSTM)	Modified StyleNet	ResNet-50	ResNet-101	ResNet-152
GOPS [1]	600	697	968	1642	1788	1224	670	2258	2458	2534
$\frac{\text{mults}}{\text{multiplier}}$ [2]	0.099	0.115	0.159	0.305	0.333	0.264	0.109	0.527	0.573	0.591
$\frac{\frac{\text{mults}}{\text{multiplier}}}{\text{clock cycle}}$ [3]	0.412	0.478	0.664	1.222 [5]	1.330 [5]	1.617 [5,6]	0.545	1.521	1.655	1.707

[1-3] See the corresponding definitions from Table 4.1.
[4] Weights and layer outputs are quantized to 8 bits. Layer input is quantized to 16-bit due to Winograd convolutional transformations.
[5] These works use Winograd's minimal filtering algorithms [65] to reduce multiplication complexity.
[6] This is a central processing unit (CPU)-FPGA heterogeneous work where portions of the inference are performed on CPU.

Table 4.3: Comparison with state-of-the-art accelerators on different FPGAs for the same models and input bitwidths.

	TVLSI '19 [101]	TCAS-II '21 [102]	Ours (FFIP 64×64)	TNNLS '22 [95]	TCAS-I '23 [103]	Ours (FFIP 64×64)	TCAD '20 [98]	Ours (FFIP 64×64)	TNNLS '22 [104]	Ours (FFIP 64×64)	TCAD '20 [98]	Ours (FFIP 64×64)
Model	AlexNet	AlexNet	AlexNet	ResNet-50	ResNet-50	ResNet-50	ResNet-50	ResNet-50	ResNet-101	ResNet-101	ResNet-152	ResNet-152
Data type	16-bit fixed	8/16-bit fixed [4]	16-bit fixed	8-bit fixed	8-bit fixed	8-bit fixed	16-bit fixed	16-bit fixed	8/16-bit fixed [5]	16-bit fixed	16-bit fixed	16-bit fixed
FPGA	XC7VX 690T	VC709	Arria 10 GX 1150	Arria 10 GX 1150	XCV U9P	Arria 10 GX 1150	Arria 10 GX 1150	Arria 10 GX 1150	VX980	Arria 10 GX 1150	Arria 10 GX 1150	Arria 10 GX 1150
ALMs (Intel) / LUTs (AMD)	468K	121K	199K	304K	-	118K	286K	199K	480K	199K	335K	199K
Registers	649K	160K	530K	889K	-	311K	-	530K	-	530K	-	530K
Memories (20Kb Intel) / (36Kb AMD)	1465	1470	2713	2334	-	1782	2356	2713	1457	2713	2692	2713
DSPs	1436	664	1072	1473	2048	1072	1518	1072	3121	1072	1518	1072
Frequency (MHz)	200	200	346	200	200	388	240	346	100	346	240	346
GOPS [1]	434	220	1974	1519	287	2529	600	2258	600	2458	697	2534
$\frac{\text{mults}}{\text{multiplier}}$ [2]	0.151	0.166	0.460	0.129	0.070	0.590	0.099	0.527	0.096	0.573	0.115	0.591
$\frac{\frac{\text{mults}}{\text{multiplier}}}{\text{clock cycle}}$ [3]	0.756	0.829	1.330	0.645	0.351	1.521	0.412	1.521	0.961	1.655	0.479	1.707

[1-3] See the corresponding definitions from Table 4.1.
[4] Weights and layer inputs are quantized to 8 and 16 bits, respectively.
[5] Weights are quantized to 8 bits and layer input/output is quantized to 8 or 16 bits at different stages.

shown by the work from Lavin et al. [65]. However, as shown in Table 4.2, the FFIP architectures surpass the prior works in effective ability to reduce the multiplication complexity of the neural network workloads, shown by comparing the *operations/multiplier/clock cycle* metric defined in (4.18c). Furthermore, Winograd's minimal filtering algorithms are applicable only to convolutional layers, whereas FIP and FFIP are applicable to all deep learning models and layer types that mainly decompose to matrix multiplication, such as fully-connected, convolutional, recurrent, and attention/transformer layers. Finally, the Winograd convolution technique [65] still results in matrix multiplication, which can therefore still achieve further compute efficiency improvements by also executing the resulting matrix multiplication on a systolic-array architecture housing FFIP PEs.

Additionally, the Arria 10 DSPs [94] can each perform two multiplications between one 18-bit and one 19-bit integer input. As demonstrated in the works by Liu et al. [95] and Fan et al. [96], it is possible to pack the multiplier inputs such that each 18×19-bit multiplier performs two multiplications of 6-bit inputs. The remaining two 2×8-bit multiplications and two 2×6-bit multiplications required are then performed and summed using extra ALMs. The works by Liu et al. [95] and Fan et al. [96] use this technique to improve the DSP efficiency for 8-bit inputs. However, this technique requires a noticeable extra cost in ALMs, it is specific to the particular DSP architecture currently being used in Intel/Altera FPGAs [94], and it does not work for 16-bit inputs. We also note that the works from Liu et al. [95] and Fan et al. [96] report the total resources available on the FPGA in their comparison with prior work, however, we list their total instantiated resources that are also reported in their work in our comparison since it is the more common practice and is more favourable for them to use in the comparison here.

We also note that, while our system implementation required more FPGA memory

resources than some of the other works, the FFIP architecture proposed in this chapter affects only to the systolic array of an accelerator and does not consume any of the memory resources itself. The memory subsystem design used in our example accelerator implementation that is consuming the majority of the FPGA memory resources was used to ensure that off-chip memory bandwidth was not a bottleneck on the Arria 10 SoC Developement Kit available to us in order to ensure we could properly evaluate the benefits of FFIP on that platform. FFIP can be used in any accelerator system that uses traditional fixed-point systolic arrays for the arithmetic without fundamentally altering the accelerator's functionality or internal interfaces in any way, and its usage is orthogonal to the memory subsystem being used in the accelerator.

To make comparisons as fair as possible, we aligned as many variables as possible to the best of our ability in the comparisons in Tables 4.1 - 4.3 in regards to making comparisons against implementations on the same FPGA, the same input/datapath bitwidths, and the same deep learning models. However, since it is not feasible in implementation time to evaluate any one solution for all deep learning models on all platforms, Tables 4.1 and 4.2 compare prior works on the same FPGA family and the same datapath/input bitwidths but for varying deep learning models. However, since it was not possible to find any more recent prior works evaluated on Arria 10 devices that also evaluate identical deep learning models as we did, we also provide Table 4.3, which compares prior works sometimes evaluated on different FPGAs than our validation platform, but for identical deep learning models and identical or similar input/datapath bitwidths.

Despite the strategies used in some prior works for increasing the theoretical limits of the performance metrics defined in Eqs. (4.18a) - (4.18c), the results from Tables 4.1 - 4.3, demonstrate that the proposed FFIP approach is the most effective at doing so and

can provide performance improvements against the best-in-class prior works in almost all evaluated performance metrics and every evaluated comparison criteria. When comparing the lowest to highest performing models evaluated on FFIP for *throughput per compute area per clock cycle* against the next-most competitive result from the prior works in each table, FFIP is approximately 1.6-2× higher in Table 4.1, it is overall on-par in Table 4.2 (even when including the CPU-FPGA heterogeneous system in the comparison [99]), and 1.6-3.7× higher in Table 4.3. Furthermore, once we account for the higher clock frequency achieved due to the inherent hardware-efficient nature of FFIP, it is further improved over the next-most competitive works in each table, achieving approximately a 3.7-4.6× improvement in Table 4.1, a 1.4-1.8× improvement in Table 4.2, and a 2.8-6× improvement in Table 4.3. Finally, our FFIP implementation allowed us to achieve the highest overall throughput over the next-most competitive prior works in each table, achieving approximately a 1.4-1.8× improvement in Table 4.1, a 1.1-1.4× improvement in Table 4.2, and a 1.7-4.6× improvement in Table 4.3, allowing us to increase the throughput limits of the evaluation device well beyond its theoretical throughput limits.

4.4 Summary

We present an algorithm and general architecture that improve Winograd's under-explored inner-product algorithm [7] that can be seamlessly incorporated into any deep learning accelerator system that uses traditional fixed-point systolic arrays to double the throughput per MAC unit, significantly increasing the accelerator's performance per compute area across all deep learning models that will execute on the systolic array.

We implement and evaluate FIP for the first time in a deep learning accelerator system. We then identify a weakness of FIP and propose the new FFIP algorithm and generalized

hardware architecture that inherently address that weakness in the general case. We provide deep learning-specific optimizations for the FIP and FFIP algorithms and systolic-array hardware architectures. We derive how the (F)FIP architectures increase the theoretical compute efficiency and performance limits in the general case.

Finally, although the theoretical concepts presented in this chapter are general and applicable to both custom integrated circuits and FPGA implementations, since our proposed FFIP algorithm and architecture were validated on FPGA, we empirically confirm our theoretical insights through comparison with prior state-of-the-art solutions also evaluated on FPGA. As anticipated from our theoretical analysis, our full-system validation results shown in Fig. 4.4 and Tables 4.1 - 4.3 confirm that the generalized FFIP systolic-array architecture we propose, when implemented, does in-fact surpass the traditional theoretical compute efficiency and performance limits, and can improve performance compared to the best-in-class prior state-of-the-art solutions in almost all evaluated performance metrics, each evaluated comparison criteria, and all evaluated deep learning models. Most importantly, our results indicate that, when overlaid on top of the most efficient systolic-array systems used in practice, FFIP can further increase compute efficiency in the general case across a wide range of devices, system implementations, and deep learning models.

Chapter 5

Karatsuba Matrix Multiplication Algorithm and Hardware Architectures

In 1962, Karatsuba proposed one of the first scalar multiplication algorithms asymptotically faster than the traditional approach [9], which can theoretically be used to reduce the complexity of integer multiplication. However, the extra additions it introduces can increase its execution speed in general-purpose computers or limit its area reduction in custom multiplier circuits for smaller integers of more commonly-used bitwidths [77], [78].

In this work, we show how the scalar Karatsuba multiplication algorithm can be extended to integer matrix multiplication, after which the impact and complexity of the extra additions is reduced. Furthermore, we investigate and present new fixed-precision and precision-scalable hardware architectures for efficiently exploiting the Karatsuba algorithm extended to matrix multiplication (referred to as Karatsuba matrix multiplication or KMM), showing how the proposed algorithm and hardware architectures can provide real area or execution time reductions for integer matrix multiplication compared to scalar Karatsuba or conventional matrix multiplication.

The proposed architectures can also be implemented using proven systolic array and conventional multiplier architectures at their core, maintaining all the implementation benefits of these architectures. Systolic arrays, which we will also refer to as matrix multiplication units (MXU)s for convenience, are an effective choice for use in GEMM accelerators as they significantly reduce the required memory traffic and can reach high clock frequencies due to their short and regular interconnects. Systolic-array architectures have been used in state-of-the-art GEMM and deep learning accelerators such as the Tensor Processing Unit (TPU) [3], [5], [6], among others [8], [16].

In summary, our key contributions in this chapter are the following:

- We propose the Karatsuba matrix multiplication (KMM) algorithm and carry out a complexity analysis of the algorithm compared to conventional scalar Karatsuba and matrix multiplication algorithms to facilitate further future investigations of potential applications and hardware implementations of KMM. We also identify complexity shortcomings of KMM that restrict its benefits in hardware and show how this is mitigated when KMM is combined with an alternative accumulation algorithm.
- We present a new family of hardware architectures for efficiently exploiting KMM in custom hardware. We then model the area or execution time benefits of the KMM architectures and evaluate the proposed architectures both in isolation and in an end-to-end accelerator system compared to baseline designs and prior state-of-the-art works implemented on the same type of compute platform.

5.1 Notation

We use the following notation throughput this chapter:

- $\text{ALG}_n^{[w]}$: An algorithm that operates on w-bit scalars or matrices with w-bit elements, where each scalar or matrix element is divided into n digits. For example, $\text{SM}_2^{[8]}$ represents a scalar multiplication (SM) algorithm for operating on 8-bit 2-digit numbers where each digit is 4 bits wide, such as the multiplication between the hexadecimal values $0\text{x}12 \times 0\text{x}10 = 0\text{x}120$.
  - ALG_n or ALG: The algorithm acronym may also be specified without the subscript $_n$ and/or superscript $^{[w]}$ when the number of digits and/or input bitwidths are not directly relevant for the current context, and it may refer to the use of the algorithm for any value of n or w for each missing subscript and/or superscript.
- $\text{OPERATION}^{[w]}$: An arithmetic operation that works with w-bit values. For example, $\text{MULT}^{[w]}$, $\text{ADD}^{[w]}$, $\text{ACCUM}^{[w]}$ represent a multiplication, addition, and accumulation of w-bit values, respectively, and $\text{SHIFT}^{[w]}$ represents a left or right shift by w bits.
- $\text{x}^{[a:b]}$: The value contained in bits a down to b of a scalar x. For example, the value of bits 7 down to 4 in the hexadecimal number $0\text{x}AE$ is equal to $0\text{x}A$ and is written as $0\text{x}AE^{[7:4]} = 0\text{x}A$. Similarly, $0\text{x}AE^{[3:0]} = 0\text{x}E$.
- $C(\text{ALG}_n^{[w]})$: The complexity of algorithm ALG in number of w-bit multiplications, additions, accumulations, and shift operations.
- $C(\text{ALG}_n)$: The complexity of algorithm ALG in number of arithmetic operations.
- r: The number of recursion levels implemented in KSM or KMM, equal to $\lceil \log_2 n \rceil$.
- d: The height and width of two matrices being multiplied.

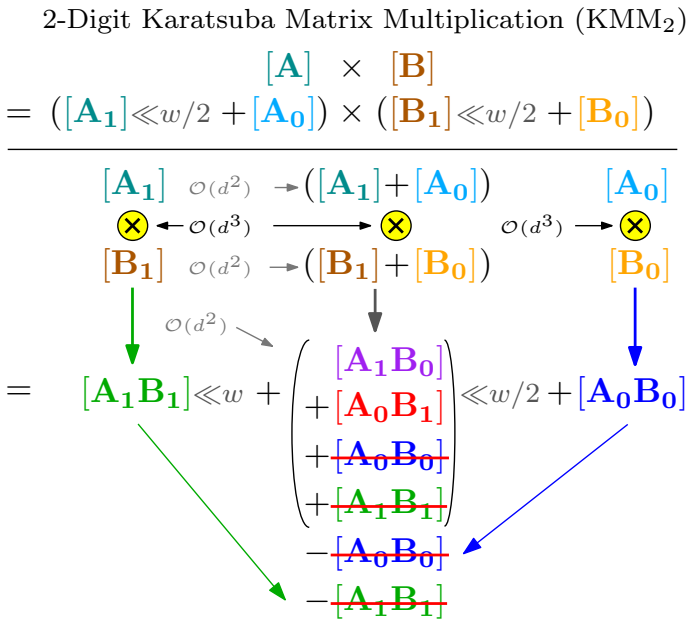

Figure 5.1: KMM_2 algorithm illustration. Compared to the scalar algorithms KSM_2 versus SM_2, the increase in number of additions with complexity $\mathcal{O}(d^2)$ in KMM_2 versus MM_2 is now insignificant relative to the reduction of 3 instead of 4 single-digit matrix multiplications of complexity $\mathcal{O}(d^3)$, allowing the overall #operations in KMM_2 to be less than conventional MM_2.

5.2 Karatsuba Matrix Multiplication (KMM)

In this section, we formally define KMM, analyze its complexity compared to conventional scalar Karatsuba and matrix multiplication algorithms, identify complexity shortcomings of the KMM algorithm that restrict its benefits in hardware, and show how this is mitigated when combining KMM with an alternative accumulation algorithm.

5.2.1 KMM Definition

Fig. 5.1 shows the 2-digit Karatsuba scalar multiplication algorithm [9] from Fig. 2.5 extended to matrix multiplication analogously to how Fig. 2.6 extends conventional 2-digit

Algorithm 5 n-Digit Karatsuba Matrix Multiplication.

function $\mathrm{KMM}_n^{[w]}(\mathbf{A}, \mathbf{B})$
  if $(n > 1)$ **then**
    $\mathbf{A_1} = \begin{bmatrix} a_{1,1}^{[w-1:\lceil w/2 \rceil]}, & \ldots & a_{1,K}^{[w-1:\lceil w/2 \rceil]} \\ \ldots & \ldots & \ldots \\ a_{M,1}^{[w-1:\lceil w/2 \rceil]}, & \ldots & a_{M,K}^{[w-1:\lceil w/2 \rceil]} \end{bmatrix}$
    $\mathbf{A_0} = \begin{bmatrix} a_{1,1}^{[\lceil w/2 \rceil-1:0]}, & \ldots & a_{1,K}^{[\lceil w/2 \rceil-1:0]} \\ \ldots & \ldots & \ldots \\ a_{M,1}^{[\lceil w/2 \rceil-1:0]}, & \ldots & a_{M,K}^{[\lceil w/2 \rceil-1:0]} \end{bmatrix}$
    $\mathbf{B_1} = \begin{bmatrix} b_{1,1}^{[w-1:\lceil w/2 \rceil]}, & \ldots & b_{1,N}^{[w-1:\lceil w/2 \rceil]} \\ \ldots & \ldots & \ldots \\ b_{K,1}^{[w-1:\lceil w/2 \rceil]}, & \ldots & b_{K,N}^{[w-1:\lceil w/2 \rceil]} \end{bmatrix}$
    $\mathbf{B_0} = \begin{bmatrix} b_{1,1}^{[\lceil w/2 \rceil-1:0]}, & \ldots & b_{1,N}^{[\lceil w/2 \rceil-1:0]} \\ \ldots & \ldots & \ldots \\ b_{K,1}^{[\lceil w/2 \rceil-1:0]}, & \ldots & b_{K,N}^{[\lceil w/2 \rceil-1:0]} \end{bmatrix}$
    $\mathbf{A_s} = \mathbf{A_1} + \mathbf{A_0}$
    $\mathbf{B_s} = \mathbf{B_1} + \mathbf{B_0}$
    $\mathbf{C_1} = \mathrm{KMM}_{n/2}^{[\lfloor w/2 \rfloor]}(\mathbf{A_1}, \mathbf{B_1})$
    $\mathbf{C_s} = \mathrm{KMM}_{n/2}^{[\lceil w/2 \rceil+1]}(\mathbf{A_s}, \mathbf{B_s})$
    $\mathbf{C_0} = \mathrm{KMM}_{n/2}^{[\lceil w/2 \rceil]}(\mathbf{A_0}, \mathbf{B_0})$
    $\mathbf{C} = \mathbf{C_1} \ll w$
    $\mathbf{C} \mathrel{+}= (\mathbf{C_s} - \mathbf{C_1} - \mathbf{C_0}) \ll \lceil w/2 \rceil$
    $\mathbf{C} \mathrel{+}= \mathbf{C_0}$
  else
    $\mathbf{C} = \mathrm{MM}_1^{[w]}(\mathbf{A}, \mathbf{B})$
  end if
  return $\mathbf{C}$
end function

scalar multiplication in Fig. 2.5 to matrix multiplication. Algorithm 5 shows the generalization of this, where n-digit Karatsuba matrix multiplication is performed by carrying out the same steps recursively for each smaller-bit matrix multiplication. In Algorithm 5, the full matrix product is split into three separate partial-product matrix multiplications between matrices each containing bit slices of every element. The elements in matrices $\mathbf{A_0}$ and $\mathbf{B_0}$ contain the lower bits (bits $\lceil w/2 \rceil - 1$ down to 0) of every element in the $\mathbf{A}$ and $\mathbf{B}$ matrices, while $\mathbf{A_1}$ and $\mathbf{B_1}$ contain the upper bits (bits $w-1$ down to $\lceil w/2 \rceil$) of every element in matrices $\mathbf{A}$ and $\mathbf{B}$. The $\mathbf{A_s}$ and $\mathbf{B_s}$ matrices are formed by summing $\mathbf{A_1} + \mathbf{A_0}$ and $\mathbf{B_1} + \mathbf{B_0}$, and therefore their elements have a bitwidth of $\lceil w/2 \rceil + 1$. The partial-product matrices are then summed analogously to how the partial scalar products are summed after multiplication in KSM from Algorithm 2.

5.2.2 KMM Complexity Analysis

In this subsection, we derive the complexity of KMM and compare it to the complexity of the conventional MM, and KSM algorithms. To do this, we decompose each algorithms' complexity to number of w-bit multiplications, additions, and shift operations. This provides a general technology-agnostic foundation for evaluating different possible KMM hardware implementations and modelling the costs and benefits of implementing the algorithm in hardware across different possible implementation technologies where the cost of each type of operation may vary depending on the implementation platform used. For example, implementations on FPGA may result in multipliers mapping to DSP units, additions and accumulations mapping to soft look-up-table (LUT) and register resources, whereas ASIC implementations will result in different costs and trade-offs than this for each type of operation.

Additionally, while the main focus of this work is on leveraging KMM in custom hardware designs, we also compare KMM's complexity more simply in number of arithmetic operations to allow modelling the time complexity of KMM execution on general-purpose hardware containing fixed operator word sizes. This analysis (plotted in Fig. 5.2) indicates that KMM requires significantly fewer operations to execute large-integer matrix multiplication on general-purpose hardware than conventional KSM or MM algorithms. This is relevant when the matrix element bitwidths are larger than the word size of the general-purpose hardware operators, for example, inputs larger than 32 bits when executing on a CPU containing arithmetic logic units (ALU)s that support 32-bit inputs.

MM Complexity

The complexity of conventional n-digit MM between two matrices of size $d \times d$ is derived by counting the number of operations that are performed in Algorithm 3:

$$\begin{aligned} C(\text{MM}_n^{[w]}) = {} & C(\text{MM}_{n/2}^{[\lfloor w/2 \rfloor]}) + 3\, C(\text{MM}_{n/2}^{[\lceil w/2 \rceil]}) \\ & + d^2 \left(\text{ADD}^{[w+w_a]} + 2\, \text{ADD}^{[2w+w_a]}\right) \\ & + d^2 \left(\text{SHIFT}^{[w]} + \text{SHIFT}^{[\lceil w/2 \rceil]}\right) \qquad (5.1a) \\ C(\text{MM}_1^{[w]}) = {} & d^3 \left(\text{MULT}^{[w]} + \text{ACCUM}^{[2w]}\right) . \qquad (5.1b) \end{aligned}$$

Typically, $\text{ACCUM}^{[2w]} = \text{ADD}^{[2w+w_a]}$, where w_a is an additional bitwidth added to account for accumulation. However, in Section 5.2.3, we discuss a method for reducing the complexity of the accumulations to be less than this.

The $\text{ADD}^{[w+w_a]}$ terms in (5.1a) come from the additions forming the $(\mathbf{C_{10}} + \mathbf{C_{01}})$ term on line 12 of Algorithm 3. Here, the bitwidth of the $\mathbf{C_{10}}$ and $\mathbf{C_{01}}$ elements is $w + w_a$

because they are accumulations of w-bit products of $\lfloor w/2 \rfloor$ and $\lceil w/2 \rceil$-bit values. The two $\text{ADD}^{[2w+w_a]}$ terms in (5.1a) come from the additions to $\mathbf{C}$ on lines 12 and 13 of Algorithm 3. The bitwidth of these additions is kept on $2w + w_a$ bits since $\mathbf{C}$ results in accumulations of $2w$-bit products of w-bit values.

KSM Complexity

The complexity of KSM is derived by counting the operations performed in Algorithm 2:

$$\begin{aligned} C(\text{KSM}_n^{[w]}) = 2 \ &\left(\text{ADD}^{[2w]} + \text{ADD}^{[\lceil w/2 \rceil]} + \text{ADD}^{[2\lceil w/2 \rceil + 4]}\right) \\ &+ \text{SHIFT}^{[w]} + \text{SHIFT}^{[\lceil w/2 \rceil]} \\ &+ C(\text{KSM}_{n/2}^{[\lfloor w/2 \rfloor]}) + C(\text{KSM}_{n/2}^{[\lceil w/2 \rceil + 1]}) \\ &+ C(\text{KSM}_{n/2}^{[\lceil w/2 \rceil]}) \qquad \text{(5.2a)} \\ C(\text{KSM}_1^{[w]}) = \ &\text{MULT}^{[w]} \,. \qquad \text{(5.2b)} \end{aligned}$$

The two $\text{ADD}^{[\lceil w/2 \rceil]}$ terms in (5.2a) come from the $\lceil w/2 \rceil$-bit additions forming the a_s and b_s terms on lines 7 and 8 of Algorithm 2. The two $\text{ADD}^{[2\lceil w/2 \rceil + 4]}$ terms in (5.2a) come from forming the $(c_s - c_1 - c_0)$ term on line 13 of Algorithm 2, where these terms can be first summed together on $2\lceil w/2 \rceil + 4$ bits before being shifted and added to the other product terms. The bitwidth $2\lceil w/2 \rceil + 4$ is required because c_s is a $(2\lceil w/2 \rceil + 2)$-bit product of $(\lceil w/2 \rceil + 1)$-bit values, and the additional two bits are to account for sign extension and subtraction of the c_1 and c_0 terms. The two $\text{ADD}^{[2w]}$ terms in (5.2a) come from the additions to c on lines 13 and 14 of Algorithm 2. These additions are on $2w$-bit values since c will ultimately result in the $2w$-bit product of two w-bit values.

To compare KSM to KMM and the other matrix multiplication algorithms, we analyze

the complexity of an algorithm we will refer to as KSMM which performs conventional matrix multiplication as in (2.17), except KSM is used for the multiplications between all elements rather than conventional scalar multiplication. KSMM then has the following complexity:

$$C(\text{KSMM}_n^{[w]}) = d^3 \left(C(\text{KSM}_n^{[w]}) + \text{ACCUM}^{[2w]} \right) . \tag{5.3}$$

KMM Complexity

The complexity of KMM is derived by counting the operations performed in Algorithm 5:

$$\begin{aligned} C(\text{KMM}_n^{[w]}) = \; & 2\, d^2 \left(\text{ADD}^{[2\lceil w/2 \rceil + 4 + w_a]} + \; \text{ADD}^{[2w + w_a]} \right) \\ & + d^2 \left(2\, \text{ADD}^{[\lceil w/2 \rceil]} + \text{SHIFT}^{[w]} + \text{SHIFT}^{[\lceil w/2 \rceil]} \right) \\ & + C(\text{KMM}_{n/2}^{[\lfloor w/2 \rfloor]}) + C(\text{KMM}_{n/2}^{[\lceil w/2 \rceil + 1]}) \\ & + C(\text{KMM}_{n/2}^{[\lceil w/2 \rceil]}) \end{aligned} \tag{5.4a}$$

$$C(\text{KMM}_1^{[w]}) = C(\text{MM}_1^{[w]}) . \tag{5.4b}$$

The two $\text{ADD}^{[\lceil w/2 \rceil]}$ terms in (5.4a) come from the $\lceil w/2 \rceil$-bit additions forming the $\mathbf{A_s}$ and $\mathbf{B_s}$ terms on lines 7 and 8 of Algorithm 5. The two $\text{ADD}^{[2\lceil w/2 \rceil + 4 + w_a]}$ terms in (5.4a) come from forming the $(\mathbf{C_s} - \mathbf{C_1} - \mathbf{C_0})$ term on line 13 of Algorithm 5, where these terms can be first summed together on $2\lceil w/2 \rceil + 4 + w_a$ bits before being shifted and added to the other product terms. The bitwidth $2\lceil w/2 \rceil + 4 + w_a$ is required because the bitwidth of $\mathbf{C_s}$ is $2\lceil w/2 \rceil + 2 + w_a$ since it is accumulations of $(2\lceil w/2 \rceil + 2)$-bit products of $(\lceil w/2 \rceil + 1)$-bit values, and the additional two bits are to account for sign extension and subtraction of the $\mathbf{C_1}$ and $\mathbf{C_0}$ terms. The two $\text{ADD}^{[2w + w_a]}$ terms in (5.4a) come from the additions to $\mathbf{C}$ on

lines 13 and 14 of Algorithm 5. The bitwidth of these additions is kept on $2w + w_a$ bits since $\mathbf{C}$ results in accumulations of $2w$-bit products of w-bit values.

(5.4a) shows that KMM significantly reduces the complexity of the 8 addition and shift operations in (5.2a) that are performed $(n/2)^{\log_2 3} d^3$ times in KSMM by reducing their occurrence by a factor of d. On the other hand, KMM trades d^3 accumulations of $2w$-bit values in (5.1b) or (5.3) for $n^{\log_2 3} d^3$ smaller-width accumulations in (5.4b). However, in Section 5.2.3 we show how the penalty of this in hardware is mitigated when combining KMM with an alternative accumulation algorithm.

Arithmetic complexity

If only counting the number of operations without considering operation bitwidths or type, we can simplify (5.1) to:

$$C(\text{MM}_n) = 2\, n^2 d^3 + 5\, (n/2)^2 d^2\,, \tag{5.5}$$

(5.3) can be simplified to:

$$C(\text{KSMM}_n) = (1 + 11\, (n/2)^{\log_2 3})\, d^3\,, \tag{5.6}$$

and (5.4) can be simplified to:

$$C(\text{KMM}_n) = (n/2)^{\log_2 3}(6\, d^3 + 8\, d^2)\,. \tag{5.7}$$

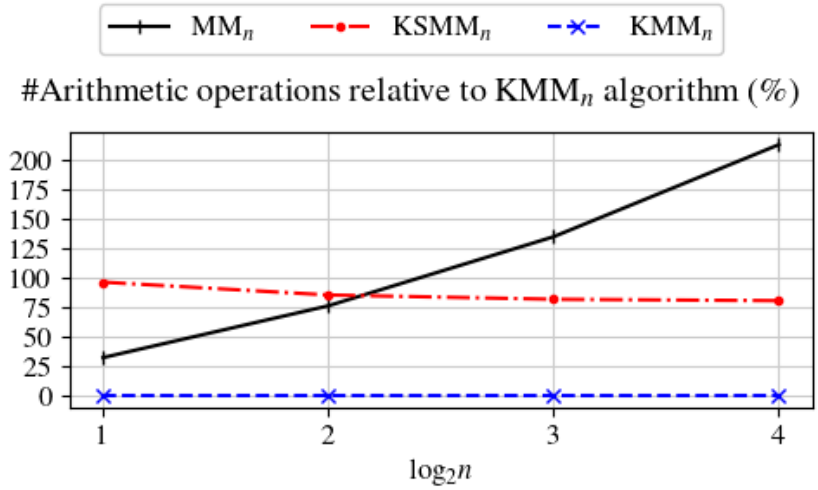

Figure 5.2: Plotting (5.5) and (5.6) relative to (5.7) for different n with $d = 64$. As can be seen, KSMM$_n$ requires over 75% more operations than KMM$_n$. Additionally, KMM$_n$ and KSMM$_n$ require exponentially fewer operations than MM$_n$ with respect to n, however, KMM$_n$ requires fewer operations than MM$_n$ even starting at $n = 2$, while KSMM$_n$ does not fall below MM$_n$ until $n > 4$.

5.2.3 Mitigating the Accumulator Complexity Increase in KMM

As found in Section 5.2.2, KMM has one penalty of trading d^3 accumulations of $2w$-bit values in (5.1b) or (5.3) for $n^{\log_2 3} d^3$ smaller-width accumulations in (5.4b). In this subsection, we show how this downside is mitigated when using Algorithm 6 as the MM$_1$ algorithm in KMM on line 16 of Algorithm 5. Algorithm 6 performs MM$_1$ using an alternative accumulation structure that reduces the accumulation hardware complexity.

In conventional matrix multiplication, each product of w-bit elements is added to a running sum kept on $2w + w_a$ bits, where $w_a = \lceil \log_2 d \rceil$ and is an extra bitwidth added to account for accumulation in order to accumulate d elements which adds extra hardware complexity. This means that normally p accumulations of $2w$-bit elements will require being added to a $(2w + w_a)$-bit running sum and each addition will be on $2w + w_a$ bits and

Algorithm 6 MM_1 algorithm with reduced accumulator complexity used in the baseline MM_1 MXUs of all compared architectures. p is defined as the number of multiplication products that are pre-accumulated on a smaller bitwidth to reduce the accumulation complexity before being added to the full-bitwidth accumulation sum. We use $p = 4$ in our evaluation.

```
1:  function MM1(A, B, p)
2:      for i = 0; i < M; i ++ do
3:          for j = 0; j < N; j ++ do
4:              C_{i,j} = 0
5:              for k = 0; k < K; k += p do
6:                  x = 0
7:                  for q = 0; q < p; q ++ do
8:                      x += A_{i,k+q} × B_{k+q,j}
9:                  end for
10:                 C_{i,j} += x
11:             end for
12:         end for
13:     end for
14:     return C
15: end function
```

therefore contain the following complexity:

$$p \text{ ACCUM}^{[2w]} = p \text{ ADD}^{[2w+w_a]} . \tag{5.8}$$

However, the average bitwidth of the addition operations is reduced when using Algorithm 6 for accumulation of p elements of bitwidth $2w$ because p elements are first added together in isolation on a smaller running sum requiring a bitwidth of only $2w + w_p$ bits for keeping p elements, where $w_p = \lceil \log_2 p \rceil$. Only after this initial pre-sum will this result then be added to the full running sum that is kept on a larger $2w + w_a$ bits for keeping d elements. This reduces the average bitwidth for every p accumulations to the following:

$$p \text{ ACCUM}^{[2w]} = \text{ADD}^{[2w+w_a]} + (p-1) \text{ ADD}^{[2w+w_p]} . \tag{5.9}$$

Furthermore, in systolic-array architectures, each accumulation output is buffered in a dedicated register, which adds further hardware complexity to the accumulation operation. However, the number of required accumulation registers when using Algorithm 6 is also reduced by a factor of p as shown in the hardware implementation from Fig. 5.3 in Sections 5.3.1 since the accumulation result only needs to be buffered after being added to the full running sum kept on $2w + w_a$ bits.

5.3 KMM Hardware Architectures

In this section, we present a general family of hardware architectures for efficiently exploiting the KMM algorithm in hardware and derive metrics for analyzing the area or execution time benefits of the KMM architectures. The first type of KMM architecture, described in Section 5.3.2, is a fixed-precision architecture optimized for executing inputs that are not expected to vary in bitwidth. We then present a precision-scalable KMM architecture in Section 5.3.3 that can more efficiently execute across multiple input precisions for applications where the input bitwidths are expected to vary.

5.3.1 Baseline MM_1 Architecture

Fig. 5.4 shows the internal structure of each baseline MM_1 MXU at the core of each KMM architecture, and Fig. 5.3 shows the internal structure of the processing elements (PE)s inside the MM_1 MXUs. Fig. 5.3 also shows the structure for how Algorithm 6 from Section 5.2.3 can be implemented in hardware and how the algorithm is able to reduce the hardware cost of the accumulator logic. This accumulation structure allows for the number of $(2w+w_a)$-bit accumulation adders and their output registers to be reduced by a factor of

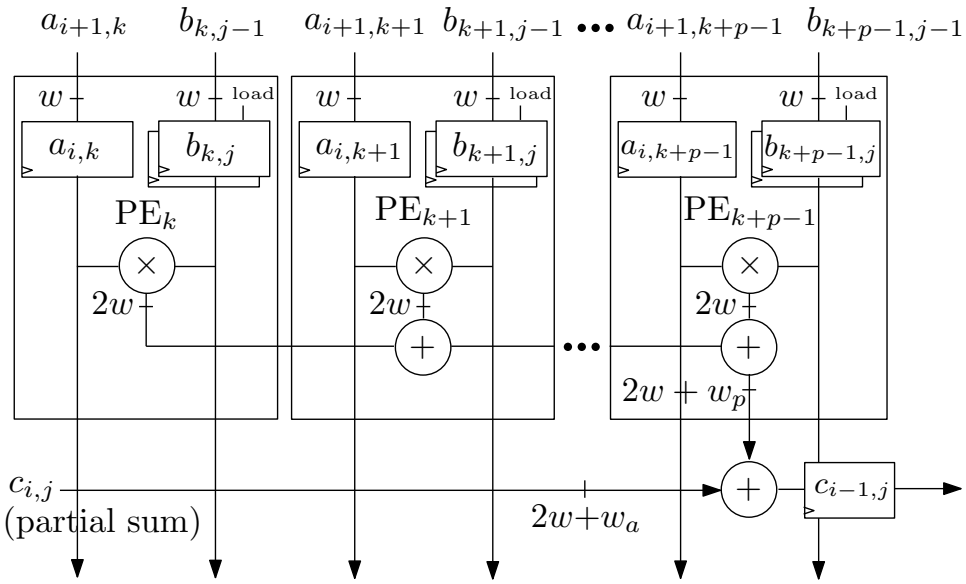

Figure 5.3: Showing the internal PE structure of the MM_1 MXUs shown in Fig. 5.4 as well as the structure for implementing Algorithm 6 in hardware to reduce the hardware cost of the accumulator logic. p is a hardware parameter equal to the number of multiplication products that are pre-accumulated on a smaller bitwidth to reduce the accumulation complexity before being added to the full-bitwidth accumulation sum. We use $p = 4$ in our evaluation.

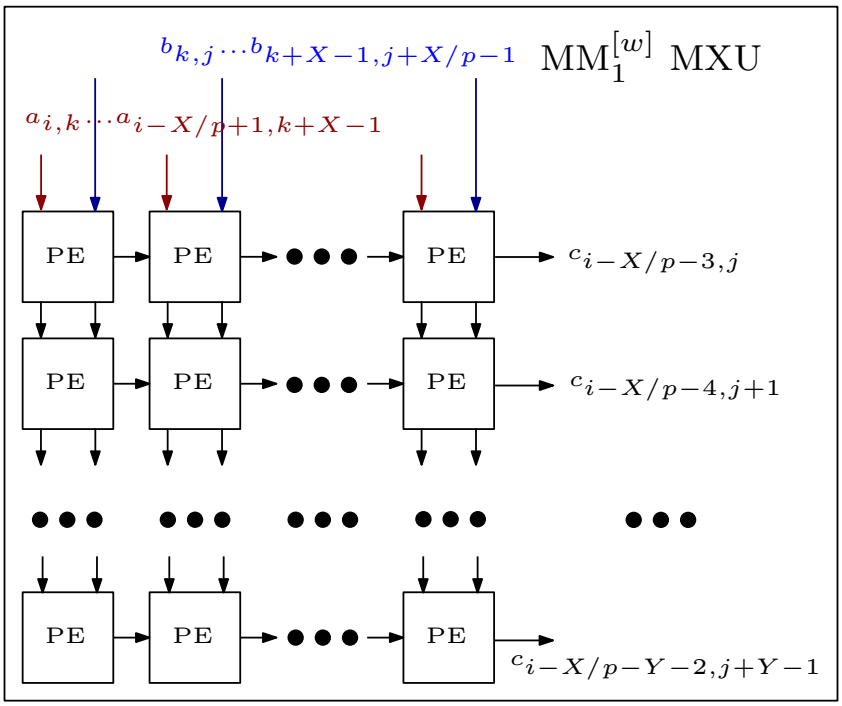

Figure 5.4: Baseline MM_1 MXU architecture present at the core of the KMM architectures, provided for context. X and Y refer to the MXU width and height in number of multipliers.

p, where they are instead traded for additions on lower-bitwidth values in the range of $2w$ to $2w + \lceil \log_2 p \rceil$ bits that do not require their output to be buffered in registers.

5.3.2 Fixed-Precision KMM Architecture

Fig. 5.5 shows the proposed fixed-precision KMM architecture for executing on inputs of a fixed precision of w bits that are not expected to vary in bitwidth. Rather than having one MXU with w-bit-input multiplier units, this architecture consists of three sub-MXUs that compute matrix multiplication on either $\lfloor w/2 \rfloor$, $\lceil w/2 \rceil + 1$, or $\lceil w/2 \rceil$-bit inputs.

The additions on lines 7 and 8 of Algorithm 5 are performed on X scalar adders at the MXU inputs. Similarly, the additions on lines 13 and 14 of Algorithm 5 are performed on Y scalar adders at the MXU outputs. Due to the nature of right/left shifting by a constant offset in custom hardware, the shift operations at the output of the MXUs do not require any area overhead. If desired, each of the three sub-MXUs can also be instantiated as another KMM MXU containing three more sub-MXUs to implement additional levels of KMM recursion. The final level of MXUs will be MM_1 MXUs.

5.3.3 Precision-Scalable KMM Architecture

Fig. 5.7 shows the proposed precision-scalable KMM architecture for implementing one level of KMM recursion. This architecture can more efficiently use m-bit-input multipliers to execute across varying input precisions of bitwidth w for applications where the input bitwidths are expected to vary. Unlike in prior works [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], the minimum possible execution time when fully utilizing the compute resources scales less than quadratically with the input bitwidths. As discussed further in Section 5.3.4, the input matrices are divided into tiles and fed into the

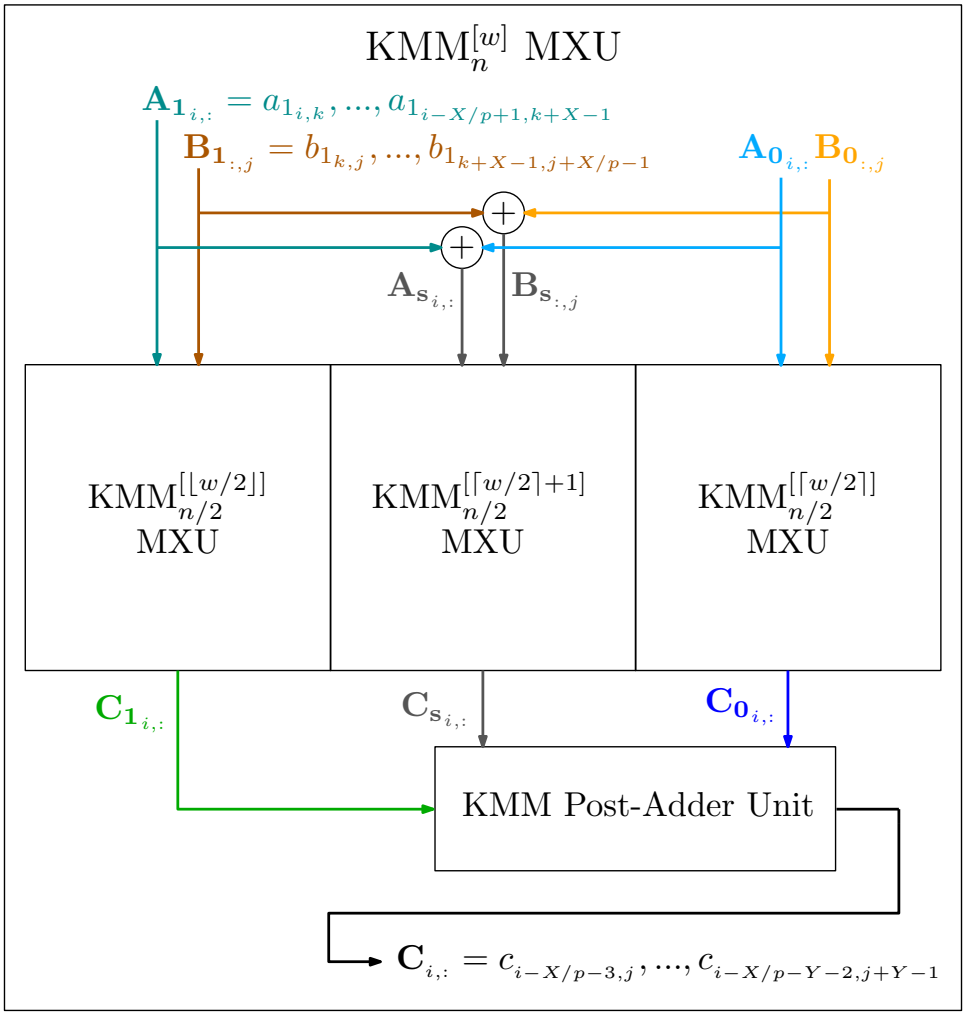

Figure 5.5: Fixed-precision KMM architecture for executing on inputs of a fixed precision of w bits.

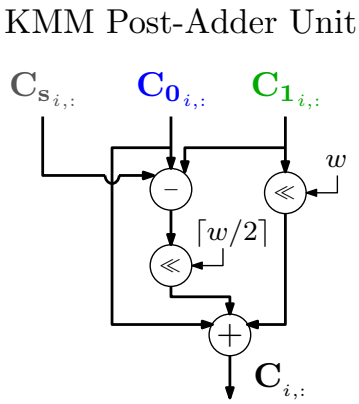

Figure 5.6: KMM Post-Adder Unit from Fig. 5.5 for executing $\mathbf{C_1}_{i,:} \ll w + \left(\mathbf{C_s}_{i,:} - \mathbf{C_1}_{i,:} - \mathbf{C_0}_{i,:}\right) \ll \lceil w/2 \rceil + \mathbf{C_0}_{i,:}$.

MXU one-by-one to perform GEMM. In this architecture, each set of input matrix tiles may be read multiple times and either the MM_1, MM_2, or KMM_2 algorithm may be executed depending on the input bitwidths w and the multiplier bitwidth m. An iteration state signal t is reset when a new set of input tiles is read and is incremented each time the same set of input tiles is re-read.

MM_1 and MM_2 Mode

If $w \leq m$, the architecture will execute the MM_1 algorithm, bypassing any MXU input/output addition or shifting steps, $\mathbf{A_0}$ and $\mathbf{B_0}$ will be fed into the MXU as inputs, and each set of input tiles is read only once.

If $2m - 2 < w \leq 2m$, the architecture will execute the MM_2 algorithm and each set of input matrix tiles will be read a total of four times before proceeding to the next set of input tiles. The MM_2 algorithm is used instead of KMM_2 for this input bitwidth range because the bitwidth of the elements in the $\mathbf{A_s}$ and $\mathbf{B_s}$ matrices in Algorithm 5 would be $m + 1$ which would be too large by 1 bit to fit onto the m-bit multipliers in the MXU. In each read for this input bitwidth range, the MXU will accept either the $\mathbf{A_1}$ and $\mathbf{B_1}$ inputs or the $\mathbf{A_0}$

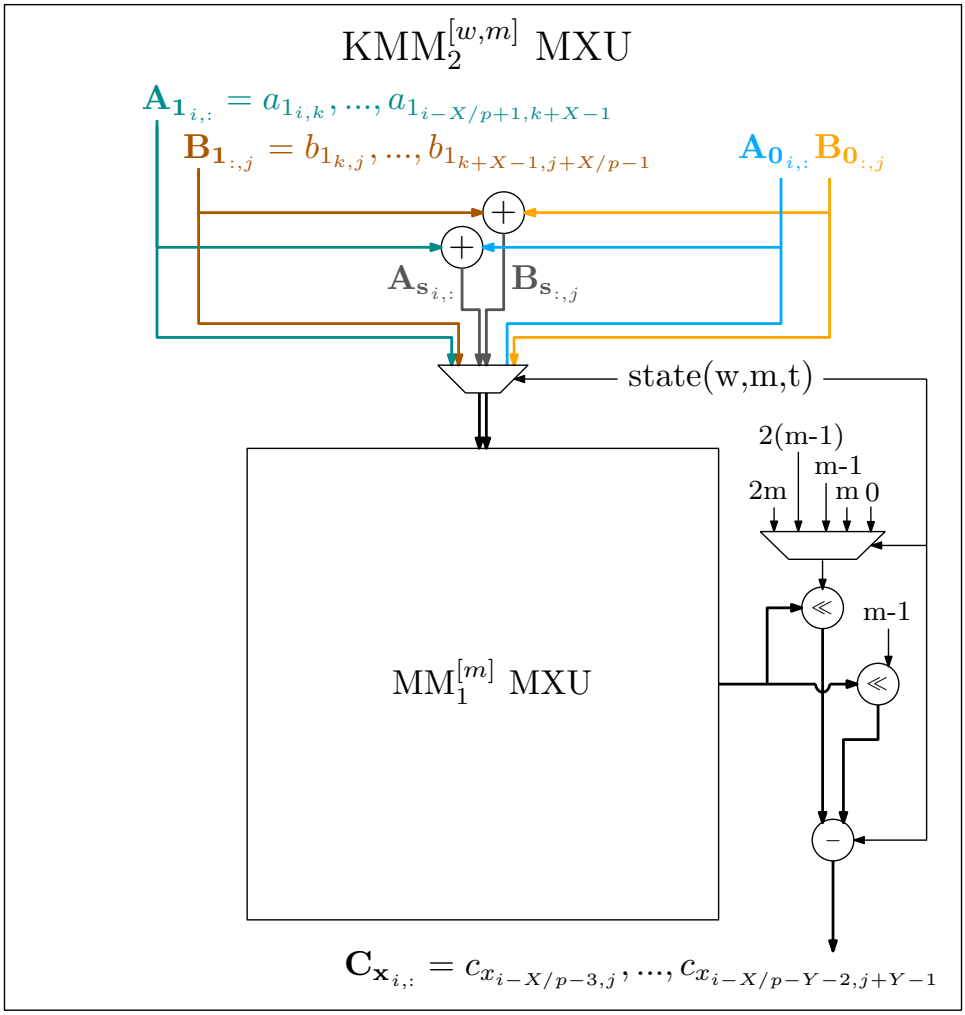

Figure 5.7: Precision-scalable KMM architecture for more efficiently using m-bit-input multipliers to execute across varying input precisions of bitwidth w for applications where the input bitwidths are expected to vary.

and $\mathbf{B_0}$ inputs depending on the tile read iteration t. $\mathbf{A_1}$ and $\mathbf{B_1}$ will contain bits $2m-1$ down to m of the $\mathbf{A}$ and $\mathbf{B}$ matrix elements. $\mathbf{A_0}$ and $\mathbf{B_0}$ will contain bits $m-1$ down to 0 of the $\mathbf{A}$ and $\mathbf{B}$ matrix elements.

The MXU output vectors $\mathbf{C}_{\mathbf{x}_{i,:}}$ in Fig. 5.7 will be equal to either $(\mathbf{C}_{\mathbf{1}_{i,:}} \ll 2m)$, $(\mathbf{C}_{\mathbf{10}_{i,:}} \ll m)$, $(\mathbf{C}_{\mathbf{01}_{i,:}} \ll m)$, or $\mathbf{C}_{\mathbf{0}_{i,:}}$ depending on the tile read iteration t to incrementally execute lines 11-13 of Algorithm 3 throughout the tile read iterations, where m is considered equivalent to the value of $\lceil w/2 \rceil$ in Algorithm 3. Specifically, depending on the tile read iteration t, the MXU output vectors will be equal to $(\mathbf{C}_{\mathbf{1}_{i,:}} \ll 2m)$ to form the addition on line 11 of Algorithm 3, $\mathbf{C}_{\mathbf{0}_{i,:}}$ to form the addition on line 13, and separately $(\mathbf{C}_{\mathbf{10}_{i,:}} \ll m)$ or $(\mathbf{C}_{\mathbf{01}_{i,:}} \ll m)$ to collectively form the addition on line 12.

Each partial matrix tile product will need to be accumulated with prior ones outside of the MXU, however, this is the same functionality already present in GEMM where multiple matrix tile products must be summed to form a final matrix product, and this functionality will therefore already be present in GEMM accelerators outside of the MXU such as in the GEMM and deep learning accelerator system from discussed in Chapter 3.

KMM$_2$ Mode

If $m < w \leq 2m-2$, the architecture will execute the KMM$_2$ algorithm and each set of input matrix tiles will be read a total of three times before proceeding to the next set of input tiles. For each read, the MXU will accept or form either the $\mathbf{A_1}$ and $\mathbf{B_1}$ inputs, the $\mathbf{A_s}$ and $\mathbf{B_s}$ inputs, or the $\mathbf{A_0}$ and $\mathbf{B_0}$ inputs depending on the tile read iteration t. $\mathbf{A_1}$ and $\mathbf{B_1}$ will contain bits $2(m-1)-1$ down to $m-1$ of the $\mathbf{A}$ and $\mathbf{B}$ matrix elements. $\mathbf{A_0}$ and $\mathbf{B_0}$ will contain bits $m-2$ down to 0 of the $\mathbf{A}$ and $\mathbf{B}$ matrix elements. The MXU output vectors $\mathbf{C}_{\mathbf{x}_{i,:}}$ in Fig. 5.7 will be equal to either $\big[(\mathbf{C}_{\mathbf{1}_{i,:}} \ll 2(m-1)) - (\mathbf{C}_{\mathbf{1}_{i,:}} \ll (m-1))\big]$,

$\left[\mathbf{C}_{\mathbf{s}_{i,:}} \ll (m-1)\right]$, or $\left[\mathbf{C}_{\mathbf{0}_{i,:}} - (\mathbf{C}_{\mathbf{0}_{i,:}} \ll (m-1))\right]$ depending on the tile read iteration t to incrementally execute lines 12-14 of Algorithm 5 throughout the tile read iterations, where $m-1$ is considered equivalent to the value of $\lceil w/2 \rceil$ in Algorithm 5.

Each partial matrix tile product will need to be accumulated with prior ones outside of the MXU, however, this functionality will already be present in GEMM accelerators as explained above in Section 5.3.3.

A precision-scalable MM_2 architecture can also be implemented that has a similar structure as the precision-scalable KMM architecture, except that it will only either execute the MM_1 algorithm if $w \leq m$ or the MM_2 algorithm if $m < w \leq 2m$. We also note that a precision-scalable KSMM architecture exploiting KSM_2 would not be as efficient to implement in hardware compared to a precision-scalable KMM architecture. This is because, in addition to the extra adders that would be required at the output/inputs of every multiplier as discussed in Section 5.2.2, multiplexers would also have to be placed at the output/inputs of every multiplier in the MXU as well for output/input arbitration depending on the width of the inputs. In contrast, the KMM architecture reduces this extra adder complexity as already discussed, and it can employ an efficient more conventional systolic array at the core not requiring multiplexers surrounding each multiplier.

5.3.4 System Integration

To house and evaluate the KMM MXU architectures, we integrate the them into a deep learning accelerator system design based on the one described in Chapter 3. We were able to swap the KMM and baseline precision-scalable MM MXU architectures into the accelerator system in place of the baseline MXU. This change was mostly seamless but also required updates to the memory system such that each set of input matrix tiles can

optionally be re-read up to three or four times before proceeding to the next set of input tiles. The number of times that the matrix tiles are re-read and the purpose for this is explained in Section 5.3.3.

The presented KMM architectures are illustrated for unsigned integer inputs, however, if the inputs are signed, a 1-dimensional adder vector can be used to add a constant offset to the inputs of an MXU to convert them to unsigned. The zero-point adjuster method from Chapter 4 can then be used to efficiently eliminate the effects of this constant offset in the matrix products before exiting the MXU.

5.3.5 Multiplier Compute Efficiency

In this subsection, we extend the the multiplier compute efficiency performance-per-area metric that was defined in (4.18c) so that it is better suited for comparing the KMM architecture against baseline models and prior works. The extended metric is used to compare the amount of computational work that can be performed per compute area per clock cycle regardless of the input bitwidths. The importance of this property is expanded upon more later in this subsection, as well as in Section 5.4.1.

The throughput metric in (4.18c) measures the number of w-bit multiplications being performed, where w is the algorithm input bitwidths. However, in order to execute KMM in hardware, the algorithm input bitwidths w must be larger than the multiplier bitwidths, and the number of larger w-bit multiplications that can be performed per multiplier will be lower than the actual effective number of multiplications being performed per multiplier. Therefore, the maximum achievable value for the metric from (4.18c) will vary depending on the input bitwidths w and is not ideal for reflecting the true amount of computational work being performed per multiplier regardless of the input widths.

To address this, we can instead measure (4.18c) directly in terms of effective m-bit multiplications being performed per multiplier, where m may be smaller than the algorithm input bitwidths w. This derives the following metric for measuring the true amount of effective multiplications being performed per multiplier regardless of the algorithm input bitwidths w:

$$\frac{m\text{-bit mults/multiplier}}{\text{clock cycle}} = \frac{(m\text{-bit mults/s})/\#\text{multipliers}}{f}\,, \tag{5.10}$$

where m-bit mults/s above is measured by taking the number of m-bit multiplications required to carry out an execution on w-bit inputs using conventional algebra and dividing it by the measured execution time, #multipliers is the number of instantiated multipliers in the design, and f is the clock frequency that the hardware design is operating at. Conventional algorithms used in prior work to perform larger w-bit multiplications on smaller m-bit multipliers are the SM or MM algorithms (Algorithm 1 and 3). The number of m-bit multiplications required to carry out a larger w-bit multiplication using conventional algebra (i.e. SM or MM) is equal to the number of w-bit multiplication in the execution times 4^r, where r is equal to:

$$r = \lceil \log_2 n \rceil = \lceil \log_2 \lceil w/m \rceil \rceil\,. \tag{5.11}$$

The limit (also referred to as the roof) of the metric in (5.10) when executing the conventional MM algorithm in hardware is then the following since it has no algebraic optimizations for reducing the computational complexity:

$$\text{MM}_n^{[w]} \;\; \frac{m\text{-bit mults/multiplier}}{\text{clock cycle}} \;\; \text{roof} = 1\,. \tag{5.12}$$

In contrast, the KMM algorithm requires only 3^r smaller-bitwidth multiplications to form every w-bit product rather than 4^r as in MM. Therefore, the multiplier compute efficiency can reach the following limit in KMM architectures:

$$\text{KMM}_n^{[w]} \ \frac{m\text{-bit mults/multiplier}}{\text{clock cycle}} \ \text{roof} = \left(\frac{4}{3}\right)^r . \tag{5.13}$$

5.3.6 Area Unit (AU) Compute Efficiency

In this subsection, we define a performance-per-area metric in (5.21) that accounts for the area overhead of registers, adder units, and multipliers all in a single unit of comparison based around the area of a full adder. Using this abstracted method for modelling the circuit area allows for a general complexity analysis that is less biased towards one specific implementation platform or technology.

We first derive the relative area of adders and registers by modeling that the area of a w-bit adder will be approximately equal to the area of w full adders. We then approximate the area of a w-bit flip-flop/register relative to a w-bit adder according to approximate transistor counts of full adders versus D-flip-flops based on several sources. While there are different specific implementations for these components, we use the approximate transistor count trends for the implementations in prior work [105], [106], [107], where a standard CMOS full adder uses 28 transistors [105] and a 1-bit flip-flop consumes 18-21 transistors [106], [107] (which we then approximate as 19.5), to arrive at the general area estimation shown in (5.14a) and (5.14b) of 1 flip-flop equalling the area of approximately $19.5/28 = 0.7$ full adders. So long as these area ratios vary within reasonable bounds as found in prior work [105], [106], [107], the conclusions from our results do not change.

We then model the approximate area of a w-bit multiplier circuit based on the area

of a w-bit adder. While there are different possible multiplier circuit implementations, the area of multiplier circuits used in practice commonly scale quadratically with the area of a full adder [71], [25], [72], [108]. Furthermore, the KMM architectures are not tied to being implemented using one specific multiplier circuit type. Therefore, in order to provide a more general analysis and insight catering to a broader range of possible KMM implementations, we approximate the area of a multiplier based on the general trend of equalling the square of the input bitwidths times the area of a full adder as shown in (5.14c). We then arrive at the following general area approximations:

$$\text{Area}(\text{ADD}^{[w]}) = w \text{ AU} \tag{5.14a}$$

$$\text{Area}(\text{FF}^{[w]}) = 0.7\ w \text{ AU} \tag{5.14b}$$

$$\text{Area}(\text{MULT}^{[w]}) = w^2 \text{ AU}\,. \tag{5.14c}$$

Based on this, we can then derive the AU of each architecture by substituting in the areas from (5.14) for each of the corresponding hardware components in the architectures. The area of a baseline MM_1 MXU is then as follows:

$$\begin{aligned}\text{Area}(\text{MM}_1^{[w]}) = XY\ \text{Area}(\text{MULT}^{[w]} + 3\ \text{FF}^{[w]} \\ + \text{ACCUM}^{[2w]})\,.\end{aligned} \tag{5.15}$$

Here, the area of an accumulator is based on Algorithm 6 and its implementation in Fig. 5.3, where the number of accumulator registers and $(2w+w_a)$-bit accumulation adders in the MXU are reduced by a factor of p. Based on this, by substituting in the areas in (5.14) for the adders and registers forming the accumulators in Fig. 5.3, every p accumulators on

average then contain the following area:

$$p\ \text{Area}(\text{ACCUM}^{[2w]}) = (p-1)\ \text{Area}(\text{ADD}^{[2w+w_p]}) + \text{Area}(\text{ADD}^{[2w+w_a]} + \text{FF}^{[2w+w_a]})\,. \tag{5.16}$$

In (5.15) - (5.16), X and Y are the MXU width and height in number of multipliers, $w_p = \lceil \log_2 p \rceil$, and w_a is the following additional bitwidth added to account for accumulation:

$$w_a = \lceil \log_2 X \rceil\,. \tag{5.17}$$

As discussed in Section 5.3.4, the register requirements in (5.15) are derived from the fact that each PE in the MM_1 MXU will contain registers for buffering the a and b inputs being multiplied, as well as one additional b buffer for loading the next b tile into the MXU as the current tile is being multiplied.

The area of the KSMM architecture, which is a baseline MM_1 MXU using KSM multipliers rather then conventional multipliers, is then:

$$\text{Area}(\text{KSMM}_n^{[w]}) = XY\ \text{Area}(\text{KSM}_n^{[w]} + 3\ \text{FF}^{[w]} + \text{ACCUM}^{[2w]})\,, \tag{5.18}$$

where:

$$\begin{aligned}\text{Area}(\text{KSM}_n^{[w]}) = \ & \text{Area}(\text{ADD}^{[2w]}) \\ & +2\,\text{Area}(\text{ADD}^{[2\lceil w/2\rceil+4]} + \text{ADD}^{[\lceil w/2\rceil]}) \\ & +\text{Area}(\text{KSM}_{n/2}^{[\lfloor w/2\rfloor]} + \text{KSM}_{n/2}^{[\lceil w/2\rceil+1]}) \\ & +\text{Area}(\text{KSM}_{n/2}^{[\lceil w/2\rceil]}) \end{aligned} \tag{5.19a}$$

$$\text{Area}(\text{KSM}_1^{[w]}) = \text{Area}(\text{MULT}^{[w]})\,. \tag{5.19b}$$

The addition of c_0 on line 14 of Algorithm 2 is not included in this area estimate because it can be performed before line 13 where c_0 will be on w bits and will not overlap with $c_1 \ll w$. Therefore, this addition can be performed at no cost in hardware by simply concatenating the two terms together.

The area of the KMM architecture is then:

$$\begin{aligned}\text{Area}(\text{KMM}_n^{[w]}) = \ & 2X\,\text{Area}(\text{ADD}^{[\lceil w/2\rceil]}) \\ & +2Y\,\text{Area}(\text{ADD}^{[2\lceil w/2\rceil+4+w_a]} + \text{ADD}^{[2w+w_a]}) \\ & +\text{Area}(\text{KMM}_{n/2}^{[\lfloor w/2\rfloor]} + \text{KMM}_{n/2}^{[\lceil w/2\rceil+1]}) \\ & +\text{Area}(\text{KMM}_{n/2}^{[\lceil w/2\rceil]}) \end{aligned} \tag{5.20a}$$

$$\text{Area}(\text{KMM}_1^{[w]}) = \text{Area}(\text{MM}_1^{[w]})\,. \tag{5.20b}$$

Due to the nature of right/left shifting by a constant offset in custom hardware, the shift operations in the KSMM and KMM algorithms do not add additional area in the corresponding architectures.

We can now compare the AU compute efficiency limits of the MM_1, KSMM, and KMM

architectures using:

$$\frac{\text{throughput}/\text{Area Unit}}{\text{clock cycle}} \text{ roof} = \frac{\text{throughput roof}/\text{Area(ARCH)}}{f}, \tag{5.21}$$

where ARCH represents one of the mentioned architectures. Throughput roofs are equal for fixed-precision MM_1, KSMM, and KMM architectures with equal X/Y MXU dimensions. Therefore, the value of (5.21) for each architecture relative to the MM_1 architecture can be found through the inverse of its AU from (5.15), (5.18), or (5.20) relative to the inverse of the MM_1 AU in (5.15) as plotted later in Fig. 5.9.

5.4 Results

5.4.1 Evaluation Metrics

In Section 5.4, we compare the KMM architectures against other designs using the multiplier and Area Unit compute efficiency metrics defined in (5.10) and (5.21) from Sections 5.3.5 and 5.3.6, respectively. These are both used to compare an architecture's throughput per area capabilities regardless of the clock frequency.

Additionally, the multiplier compute efficiency also measures the amount of computational work being performed per compute area regardless of the clock frequency *or input bitwidths*. This is an important quality because prior works using the same compute platform as us for evaluation only evaluate throughput for input bitwidths w that are equal to the multiplier bitwidths m. However, in order to execute KMM in hardware, the input bitwidths w must be larger than the multiplier bitwidths. Therefore, to fairly compare the

performance of the prior works against our KMM architecture, we need to use a performance metric with a maximum achievable value that does not change regardless of the input bitwidths w being executed, which is not the case for the GOPS metric. Furthermore, the multiplier compute efficiency is also useful for comparison with prior works because it is measurable using only throughput, number of multipliers, and frequency, which are commonly provided or derivable in prior works.

The Area Unit compute efficiency metric also accounts for the area overhead of registers and adder units and provides a more general abstracted method for modelling the circuit area that is less biased towards one specific implementation platform or technology. However, it is only useful for comparing architectures which compute on inputs of the same bitwidth, and it is only derivable when knowing not only the number of multipliers used in an architecture, but also the number of adders and registers which is information that is not readily available from prior works, but we can use it to model the efficiencies of the fixed-precision KMM architecture against our baseline designs which we know all of these details about.

5.4.2 Comparison to Prior Work

Although the theoretical concepts presented in this work are general and applicable to both custom integrated circuits and FPGA implementations, our example KMM implementations were validated on FPGA, and we therefore compare against state-of-the-art prior works that are also evaluated on FPGA.

As discussed in Chapter 5.3.4, we use the deep learning accelerator system design described in Chapter 3 to house and evaluate our example KMM and baseline MXU architectures. Full system-level validation of the experimental accelerator as integrated into the

Table 5.1: Proposed precision-scalable KMM and baseline MM systolic-array architectures integrated into a deep learning accelerator system compared with each other and prior state-of-the-art deep learning accelerators on Arria 10 GX 1150 FPGA.

	TNNLS '22 [95]		TCAD '22 [96]		Entropy '22 [97]		$MM_2^{[w,8]}$ 64×64			$KMM_2^{[w,8]}$ 64×64		
DSPs	1473		1473		1503		1518			1518		
ALMs	304K		304K		303K		149K			153K		
Registers	889K		890K		-		391K			390K		
Memories	2334		2334		1953		2446			2446		
Frequency (MHz)	200		220		172		334			335		
Model	ResNet-50	VGG16	Bayes ResNet-18	Bayes VGG11	R-CNN (ResNet-50)	R-CNN (VGG16)	ResNet-50	ResNet-101	ResNet-152	ResNet-50	ResNet-101	ResNet-152
Input bitwidth (w)	8	8	8	8	8	8	1-8 / 9-16	1-8 / 9-16	1-8 / 9-16	1-8 / 9-14 / 15-16	1-8 / 9-14 / 15-16	1-8 / 9-14 / 15-16
Throughput (GOPS)	1519	1295	1590	534	719	865	2200 / 550	2405 / 601	2495 / 624	2200 / 735 / 552	2412 / 804 / 603	2502 / 834 / 626
$\frac{\text{8-bit mults}^{1}}{\text{multiplier}}$ clock cycle	0.645	0.550	0.639	0.206	0.696	0.837	0.792 / 0.792	0.865 / 0.865	0.898 / 0.898	0.792 / **1.055** / 0.792	0.865 / **1.154** / 0.865	0.898 / **1.197** / 0.898

[1] Multiplier compute efficiency, used to compare the amount of computational work being performed per compute area regardless of the input bitwidths or clock frequency, defined in (5.10) from Section 5.3.5, relevance explained in Section 5.3.5 and 5.4.1.

system from Chapter 3 has been done on an Arria 10 SoC Developement Kit [92] containing the Arria 10 SX 660 device by measuring throughput in real-time. However, this device contains fewer soft logic resources than the Arria 10 GX 1150 used in the prior works we compare against, and we generate compilation results for our design on the same Arria 10 GX 1150 device used in prior works for a more fair and consistent comparison. Throughput values of our designs on the Arria 10 GX 1150 device are then calculated using an accurate throughput estimation model based on our highly deterministic and time-predictable system implementation, which accurately predicts actual throughputs measured on the Arria 10 SX 660 device available to us. Tables 5.1-5.3 show throughputs for ResNet [17] neural network models.

Table 5.2: Comparison of an FFIP [8] systolic-array architecture, which doubles performance per MAC unit, with combined FFIP+KMM systolic-array architectures when integrated into deep learning accelerator systems on Arria 10 GX 1150 FPGA.

	TC '24 [8] (FFIP 64×64)			FFIP+KMM$_2^{[w,8]}$ 64×64		
DSPs	1072			1072		
ALMs	118K			133K		
Registers	311K			334K		
Memories	1782			2445		
Frequency (MHz)	388			353		
Model	ResNet-50	ResNet-101	ResNet-152	ResNet-50	ResNet-101	ResNet-152
Input bitwidth (w)	8	8	8	1-8 / 9-14 / 15-16	1-8 / 9-14 / 15-16	1-8 / 9-14 / 15-16
Throughput (GOPS)	2529	2752	2838	2325 / 775 / 581	2542 / 847 / 635	2637 / 879 / 659
8-bit mults [1] / multiplier / clock cycle	1.521	1.655	1.707	1.536 / **2.048** / 1.536	1.679 / **2.239** / 1.679	1.742 / **2.322** / 1.742

[1] Multiplier compute efficiency, used to compare the amount of computational work being performed per compute area regardless of the input bitwidths or clock frequency, defined in (5.10) from Section 5.3.5, relevance explained in Section 5.3.5 and 5.4.1.

In Table 5.1, the number of multipliers in the work from An et al. [97] is calculated as $\#DSPs \times 2$, where each DSP in the Intel/Altera FPGAs instantiates two 18×19-bit multipliers [94]. This is also how the number of multipliers is calculated in the architectures in Table 5.2. The works from Liu et al. [95] and Fan et al. [96] in Table 5.1 pack two 8-bit multiplications onto each 18×19-bit multiplier in the DSPs and additional ALMs, and therefore $\#multipliers = \#DSP \times 4$ in those works. Our architectures in Tables 5.1 and 5.1 contain 64×64 + 64 multipliers, where 3036 are instantiated on DSPs, and the remainder are instantiated using soft logic resources due to the DSP resources being fully utilized. 64×64 multipliers are used in the MXU of our designs from Tables 5.1 and 5.1, while the remaining 64 multipliers in the each design are located outside the MXU in the Post-GEMM Unit discussed in Section 3.4 for performing inter-layer quantization rescaling functions.

Table 5.1 compares the KMM architecture with state-of-the-art accelerators evaluated on the same FPGA family for the same instantiated multiplier bitwidths and similar neural network models. The proposed KMM architecture is very efficient, achieving the highest throughput and compute efficiency compared to the prior works in Table 5.1. The KMM design here achieves compute efficiencies approaching the KMM_2 limit of 1.33 when executing on bitwidths in the range of 9-14 bits that is derived in (5.13) and surpasses the limit of 1 in prior works that is derived in (5.12).

It is also noted that the proposed systolic arrays in Tables 5.1 and 5.2 that are integrated into a full accelerator system include a number of other components such as memory subsystems and control as described in Chapter 3, and these other system components form the frequency-limiting critical path as opposed to the proposed systolic-array architectures.

Table 5.2 shows an example of how KMM can be combined with other algebraic techniques to further increase compute efficiency limits. FFIP [8] (Chapter 4) provides a way to reduce the number of required multiplications by a factor of 2, by trading half the multiplications for cheap low-bitwidth additions. Because the number of required multiplications is reduced by 2, the limit for the multiplier compute efficiency metric in (5.13) becomes 2 for FFIP, and $(8/3)^r$ for FFIP+KMM. In Table 5.2, we combine KMM with FFIP [8] by using an FFIP MXU as the base MXU in the KMM architecture instead of a conventional MM_1 MXU to further increase the compute efficiency compared to standalone FFIP. The FFIP+KMM architectures in Table 5.2 have additional memory resources instantiated compared to the FFIP-only design in order to support inference on up to 16-bit inputs, and this also adds a penalty in the soft logic resources and clock frequency. However, the multiplier compute efficiency of the FFIP+KMM designs surpass the FFIP limit of 2, and approach the FFIP+KMM_2 limit of 2.67.

5.4.3 Comparison to Baseline Designs

Precision-Scalable Architectures

Table 5.1 includes the resource usage and performance comparison between the proposed KMM and the baseline MM architectures. The multiplier compute efficiency of KMM surpasses that of the baseline MM architecture when executing on bitwidths in the range of 9-14 bits, achieving compute efficiencies approaching the KMM_2 limit of 1.33 that is derived in (5.13) and surpassing the limit of 1 of the baseline MM architecture and prior works that is derived in (5.12), validating KMM's ability to increase compute efficiency as expected from our analysis. This is also reflected in the GOPS from Table 5.1, where the KMM architecture achieves a 1.33$\times$ speedup over MM for input bitwidths in the range of

Table 5.3: Comparison of proposed fixed-precision KMM and baseline MM_1 and KSMM systolic-array architectures in isolation (without integration into a deep learning accelerator system) on Arria 10 GX 1150 FPGA. All designs in this table contain 0 memory resources.

	$MM_1^{[32]}$ 24×24	$MM_1^{[32]}$ 24×24	$KSMM_2^{[32]}$ 24×24	$KSMM_2^{[32]}$ 24×24	$KMM_2^{[32]}$ 24×24	$MM_1^{[64]}$ 12×12	$MM_1^{[64]}$ 12×12	$KSMM_4^{[64]}$ 12×12	$KSMM_4^{[64]}$ 12×12	$KMM_4^{[64]}$ 12×12
Input bitwidth	32	32	32	32	32	64	64	64	64	64
DSPs	1152	1152	864	864	864	1224	1224	648	648	648
ALMs	30K	46K	60K	88K	31K	33K	31K	66K	84K	30K
Registers	104K	182K	185K	310K	121K	40K	119K	68K	309K	121K
Frequency (MHz)	232	374	279	373	395	120	311	102	354	402
Throughput roof (GOPS)	267	431	321	430	455	35	90	29	102	116

9-14 bits.

For illustration, Fig. 5.8 plots the limits of the multiplier compute efficiency metric defined in (5.10) from Section 5.3.5 for the precision-scalable KMM_2 architecture compared to the conventional precision-scalable MM_2 architecture for $X = Y = 64$. As shown, the KMM architecture surpasses the MM architecture's limit of 1 for this metric, extending the limit to 1.33 for bitwidths 9-14 since the KMM_2 algorithm requires only 3 m-bit multiplications for every w-bit product rather than 4 as in the MM_2 algorithm.

Fixed-Precision Architectures

Table 5.3 shows synthesis results for baseline MM_1, KSMM, and proposed KMM systolic-array architectures in isolation (not integrated into a deep learning accelerator) for different input bitwidths and levels of KSM and KMM recursion. The input bitwidths are intentionally larger than the DSP units' native multiplier bitwidths and are chosen to allow for larger multiplications to be broken down into smaller multiplications of bitwidths at or just below the native widths supported by the DSPs, which house 18-bit multipliers. It is expected that the larger-bit multiplications in the MM_1 designs will be mapped to smaller 16-bit

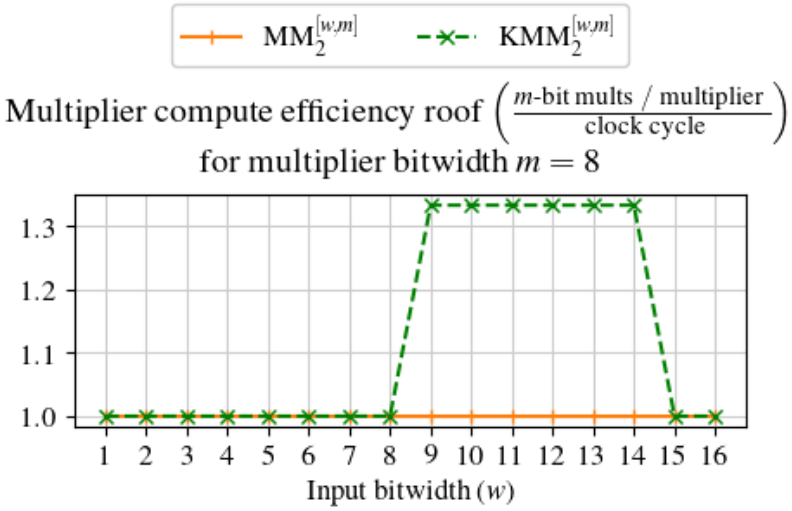

Figure 5.8: Maximum achievable multiplier compute efficiencies (derived in Section 5.3.5) for the precision-scalable MM_2 and KMM_2 architectures.

multipliers, and onto fewer 16 to 18-bit multipliers in the KMM and KSMM designs.

The reduction in multiplication complexity of KMM and KSMM achieved through breaking down larger multiplications into smaller-bitwidth multiplications can be seen relative to conventional approaches (evaluated through the MM_1 architectures) by comparing the reduction in number of DSP units for the KMM and KSMM designs relative to MM_1. Furthermore, the reduction in addition complexity of KMM relative to KSMM can be seen in the reduction in ALMs in the KMM architectures compared to the KSMM architectures.

The MM_1 and KSMM architectures innately have a lower clock frequency than KMM because it is expected that each multiplication being performed in the PEs require n^2 or $n^{\log_2 3}$ DSP units, respectively, whereas the KMM designs require only 1 DSP unit in each individual KMM systolic-array PE. This leads to a less localized design. In contrast, the KMM design uses multiple independent systolic arrays requiring 1 DSP unit per multiplication to perform a single 16 to 18-bit multiplication, and the DSPs in each systolic array do not require interconnections with the DSPs in other systolic arrays, leading to a more

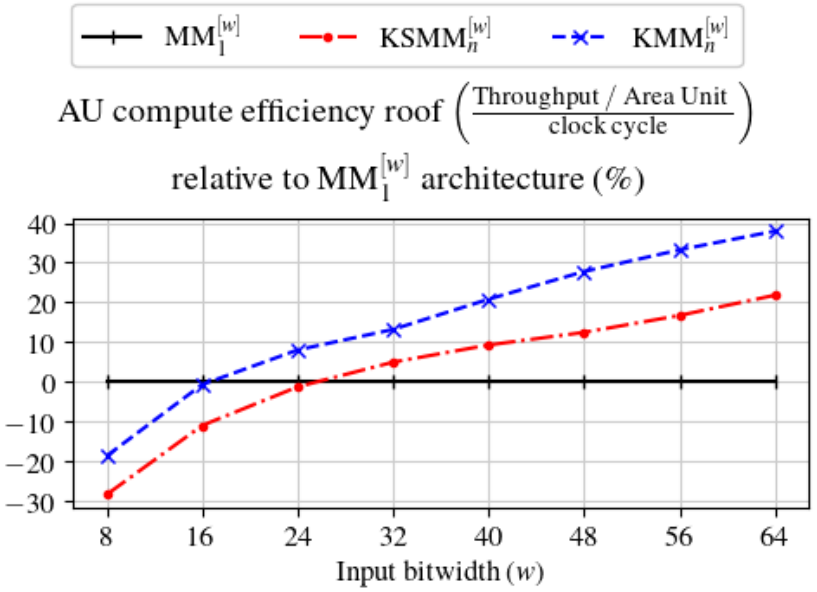

Figure 5.9: Maximum achievable AU compute efficiencies (derived in Section 5.3.6) for the fixed-precision MM_1, $KSMM_n$, and KMM_n architectures.

localized design. Due to this, we provide results of two design variants for each of the MM_1 and KSMM architectures, where one variant contains additional pipelining registers added into the PE datapaths such that the clock frequency can reach closer to that of the KMM designs. However, it can be seen that the MM_1 and KSMM designs are still unable to match the frequency of KMM even with extra pipelining registers, especially for the 64-bit input designs.

In summary, the trend in Table 5.3 is that the KMM designs may contain more register resources than the MM_1 and KSMM designs depending on the amount of pipelining registers used, however, the KMM designs use significantly fewer ALM resource than the KSMM designs, significantly fewer DSP units than the MM_1 designs, and achieve significantly higher clock frequencies than both KSMM and MM_1.

Fig. 5.9 also provides a more general modelling of the performance-per-area of the

KMM architectures that is less biased towards one specific implementation platform or technology by plotting the AU compute efficiency limits derived in Section 5.3.6 that can be achieved for the fixed-precision MM_1, KSMM, and KMM architectures for different supported fixed-precision input widths and instantiated multiplier bitwidths for $X = Y = 64$. The KMM and KSMM architectures for each bitwidth implement as many levels of Karatsuba recursion as possible while still reducing the area, with a minimum of least one level of Karatsuba recursion being implemented (even if the one level has a larger area than using conventional MM_1). This results in one recursion level being implemented in the KSMM architectures for every bitwidth. For the KMM architectures, this results in one recursion level for bitwidths 8-32, two recursion levels for bitwidths 40-56, and three recursion levels for bitwidth 64.

As can be seen, the KMM architecture achieves a higher throughput per Area Unit than the conventional MM_1 architecture starting sooner at a lower bitwidth compared to the KSMM architecture, and it is consistently higher than the KSMM architecture across all input/multiplier bitwidths.

5.5 Summary

In this chapter, we propose the extension of the scalar Karatsuba multiplication algorithm to matrix multiplication, showing how this maintains the reduction in multiplication complexity of the original Karatsuba algorithm while reducing the complexity of the extra additions. Furthermore, we propose new matrix multiplication hardware architectures for efficiently exploiting the proposed algorithm in custom hardware, showing that they can provide real area or execution time improvements for integer matrix multiplication compared to designs

implementing scalar Karatsuba or conventional matrix multiplication algorithms. The proposed architectures are well suited for increasing the efficiency in acceleration of modern workloads that can decompose to large matrix multiplications on integer arithmetic, such as the computationally dominant portion of convolutional neural networks or the attention mechanism of transformer models [13]. We provide a complexity analysis of the algorithm and architectures and evaluate the proposed designs both in isolation and in an end-to-end deep learning accelerator system described in Chapter 3 compared to baseline designs and prior state-of-the-art works, showing how they increase the performance-per-area of matrix multiplication hardware.

Chapter 6

Strassen Multi-Systolic-Array Hardware Architectures

In 1969, Strassen proposed a matrix multiplication algorithm [79] that we refer to as SMM showing that the n^3 complexity of the traditional approach is not optimal and the algorithm can theoretically be used to reduce the complexity of naive matrix multiplication. However, as discussed in Section 2.3.3, its execution on general-purpose central processing units (CPU)s and graphics processing units (GPU)s has been shown to be not suitable for achieving the algorithm's promised theoretical speedups, and execution time even worsens unless the matrix widths/heights are in the range of at least 1024 elements or larger, limiting its applicability for modern workloads which do not decompose to such large matrix multiplications. This is because the irregularities introduced in Strassen's algorithm such as the extra data accesses required for reading/computing/storing additional intermediate matrices before/after the matrix multiplication steps all add to the overall execution time beyond what is expected from a theoretical analysis based on the number of arithmetic operations performed alone.

This leaves the question of if the promised theoretical complexity reductions could be more closely achieved in custom hardware architectures designed specifically for executing Strassen's algorithm. However, as discussed in Section 2.3.3, there is limited prior work on this and it is not immediately clear how to derive such architectures or if they can ultimately lead to real improvements. In this chapter, we bridge this gap to present and evaluate new systolic-array hardware architectures for efficiently exploiting Strassen's algorithm. The proposed architectures achieve a more efficient implementation of Strassen's algorithm compared to what is possible through execution on CPUs and GPUs by pipelining and performing the extra data movement and addition steps at all levels of recursion in parallel with the matrix multiplications. The Strassen architectures are functionally equivalent to conventional multi-systolic-array designs while allowing the theoretical complexity reductions of Strassen's algorithm to be translated directly into hardware resource savings, even for multiplication of small matrices. Furthermore, the architectures are multi-systolic-array designs, which is a type of design that can multiply smaller matrices with higher utilization than a single-systolic-array design.

Compared to a conventional multi-systolic-array design, the proposed architecture implemented on FPGA uses approximately 10% fewer soft logic resources and 1.3$\times$ fewer DSP units when instantiated for multiplying matrix sizes down to 24$\times$24 at 2 levels of Strassen recursion. We compare the proposed systolic array architectures in isolation as well as in a larger end-to-end deep learning accelerator system implementation described in Chapter 3 compared to baseline designs and prior works implemented on the same type of compute platform, demonstrating their ability to increase compute efficiency and achieve state-of-the-art performance.

6.1 Strassen Architecture

The proposed Strassen architectures achieve a more efficient implementation of Strassen's algorithm than what is possible through execution on CPUs and GPUs by pipelining and performing the extra additions and data movement steps at all levels of recursion in parallel with the matrix multiplications. The architectures are functionally equivalent to conventional multi-systolic-array designs while allowing the theoretical complexity reductions of Strassen's algorithm to be translated directly into hardware resource savings.

6.1.1 Memory Layout and Access Algorithm

In order to perform the extra Strassen data movement and addition steps at all levels of recursion in parallel with the matrix multiplications, the architecture reads one row/column at a time of the $\mathbf{A}$ and $\mathbf{B}$ input matrix sub-blocks from the lowest level of recursion in (2.24) simultaneously to generate and provide all $\mathbf{T}$ and $\mathbf{S}$ sub-blocks one row/column at a time for performing all the matrix multiplications in (2.25) at the lowest level of recursion in parallel. The $\mathbf{T}$ and $\mathbf{S}$ sub-blocks are all immediately generated from the $\mathbf{A}$ and $\mathbf{B}$ input sub-blocks and consumed in parallel like this to eliminate any additional execution time or hardware resources needed for storing/re-accessing them for later use.

To achieve this, each $\mathbf{A}$ and $\mathbf{B}$ matrix fed into the MXU is divided into 4^r equal sub-blocks of size $m \times k$ for $\mathbf{A}$ and of size $k \times n$ for $\mathbf{B}$, where each row/column i/j of each $\mathbf{A}/\mathbf{B}$ sub-block is stored in the accelerator's $\mathbf{A}$ and $\mathbf{B}$ memories at location i/j plus an offset. An example of this memory layout for implementing 2 levels of Strassen recursion is shown in Fig. 6.1. This means that each $\mathbf{A}$ memory location i is a vector containing every m^{th} row of $\mathbf{A}$ starting at row i concatenated together (notated as $\mathbf{A}_{i:m:,:}$), and each $\mathbf{B}$ memory location j is a vector containing every n^{th} column of $\mathbf{B}$ starting at column j concatenated

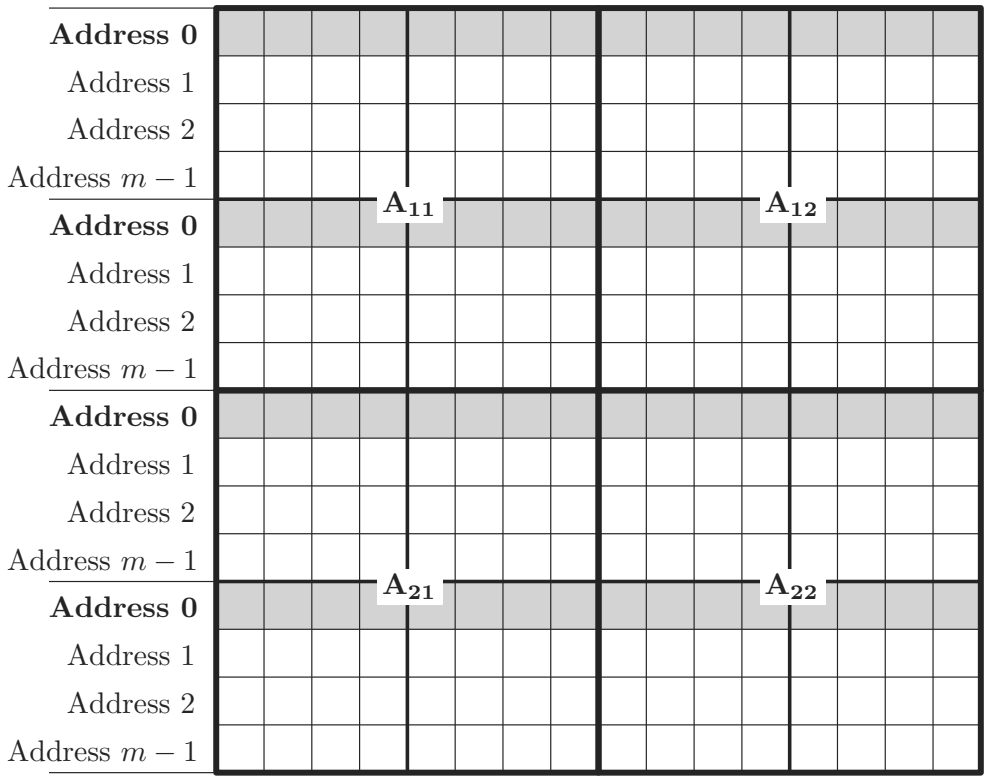

Figure 6.1: Example data layout for the $\mathbf{A}$ matrix in memory for an architecture implementing Strassen matrix multiplication for 2 levels of recursion (SMM_2). Each address i contains every m^{th} row of $\mathbf{A}$ concatenated together starting at row i (notated as $\mathbf{A}_{i:m:,:}$). To help illustrate this, the gray coloured rows are all elements of $\mathbf{A}$ belonging to address 0, which forms $\mathbf{A}_{0:m:,:}$ containing row 0 of every $\mathbf{A}$ sub-block from the lowest level of recursion in (2.24). The organization for the $\mathbf{B}$ matrices in memory are the same, except that the order of the elements is transposed compared to the $\mathbf{A}$ matrix layout shown here.

together (notated as $\mathbf{B}_{:,j:n:}$). This allows one row or column of all 4^r **A/B** sub-blocks from the lowest level of recursion in (2.24) to all be read at once from a single memory location and fed into the MXU each clock cycle. $\mathbf{A}_{i:m:,:}$ and $\mathbf{B}_{:,j:n:}$ rows/columns are then read consecutively when feeding the $\mathbf{A}$ and $\mathbf{B}$ blocks into the MXU.

As shown in (2.24), the input $\mathbf{A}$ and $\mathbf{B}$ matrices at each level of recursion are divided into four block quadrants labelled $\mathbf{A_{ij}}$ and $\mathbf{B_{ji}}$ of size $M \times K$ for $\mathbf{A_{ij}}$ quadrants and of size $K \times N$ for $\mathbf{B_{ji}}$ quadrants. The portions of each $\mathbf{A}_{i:m:,:}$ and $\mathbf{B}_{:,j:n:}$ vector belonging to quadrant $\mathbf{A_{ij}}$ and $\mathbf{B_{ji}}$ are notated as $\mathbf{A_{ij}}_{i:m:,:}$ and $\mathbf{B_{ji}}_{:,j:n:}$. The MXU then computes and returns row i of all $\mathbf{C}$ sub-blocks from the lowest level of recursion in (2.25) in every clock cycle i, allowing $\mathbf{C}_{i:m:,:}$ to be stored in the same format as $\mathbf{A}$ in memory for if $\mathbf{C}$ will later be taken as an $\mathbf{A}$ input for a later matrix multiplication.

6.1.2 Strassen Multi-Systolic-Array Design

Fig. 6.2 shows the proposed SMM$_r$ multi-systolic-array architecture. Rather than having one $X \times Y$ MXU with X columns and Y rows of MAC units for efficiently multiplying matrices down to size $X \times Y$, this architecture consists of 7^r smaller $X/2^r \times Y/2^r$ MXUs that together efficiently multiply matrices down to the same size but at a higher throughput, and it achieves this with fewer MAC units than a conventional multi-systolic-array design. This increases the ratio of throughput strength versus smallest matrix size that can be multiplied at full utilization and increases the throughput per MAC unit.

The $\mathbf{A}_{i:m:,:}$ and $\mathbf{B}_{:,j:n:}$ vectors read into the MXU are first divided into their four $\mathbf{A_{ij}}_{i:m:,:}$ and $\mathbf{B_{ji}}_{:,j:n:}$ portions depending on which quadrant of **A/B** each element belongs to as shown in Fig. 6.2. They then pass through the **A/B** addition vectors shown in Fig. 6.3 (a) and (b) to form the $\mathbf{T}_{i:m:,:}$/$\mathbf{S}_{:,j:n:}$ matrices. The **A/B** addition vectors both contain 5

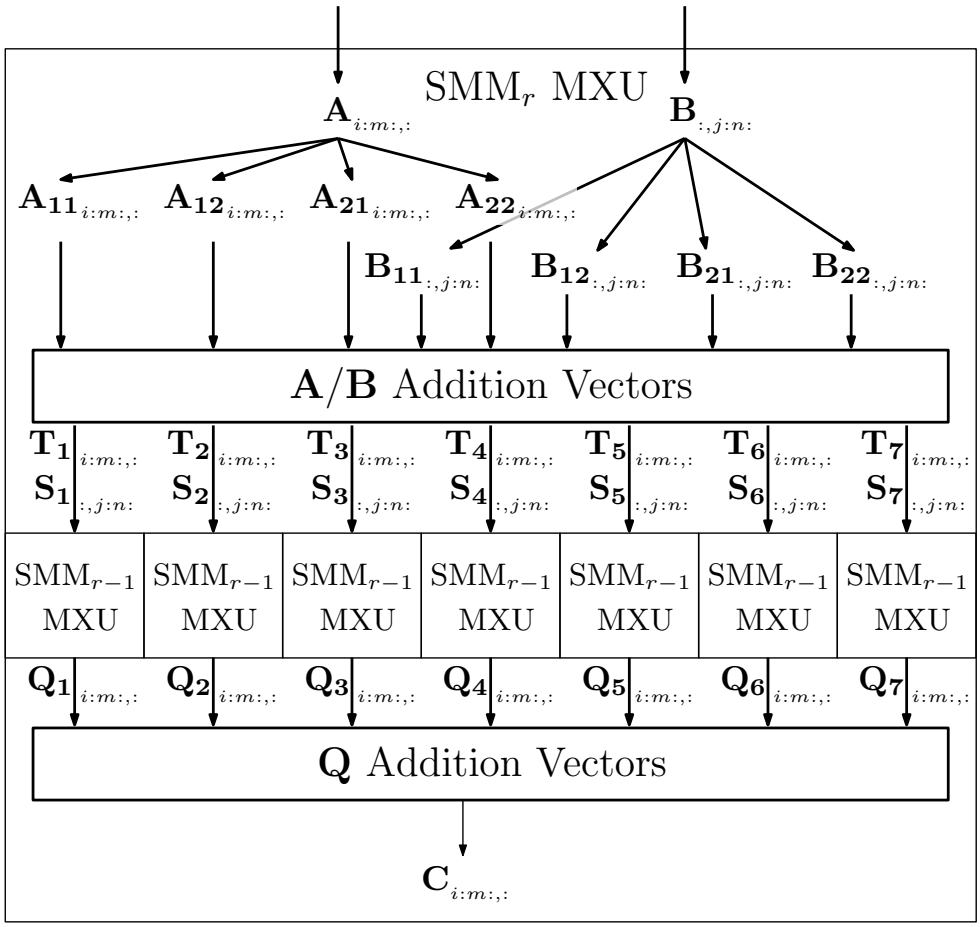

Figure 6.2: Top-level diagram of the proposed multi-systolic-array architecture for implementing r levels of recursion of Strassen matrix multiplication (SMM$_r$).

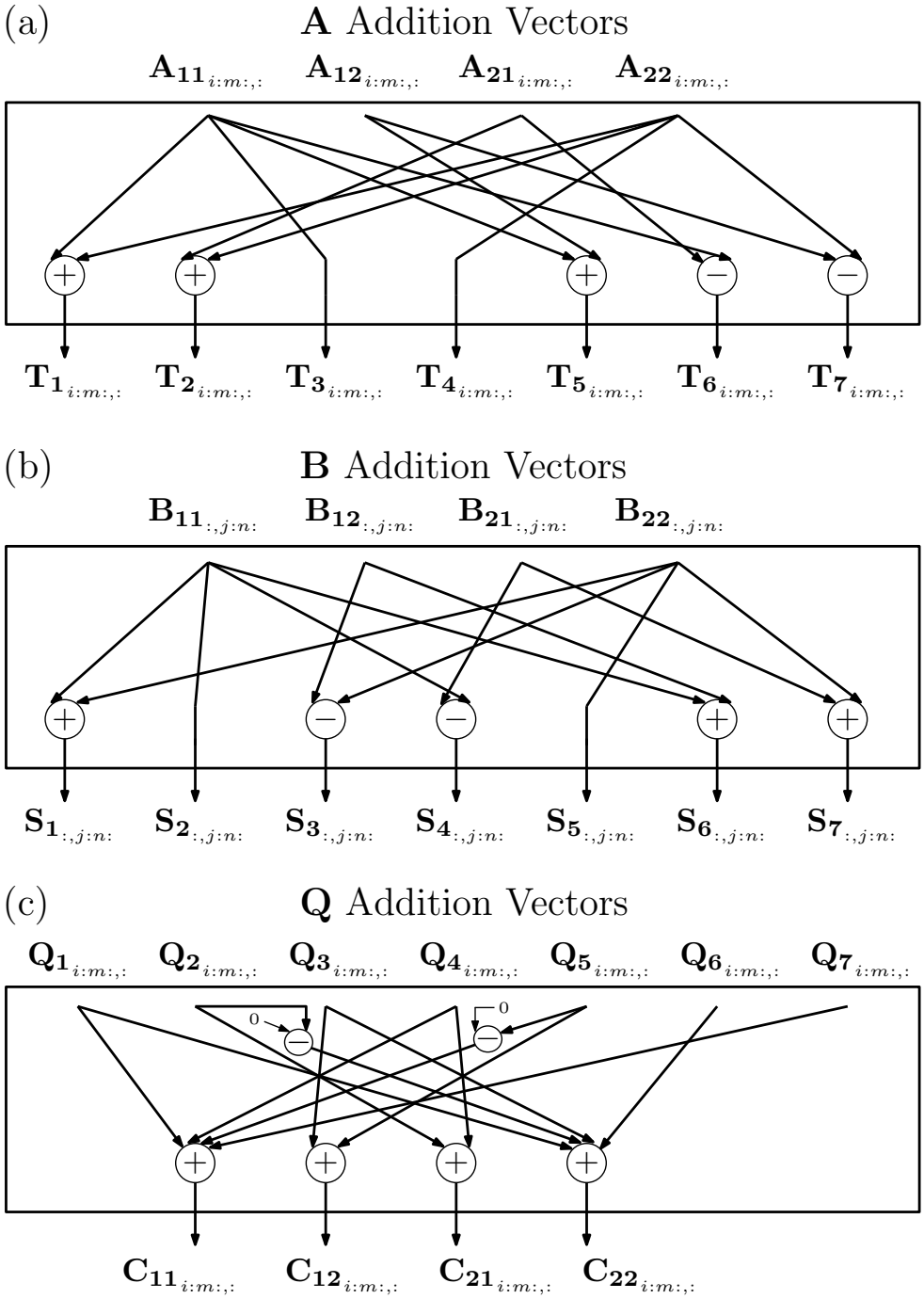

Figure 6.3: Internal structure of the SMM$_r$ MXU addition vectors from Fig. 6.2.

addition vectors each consisting of K scalar adders or subtractors, where K is the width of the four $\mathbf{A_{ij}}$ blocks and the height of the four $\mathbf{B_{ji}}$ blocks as define in Section 6.1.1. The 7 $\mathbf{T}_{i:m:,:}$/$\mathbf{S}_{:,j:n:}$ vectors then pass into the next level of SMM$_{r-1}$ MXUs to perform the 7 matrix block multiplications. The $\mathbf{Q}_{i:m:,:}$ vectors of the matrix block multiplication outputs then pass through the $\mathbf{Q}_{i:m:,:}$ addition vectors shown in Fig. 6.3 (c) consisting of 8 addition vectors each containing N scalar adders or subtractors to form the final $\mathbf{C}$ product, where N is the width of the four $\mathbf{B_{ji}}$ blocks as define in Section 6.1.1.

Each of the 7 SMM$_{r-1}$ MXUs can contain 7 more SMM$_{r-2}$ MXUs for implementing another level of Strassen recursion and repeating the process above, or they can be instantiated as a baseline MM$_0$ MXU shown in Fig. 6.5. For implementing the next level of SMM$_{r-2}$ MXUs inside each SMM$_{r-1}$ MXU, each $\mathbf{T}_{i:m:,:}$/$\mathbf{S}_{:,j:n:}$ input passed into an SMM$_{r-1}$ MXU will then be considered as the full $\mathbf{A}_{i:m:,:}$/$\mathbf{B}_{:,j:n:}$ inputs within that MXU and are split again into the next level of four $\mathbf{A_{ij}}_{i:m:,:}$/$\mathbf{B_{ji}}_{:,j:n:}$ vectors. The dimensions of the matrix blocks being read/computed and the number of scalar adders in the addition vectors within each SMM$_{r-1}$ MXU will then be reduced by a factor of 2 at each level of recursion. For fixed-point implementations, the $\mathbf{T}_{i:m:,:}$/$\mathbf{S}_{:,j:n:}$ inputs to each SMM$_{r-1}$ MXU that were formed from an addition or subtraction in the $\mathbf{A}$ or $\mathbf{B}$ vector addition units will have an increased bitwidth by 1 bit.

6.1.3 Baseline Designs

We later compare the SMM$_r$ architectures with baseline MM$_r$ multi-systolic-array architectures shown in Fig. 6.4 which execute (2.23) in parallel for r levels of recursion. The baseline MM$_r$ architectures are functionally identical to the SMM$_r$ architectures, but they

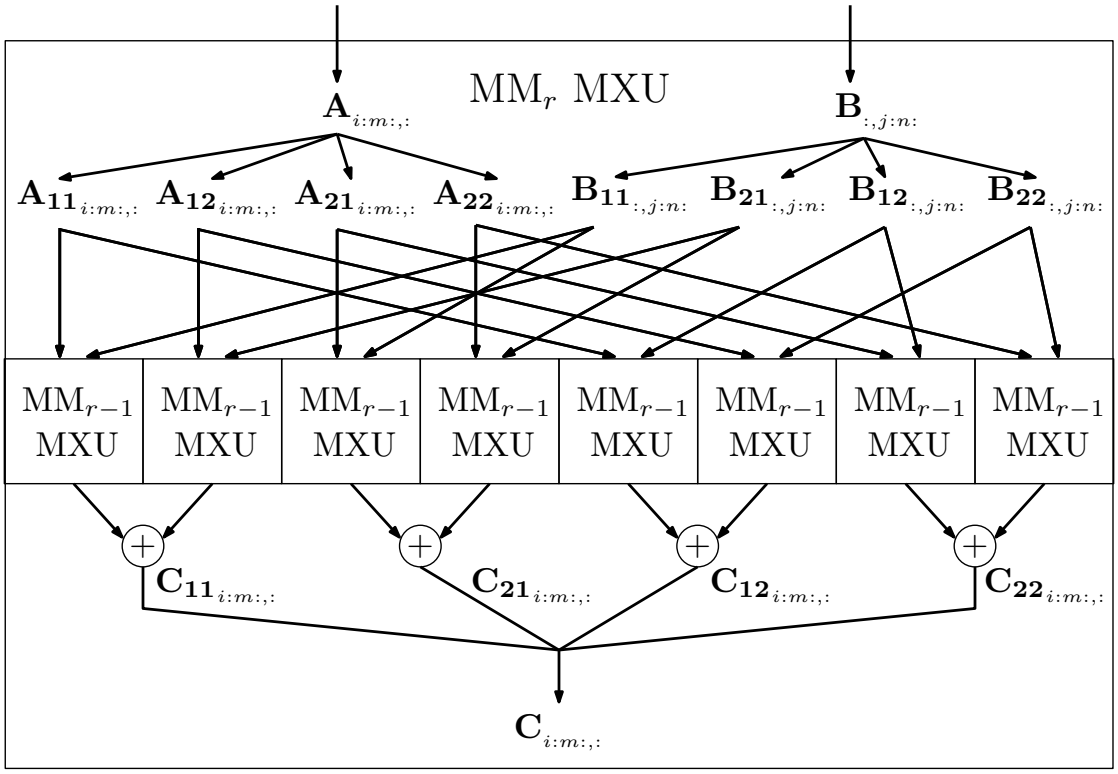

Figure 6.4: Baseline multi-systolic-array architecture for implementing conventional matrix multiplication from (2.23) for r levels of recursion (MM$_r$) in hardware.

consist of 8^r smaller $X/2^r \times Y/2^r$ MXUs rather than 7^r. Fig. 6.5 shows the internal structure of each baseline MM$_0$ MXU present at the lowest level of recursion in each SMM$_r$ and MM$_r$ architecture, and Fig. 6.6 shows the internal structure of the processing elements (PE)s inside the MM$_0$ MXUs.

We later compare the SMM$_r$ architecture with a baseline multi-systolic-array architecture shown in Fig. 6.4 which executes (2.23) in parallel for r levels of recursion. It is functionally identical to the SMM$_r$ MXU but it consists of 8^r smaller $X/2^r \times Y/2^r$ MXUs rather than 7^r. Fig. 6.5 shows the internal structure of each baseline MM$_1$ MXU at the core of each SMM$_r$ architecture, and Fig. 6.6 shows the internal structure of the processing elements (PE)s inside the MM$_1$ MXUs.

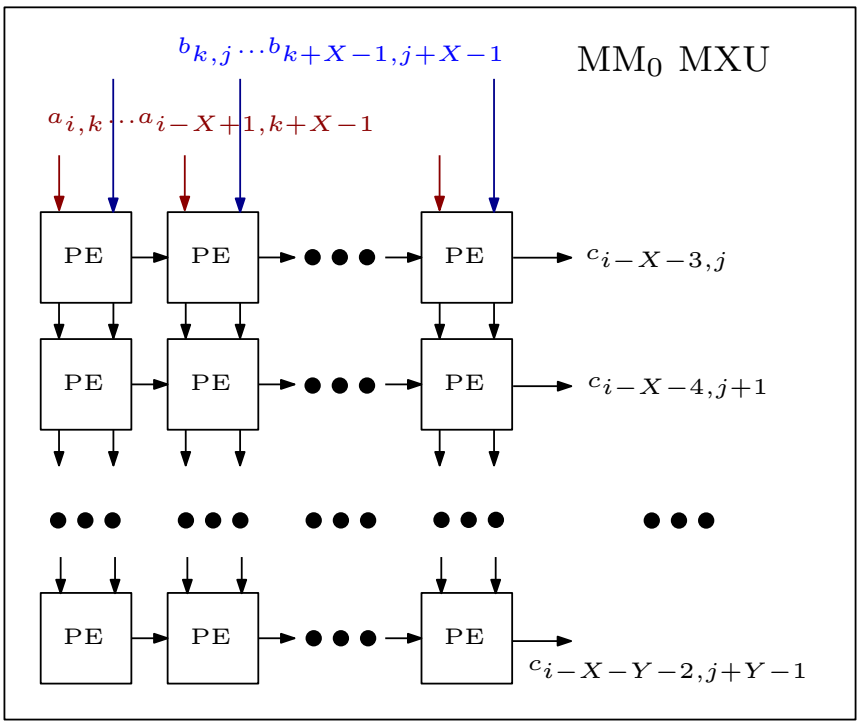

Figure 6.5: Baseline MM_0 single-systolic-array architecture present at the lowest level of recursion in the SMM_r and MM_r MXU architectures, provided for completeness. X here represents the width of the a and b vectors entering the MM_0 MXU, and Y represents the width of the c vectors exiting the MXU.

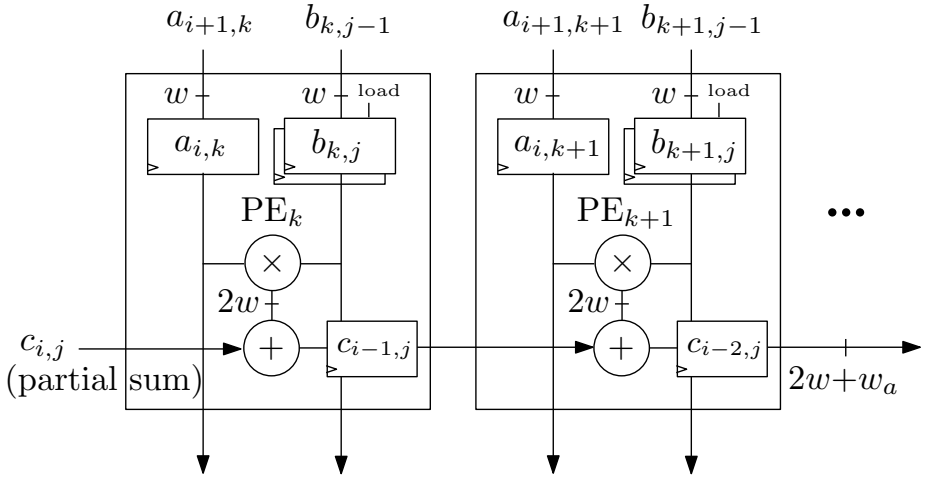

Figure 6.6: The internal PE structure of each MM_0 MXU from Fig. 6.5, provided for completeness. Here, w_a is the additional bitwidth added to account for accumulation, equal to $\lceil \log_2(X) \rceil$, where X is the width of the a and b vectors entering the MM_0 MXU.

6.1.4 System Integration

We evaluate the proposed Strassen systolic-array architectures integrated into the deep learning accelerator system described in Chapter 3 by inserting the SMM$_r$ MXU architectures from Fig. 6.2 into the GEMM Unit in place of the baseline MXU.

In order to perform GEMM on an MXU and multiply matrices of arbitrary sizes that can be larger than the MXU dimensions, the full $\mathbf{A}$ and $\mathbf{B}$ matrices are first divided into GEMM tiles prior to being divided further into smaller blocks for executing (2.23) or (2.24)-(2.25). The GEMM tiles are then fed into the MXU one-by-one. Each GEMM tile is then considered as the full $\mathbf{A}$ and $\mathbf{B}$ matrix from (2.23) or (2.24)-(2.25) while being fed into the MXU and gets further divided into smaller $\mathbf{A_{ij}}$/$\mathbf{B_{ji}}$ blocks within the MXU.

Following each GEMM tile multiplication, the partial GEMM tile products are accumulated outside of the MXU to generate each final GEMM tile product. Prior to each GEMM tile multiplication, a $\mathbf{B}$ GEMM tile is loaded into the MXU. It then remains in place as the $\mathbf{A}$ GEMM tile flows through the MXU producing the GEMM tile product, during which a new $\mathbf{A}_{i:m:,:}$ vector is fed into the MXU each clock cycle. Additionally, to hide the latency of loading $\mathbf{B}$ GEMM tiles, the MXU PEs each contain one extra b buffer to load the next $\mathbf{B}$ GEMM tile into the MXU as the current GEMM tile is being multiplied.

Each $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ sub-block entering or exiting the top-level MXU for the SMM and baseline MXUs first pass through triangular-shaped register arrays each containing X shift registers of varying depths, where each shift register SR_k has a depth of k and loads one $a_{i,k}$ or $b_{k,j}$ element per clock cycle. These triangular buffers are explained further in Chapter 3 and they allow the vector elements to enter the MXU in the necessary order as depicted in the element indices in Figs. 6.5 and 6.6.

6.1.5 Multiplier Compute Efficiency

We use the multiplier compute efficiency defined in (4.18c) as a metric for evaluating the Strassen architectures to quantify how much the algebraic optimizations exploited in an architecture reduce the computational complexity to measure how effectively an architecture can utilize its multipliers relative to a conventional design using no algebraic optimizations.

The limit/maximum achievable value (also referred to as the roof) of the metric in (4.18c) is then the following when executing a conventional matrix multiplication algorithm such as (2.23) (MM_r) in hardware since it has no algebraic optimizations for reducing the computational complexity:

$$\text{MM}_r \ \frac{\text{mults/multiplier}}{\text{clock cycle}} \ \text{roof} = 1 \, . \tag{6.1}$$

In contrast, Strassen's algorithm (SMM) requires $8^r/7^r$ times fewer multiplications than a conventional MM_1 algorithm, where r is the number of levels of recursion implemented in Strassen's algorithm. Therefore, the multiplier compute efficiency can reach the following limit in SMM_r architectures:

$$\text{SMM}_r \ \frac{\text{mults/multiplier}}{\text{clock cycle}} \ \text{roof} = \left(\frac{8}{7}\right)^r \, . \tag{6.2}$$

One of the impediments of using Strassen's algorithm for fixed-point implementations is that the bitwidths of the multiplication inputs increase by r bits for r levels of Strassen recursion that are implemented, reducing its potential area savings for custom fixed-point hardware designs. Nonetheless, this impediment for fixed-point designs can be inherently mitigated in FPGA implementations so long as r plus the initial input width is not larger than the maximum input width supported by the FPGA's DSP units. For example, each DSP

in common Intel/Altera FPGAs instantiate two 18×19-bit multipliers [94], and common input bitwidths for applications such as deep learning are 16 bits or less, leaving room for at least 2 or more levels of Strassen recursion to be implemented before surpassing the bitwidth limit supported by the DSPs.

6.1.6 Supporting Smaller Matrices with the Same Performance

Multi-systolic-array designs such as the SMM$_r$ and baseline MM$_r$ architectures have the ability to efficiently multiply smaller matrices than a single-systolic-array design with the same performance capability. By executing (2.23) or (2.24)-(2.25) fully in parallel for r levels of recursion, matrix products of size as small as $n \times n$, which require n^3 multiplications to calculate using conventional algebra, can be computed up to once every $n/2^r$ clock cycles in an MM$_r$ or SMM$_r$ multi-systolic-array design. Therefore, the ratio of an architecture's throughput per clock cycle roof versus its smallest supported matrix sizes it can multiply is:

$$\text{(S)MM}_r \ \frac{\text{mults/clock cycle}}{\text{min. mat. size (h}\times\text{w)}} \ \text{roof} = \frac{n^3/(n/2^r)}{n \times n} = 2^r \,. \tag{6.3}$$

In contrast, a single-systolic-array design can produce matrix products of size as small as $n \times n$ up to once every n clock cycles, making this ratio the following for a single-systolic-array design:

$$\text{MM}_0 \ \frac{\text{mults/clock cycle}}{\text{min. mat. size (h}\times\text{w)}} \ \text{roof} = \frac{n^3/n}{n \times n} = 1 \,. \tag{6.4}$$

This shows that the SMM$_r$ and baseline MM$_r$ multi-systolic-array designs can efficiently multiply matrices 2^r times smaller than a single-systolic-array architecture with the same

performance capability.

As discussed in Section 2.2.1, this is an important property for increasing a systolic-array accelerator's maximum achievable throughput on real-life workloads because, even if more compute resources are instantiated to scale up the size of the systolic array, the systolic array will begin to be underutilized after its size surpasses the workload's matrix sizes. This is particularly true in modern workloads such as deep learning acceleration, where the matrix sizes that the workloads break down to can be smaller than the maximum systolic array size that could be instantiated in an accelerator [5], [6], [18], [19]. In Section 6.2.2, we demonstrate how this property allowed us to scale up our deep learning accelerator design without compromising utilization to achieve state-of-the-art ResNet [17] throughput.

6.2 Results

In this section, we evaluate example implementations of the proposed SMM architectures. These implementations are validated on FPGA, and we therefore compare against state-of-the-art prior works that are also evaluated on the same FPGA family as ours. We compare the SMM MXU architectures in isolation against our baseline MXU designs in Table 6.1, and in Tables 6.2-6.3 we evaluate the SMM MXU architectures compared to prior work when integrated into an end-to-end deep learning accelerator system described in Chapter 3.

Full system-level validation of the experimental accelerator as integrated into the the deep learning accelerator system described in Chapter 3 has been done on an Arria 10 SoC Developement Kit [92] containing the Arria 10 SX 660 device by measuring throughput in real-time. However, this device contains fewer soft logic resources than the Arria 10 GX

1150 used in the prior works we compare against, and we generate compilation results for our design on the same Arria 10 GX 1150 device used in prior works for a more fair and consistent comparison. Throughput values of our designs on the Arria 10 GX 1150 device are then calculated using an accurate throughput estimation model based on our highly deterministic and time-predictable system implementation, which accurately predicts actual throughputs measured on the Arria 10 SX 660 device available to us. Tables 6.2-6.3 show throughputs for ResNet [17] neural networks.

In Table 6.2, the number of multipliers in the work from An et al. [97] is equal to #DSPs×2, where each DSP in the Intel/Altera FPGAs instantiates two 18×19-bit multipliers [94]. The works from Liu et al. [95] and Fan et al. [96] in Table 6.2 use a technique to pack two 8-bit multiplications onto each 18×19-bit multiplier in the DSPs and additional ALMs, and therefore #multipliers = #DSP×4 in those works. The number of multipliers used in the MXUs from our architectures in Tables 6.1-6.2 is equal to 8^r or 7^r times $X \times Y$ for the MM$_r$ and SMM$_r$ MXUs, respectively. For example, an MM$_0$ 64×64 MXU (meaning $r = 0$ and $X = Y = 64$) would contain $8^0 \times 64^2$ multipliers, an MM$_1$ 32×32 MXU would contain $8^1 \times 32^2$ multipliers, and an SMM$_2$ 8×8 MXU would contain $7^2 \times 8^2$ multipliers. Due to the FFIP reduction in multipliers as described in our prior work from Chapter 4, the number of multipliers for the FFIP architectures in Table 6.3 is equal to 8^r or 7^r times $X \times Y/2 + X/2$ for the FFIP and FFIP+SMM$_r$ designs, respectively. Additionally, for our deep learning accelerator implementations in Tables 6.2-6.3, there are an additional $Y \times 4^r$ multipliers located outside the MXU in the Post-GEMM Unit for performing inter-layer quantization rescaling functions. For our designs requiring more than 3036 multipliers, 3036 are instantiated on 1518 DSPs, and the remainder are instantiated using soft logic resources due to the DSP resources being fully utilized.

Table 6.1: Comparison of SMM_r multi-systolic-array architectures against the baseline MM_0 single-systolic-array architecture and baseline MM_r multi-systolic-array architectures. These results contain the systolic arrays in isolation (without integration into a deep learning accelerator system).

	MM_0 48×48	MM_1 16×16	SMM_1 16×16	MM_2 6×6	SMM_2 6×6
FPGA	Arria 10 GX 1150	Arria 10 GX 1150	Arria 10 GX 1150	Arria 10 GX 1150	Arria 10 GX 1150
ALMs	35,259	33,212	32,287	48,266	42,697
Registers	133,799	129,830	122,365	170,386	158,389
Memories	0	0	0	0	0
DSPs	1,152	1,024	896	1,152	882
Input bitwidth	16	16	16	16	16
Frequency (MHz)	383	364	332	312	285
Throughput roof (GOPS) [1]	1765	1491	1360	1468	1313
Min. supported matrix size [2]	48×48	32×32	32×32	24×24	24×24
$\frac{\text{mults/multiplier}}{\text{clock cycle}}$ roof [3]	1	1	1.14	1	1.31
$\frac{\text{mults/clock cycle}}{\text{min. mat. size (h×w)}}$ roof [4]	1	2	2	4	4

[1] Maximum achievable throughput in giga operations per second, where throughput is equal to the number of operations required to carry out an execution using conventional algebra divided by the measured execution time.
[2] Minimum input matrix sizes that can be multiplied at peak throughput/full utilization.
[3] Maximum achievable multiplier compute efficiency, defined in Section 4.2.5, measures how effectively the architecture can utilize its multipliers. It can surpass 1 in SMM architectures because the observed mults/s is equal to the number of multiplications required to carry out an execution using conventional algebra divided by execution time.
[4] Quantifies how much smaller the minimum supported matrix sizes of a multi-systolic-array design are relative to a single-systolic-array design with the same throughput per clock cycle roof, definition and relevance provided in Section 6.1.6.

6.2.1 Comparison to Baseline Designs

Table 6.1 shows the resource usage and performance comparison between the proposed SMM_r and the baseline MM_r systolic-array architectures in isolation (without integration into a deep learning accelerator system). Compared to the multi-systolic-array MM_1 and MM_2 designs, the SMM_1 and SMM_2 architectures are functionally equivalent, respectively, other than having a 9% lower clock frequency, which may not be a significant disadvantage when considering that the frequency-limiting critical path may be in external control or

other logic outside of the systolic array anyways after integrating it into an end-to-end accelerator system. However, while the MM_1 and MM_2 architectures have the same throughput per clock cycle roof as the SMM_1 and SMM_2 architectures, respectively, they require 3%-13% more ALMs, 6%-8% more registers, and 14%-31% more DSP units, respectively.

The throughput per clock cycle roof of the MM_0 and MM_2 baseline designs in Table 6.1 are equal and they consume the same number of DSP resources, but the MM_0 design requires fewer ALM and register resources. However, this penalty may be justified in the MM_2 design when considering that the minimum matrix size (height$\times$width) that can be multiplied while fully utilizing the MXU is $4\times$ smaller in the MM_2 design compared to the MM_0 design, which increases its performance scalability for accelerating modern workloads such as deep learning as discussed in Section 2.2.1 and 6.1.6. This same property is true for the SMM_2 design, except it achieves this with fewer DSP and soft logic resources. This benefit is demonstrated in Section 6.2.2, where this property allowed us to scale up our deep learning accelerator design without compromising utilization to achieve state-of-the-art ResNet throughput.

6.2.2 Comparison to Prior Work

Tables 6.2-6.3 show the SMM_r architectures integrated into the deep learning system from Chapter 3 compared to state-of-the-art accelerators evaluated on the same FPGA family for the same input bitwidths and similar neural network models. Integrating the SMM_r multi-systolic-array design into our deep learning accelerator allowed us to increase the multiplier compute efficiency while also scaling up the computational resources and throughput roof without increasing the minimum supported matrix sizes, allowing it to significantly surpass the throughput in our prior work from Chapter 4 and other state-of-the-art prior works

Table 6.2: SMM_r multi-systolic-array architectures integrated into a deep learning accelerator system compared with prior state-of-the-art deep learning accelerators on the same FPGA.

	TNNLS '22 [95]		TCAD '22 [96]		Entropy '22 [97]		SMM_1 32×32			SMM_2 8×8		
FPGA	Arria 10 GX 1150		Arria 10 GX 1150		Arria 10 GX 1150		Arria 10 GX 1150			Arria 10 GX 1150		
ALMs	304K		304K		303K		306K			145K		
Registers	889K		890K		-		641K			386K		
Memories	2334		2334		1953		2713			2036		
DSPs	1473		1473		1503		1518			1518		
Frequency (MHz)	200		220		172		293			295		
Input bitwidth	8	8	8	8	8	8	8	8	8	8	8	8
Model	ResNet-50	VGG 16	Bayes ResNet-18	Bayes VGG 11	R-CNN (ResNet-50)	R-CNN (VGG 16)	ResNet-50	ResNet-101	ResNet-152	ResNet-50	ResNet-101	ResNet-152
Throughput (GOPS)[1]	1519	1295	1590	534	719	865	3750	4116	4276	2024	2115	2158
$\frac{\text{mults/multiplier}}{\text{clock cycle}}$ [2]	0.645	0.550	0.639	0.206	0.696	0.837	0.877	0.963	1.002	1.051	1.098	1.120

[1] Throughput in giga operations per second, equal to the number of operations required to carry out an execution using conventional algebra divided by execution time.

[2] Multiplier compute efficiency, defined in Section 4.2.5, measures how effectively the architecture utilizes its multipliers. It can surpass 1 in SMM architectures because the observed mults/s is equal to the number of multiplications required to carry out an execution using conventional algebra divided by the measured execution time.

Table 6.3: Comparison of an FFIP single-systolic-array architecture from Chapter 4, which doubles performance per MAC unit, with combined FFIP+SMM$_r$ multi-systolic-array architectures when integrated into deep learning accelerator systems.

	TC '24 [8] (FFIP 64×64)			FFIP+SMM$_1$ 32×32			FFIP+SMM$_2$ 8×8		
FPGA	Arria 10 GX 1150			Arria 10 GX 1150			Arria 10 GX 1150		
ALMs	118K			216K			165K		
Registers	311K			627K			463K		
Memories	1782			2713			2036		
DSPs	1072			1518			946		
Frequency (MHz)	388			313			297		
Input bitwidth	8	8	8	8	8	8	8	8	8
Model	ResNet-50	ResNet-101	ResNet-152	ResNet-50	ResNet-101	ResNet-152	ResNet-50	ResNet-101	ResNet-152
Throughput (GOPS)[1]	2529	2752	2838	4006	4397	4568	2038	2130	2172
$\frac{\text{mults/multiplier}}{\text{clock cycle}}$ [2]	1.521	1.655	1.707	1.674	1.837	1.908	1.813	1.895	1.933

[1] Throughput in giga operations per second, equal to the number of operations required to carry out an execution using conventional algebra divided by execution time.
[2] Multiplier compute efficiency, defined in Section 4.2.5, measures how effectively the architecture utilizes its multipliers. It can surpass 1 in SMM architectures because the observed mults/s is equal to the number of multiplications required to carry out an execution using conventional algebra divided by the measured execution time.

evaluated on the same FPGA family as shown in Tables 6.2-6.3. If the design is scaled up using a single systolic array, the minimum supported matrix size increases and compute resources begin to be underutilized for ResNet execution based on the smaller matrix sizes that its workload decomposes to, and the effective throughput does not increase well despite the design having a larger throughput roof.

The SMM_1 32×32 and FFIP+SMM_1 32×32 designs consume noticeably more memory resources than the SMM_2 8×8 and FFIP+SMM_2 8×8 designs. However, it is worth noting that this is not due to increased memory requirements, but rather is due to the compiler favouring to swap some register resources for memory resources since the SMM_1 32×32 and FFIP+SMM_1 32×32 designs have a higher register (and overall area) overhead than the SMM_2 8×8 and FFIP+SMM_2 8×8 designs in order to achieve higher throughput roofs.

In Table 6.2, the SMM_r architectures achieve the highest throughput and multiplier compute efficiency compared to the prior works. The SMM_1 and SMM_2 architectures' multiplier compute efficiencies in Table 6.2 approach their limits of 1.14 and 1.31 that are derived in (6.2) which surpass the limit of 1 of the baseline MM_r architectures and prior works that is derived in (6.1), validating SMM's ability to increase multiplier compute efficiency and reduce computational complexity as expected from our analysis.

6.2.3 Combining FFIP and SMM

Table 6.3 shows an example of how SMM can be combined with other algebraic techniques to further increase multiplier compute efficiency limits. FFIP, described in Chapter 4, provides a way to reduce the number of required multiplications by up to a factor of 2, trading half the multiplications for cheap low-bitwidth additions. Because of this, the limit for the multiplier compute efficiency metric in (4.18c) for an FFIP architecture becomes 2,

and $2 \times (8/7)^r$ for a combined FFIP+SMM$_r$ architecture. In Table 6.3, we evaluate architectures that combine FFIP+SMM$_r$ by instantiating SMM$_r$ MXUs that use FFIP MXUs at their lowest level of recursion instead of the conventional MM$_0$ MXUs from Fig. 6.5 to further increase multiplier compute efficiency compared to a standalone SMM$_r$ or standalone FFIP MXU as seen in the achieved multiplier compute efficiencies of the FFIP+SMM$_r$ architectures listed in Table 6.3.

6.3 Summary

Strassen's fast matrix multiplication algorithm reduces the complexity of naive matrix multiplication, however, general-purpose hardware is not suitable for achieving the algorithm's promised theoretical speedups, and there is limited prior work on custom hardware architectures designed specifically for executing the algorithm in hardware. We address this by presenting custom Strassen multi-systolic-array architectures that are functionally equivalent to conventional multi-systolic-array designs while allowing the theoretical complexity reductions of Strassen's algorithm to be translated directly into hardware resource savings even for multiplication of smaller matrices. Compared to a conventional multi-systolic-array design, the proposed architecture implemented on FPGA uses approximately 10% fewer soft logic resources and 1.3$\times$ fewer DSP units when instantiated for multiplying matrix sizes down to 24$\times$24 at 2 levels of Strassen recursion. The proposed systolic array architectures are compared in isolation as well as in a larger end-to-end accelerator system design compared to baseline designs and prior works implemented on the same type of compute platform, demonstrating their ability to increase compute efficiency and achieve state-of-the-art performance.

Chapter 7

Conclusion

Recent years have seen increasing breakthroughs and commercial adoption of deep learning, which has enabled ground-breaking applications ranging from human-like chatbots like ChatGPT, to generating realistic image and video content from prompts, self-driving cars [1], detecting cancer [2], and beating human champions at the complex game of Go [3]. However, deep learning inference increasingly requires massive amounts of computational power to perform, making it ever-more difficult to execute quickly and efficiently. This can be mitigated by building special-purpose computer hardware that can perform deep learning inference more efficiently than general-purpose hardware like conventional central processing units (CPU)s.

To address this need, the field of deep learning acceleration has seen many recent works on system-level improvements for deep learning hardware acceleration and hardware-oriented deep neural network (DNN) model optimizations. At a certain point, however, after hardware-oriented DNN model optimizations reach their limit, after the known parallelism and system-level optimizations for executing their compute patterns are exploited, and after technology scaling slows to a halt, there is an accelerator wall which causes limited improvement on

the implementation side [69]. A less-explored avenue to continue advancement after this point is to reduce the workload at the algebraic level, by calculating the same deep learning model algebra, nevertheless using a re-arranged compute pattern which produces the same output from fewer or cheaper operations performed in hardware.

As discussed in Section 2.1, the majority of the computational workload in deep learning models can commonly be mapped to the matrix multiplication shown in (2.17), and as can be seen, the operations in this equation are a series of multiply-accumulate operations. For all deep learning accelerators, unless additional algebraic innovations are used, the throughput is ultimately limited by the maximum number of multiply-accumulate operations from (2.17) that can be performed per clock cycle. Due to this, deep learning accelerators contain a large number of MAC units, causing multipliers and MAC units to commonly be one of the area-dominant resources in GEMM and deep learning accelerators [70], [3], [5], and an accelerator's throughput can be directly limited by how many multipliers its hardware budget can afford.

As a result, surpassing this theoretical performance per multiplier limit should be a key area of interest for advancing the field of deep learning hardware acceleration. As discussed in Section 2.2.8, this approach has been touched upon with Winograd's minimal filtering algorithms applied to convolutional neural networks (CNN)s [65], [64], [66], [70]. However, this algorithm is applicable to a limited range of DNN model types, and there are numerous other efficient algebraic algorithms that remain under-explored which are also applicable to GEMM and therefore a broader range of DNN models.

In this thesis, we continue in this under-explored direction and provide advancements to several more efficient algebraic algorithms and/or their custom hardware implementations for the application of deep learning acceleration.

7.1 Summary of Contributions

Chapter 4 presents an algorithm and general architecture that improve Winograd's under-explored inner-product algorithm [7] that can be seamlessly incorporated into any deep learning accelerator system that uses traditional fixed-point systolic arrays to double the throughput per MAC unit, significantly increasing the accelerator's performance per compute area for inference of all deep learning models that will execute on the systolic array. We implement and evaluate FIP for the first time in a deep learning accelerator system described in Chapter 3. We then identify a weakness of FIP and propose the new FFIP algorithm and generalized hardware architecture that inherently address that weakness in the general case. We provide deep learning-specific optimizations for the FIP and FFIP algorithms and systolic array hardware architectures. We derive how the (F)FIP architectures increase the theoretical compute efficiency and performance limits in the general case.

Chapter 5 proposes an algorithm and its hardware architectures that extend the Karatsuba algorithm [9] to matrix multiplication. While the Karatsuba algorithm reduces the complexity of large integer multiplication, the extra additions required minimize its benefits for smaller integers of more commonly-used bitwidths. In this chapter, we propose the extension of the scalar Karatsuba multiplication algorithm to matrix multiplication, showing how this maintains the reduction in multiplication complexity of the original Karatsuba algorithm while reducing the complexity of the extra additions. Furthermore, we propose new matrix multiplication hardware architectures for efficiently exploiting this extension of the Karatsuba algorithm in custom hardware. We show that the proposed algorithm and hardware architectures can provide real area or execution time improvements for integer matrix multiplication compared to scalar Karatsuba or conventional matrix multiplication

algorithms, while also supporting implementation through proven systolic array and conventional multiplier architectures at the core. We provide a complexity analysis of the algorithm and architectures and evaluate the proposed designs both in isolation and in an end-to-end deep learning accelerator system described in Chapter 3 compared to baseline designs and prior state-of-the-art works implemented on the same type of compute platform, demonstrating their ability to increase the performance-per-area of matrix multiplication hardware.

Chapter 6 explores hardware architectures for exploiting Strassen's fast matrix multiplication algorithm. While Strassen's matrix multiplication algorithm reduces the complexity of naive matrix multiplication, general-purpose hardware is not suitable for achieving the algorithm's promised theoretical speedups, leaving the question of if it could be more efficiently exploited in custom hardware architectures designed specifically for executing the algorithm. However, there is limited prior work on this and it is not immediately clear how to derive such architectures or if they can ultimately lead to real improvements. We bridge this gap, presenting and evaluating new systolic-array architectures that efficiently translate the theoretical complexity reductions of Strassen's algorithm directly into hardware resource savings. Furthermore, the architectures are multi-systolic-array designs that can multiply smaller matrices with higher utilization than single-systolic-array designs. The proposed design implemented on FPGA for multiplying matrix sizes down to 24×24 at 2 levels of Strassen recursion uses approximately 10% fewer soft logic resources and $1.3\times$ fewer DSP units than a conventional multi-systolic-array design. We evaluate the proposed Strassen systolic arrays in isolation as well as in an end-to-end deep learning accelerator system described in Chapter 3 compared to baseline designs and prior works implemented on the same type of compute platform, demonstrating their ability to increase compute

efficiency and achieve state-of-the-art performance.

7.2 Future Work

The proposed advancements in efficient matrix multiplication algorithms and/or their hardware architectures prompt a shift in focus from many previous approaches for deep learning acceleration, and as a result, open further exciting opportunities for new exploration in this direction.

7.2.1 Floating-Point Algorithms and Architectures

We primarily focus on algorithms and architectures for fixed-point/integer data types, leaving opportunities for exploration of algorithms and architectures for floating-point data types. The benefits of (F)FIP in hardware rely on the premise that the hardware footprint of adders are cheaper than that of multipliers. Since the arithmetic complexity of fixed-point multipliers typically scale quadratically with the input bitwidth compared to linearly for adders, this premise has been shown to hold true for fixed-point data types [71], [25], [72]. There may be doubts about the benefits of this for floating-point data types because both floating-point adders and multipliers contain variable shifter circuits, which do not have an insignificant hardware footprint relative to the fixed-point multiplier portion of a floating-point multiplier circuit. However, this still deserves further investigation to confirm. It may also be possible to extend the KMM algorithm to floating-point data types.

Additionally, it is well known that the SMM algorithm works on floating-point data types. For fixed-point data, SMM increases the bitwidth of the multiplications by r bits, where r is the number of implemented Strassen recursion levels. This reduces the area

benefits for fixed-point ASIC implementations but does not affect floating-point implementations. Therefore, another possible research direction is to evaluate implementations of the proposed SMM architecture for floating-point data types.

7.2.2 Toom-Cook Matrix Multiplication

Karatsuba scalar multiplication is a specific case of the Toom-Cook scalar multiplication algorithm [109] for when the two numbers being multiplied each consist of 2 digits. The Toom-Cook algorithm generalizes this to provide reduced-complexity algorithms for multiplying n-digit numbers. This leaves opportunities to explore the extension of the Toom-Cook algorithm to matrix multiplication similarly to the methods proposed in Chapter 5 for extending the Karatsuba algorithm to matrix multiplication, and to evaluate any additional benefits this may bring when implementing the algorithm in hardware.

7.2.3 Non-Systolic-Array Architectures

The proposed architectures are systolic arrays, however, the benefits of the explored algorithms in hardware are not restricted to being realized only in systolic arrays. Therefore, we encourage others to explore non-systolic array hardware implementations of these algorithms. Non-systolic array architectures may have the benefit of supporting vector operations in addition to strictly matrix multiplications, which would allow higher utilization for multiplication of smaller matrix sizes.

7.2.4 Transformer Acceleration

Transformer models, introduced in 2017 [13], are a more recent type of deep learning model that have since been shown to be superior in quality in many popular benchmarks

compared to prior deep learning models like CNNs, and it is the base model used in popular works such as BERT [14] and GPT [15] models. The computationally intensive portion of transformer models are based around a type of layer called multi-head attention, which mainly consists of a sequence of large matrix multiplications. The proposed hardware architectures are well suited for accelerating large matrix multiplications such as those in the attention mechanism of transformer models, which can be quantized to integer arithmetic, leaving this as an area of interest for future work.

7.3 Concluding Remarks

The proposed algorithms and hardware architectures surpass the traditional theoretical compute efficiency and performance limits, and our end-to-end results show that they improve performance-per-area compared to prior state-of-the-art solutions. Most importantly, our results indicate that the proposed hardware architectures, when overlaid on top of the most efficient systolic-array systems used in practice, can further increase compute efficiency across a wide range of devices, system implementations, and deep learning models.

Bibliography

[1] Chenyi Chen et al. “Deepdriving: Learning Affordance for Direct Perception in Autonomous Driving”. In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 2722–2730.

[2] Andre Esteva et al. “Dermatologist-Level Classification of Skin Cancer with Deep Neural Networks”. In: *Nature* 542.7639 (2017), pp. 115–118.

[3] Norman P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 1–12. DOI: 10.1145/3079856.3080246.

[4] 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.

[5] Thomas Norrie et al. “The Design Process for Google’s Training Chips: TPUv2 and TPUv3”. In: *IEEE Micro* 41.2 (2021), pp. 56–63. DOI: 10.1109/MM.2021.3058217.

[6] Norman Jouppi et al. “TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings”. In: *Proceedings of*

the 50th Annual International Symposium on Computer Architecture. 2023, pp. 1–14.

[7] S. Winograd. "A New Algorithm for Inner Product". In: *IEEE Transactions on Computers* C-17.7 (1968), pp. 693–694. DOI: 10.1109/TC.1968.227420.

[8] Trevor E. Pogue and Nicola Nicolici. "Fast Inner-Product Algorithms and Architectures for Deep Neural Network Accelerators". In: *IEEE Transactions on Computers* 73.2 (2024), pp. 495–509. DOI: 10.1109/TC.2023.3334140.

[9] Anatolii Alekseevich Karatsuba and Yu P. Ofman. "Multiplication of Many-Digital Numbers by Automatic Computers". In: *Doklady Akademii Nauk*. Vol. 145. 1962, pp. 293–294.

[10] 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)*. 2010, pp. 807–814.

[11] Alex Krizhevsky et al. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems* 60.6 (2017), pp. 84–90.

[12] Sharan Chetlur et al. "cuDNN: Efficient Primitives for Deep Learning". In: *arXiv preprint arXiv:1410.0759* (2014). DOI: 10.48550/arXiv.1410.0759. arXiv: 1410.0759.

[13] Ashish Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. 2017, pp. 5998–6008.

[14] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *arXiv preprint arXiv:1810.04805* (2018). DOI: 10.48550/arXiv.1810.04805. arXiv: 1810.04805.

[15] Josh Achiam et al. “GPT-4 Technical Report”. In: *arXiv preprint arXiv:2303.08774* (2023). DOI: 10.48550/arXiv.2303.08774. arXiv: 2303.08774.

[16] Chen Zhang et al. “Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.11 (2019), pp. 2072–2085. DOI: 10.1109/TCAD.2017.2785257.

[17] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[18] H. T. Kung et al. “Maestro: A Memory-on-Logic Architecture for Coordinated Parallel Use of Many Systolic Arrays”. In: *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Vol. 2160-052X. 2019, pp. 42–50. DOI: 10.1109/ASAP.2019.00-31.

[19] Ahmet Caner Yüzügüler et al. “Scale-out Systolic Arrays”. In: *ACM Transactions on Architecture and Code Optimization* 20.2 (Mar. 2023). ISSN: 1544-3566. DOI: 10.1145/3572917.

[20] Benoit Jacob et al. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 2704–2713.

[21] Yangqing Jia et al. “Caffe: Convolutional Architecture for Fast Feature Embedding”. In: *Proceedings of the 22nd ACM International Conference on Multimedia*. 2014, pp. 675–678. DOI: 10.1145/2647868.2654889.

[22] Itay Hubara et al. “Binarized Neural Networks”. In: *Advances in Neural Information Processing Systems*. Vol. 29. 2016.

[23] 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*. 2017, pp. 65–74. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021744.

[24] Renzo Andri et al. “YodaNN: An Architecture for Ultralow Power Binary-Weight CNN Acceleration”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.1 (2018), pp. 48–60. DOI: 10.1109/TCAD.2017.2682138.

[25] Kaiyuan Guo et al. “[DL] A Survey of FPGA-based Neural Network Inference Accelerators”. In: *ACM Transactions on Reconfigurable Technology and Systems* 12.1 (2019), pp. 1–26. DOI: 10.1145/3289185.

[26] Erwei Wang et al. “LUTNet: Learning FPGA Configurations for Highly Efficient Neural Network Inference”. In: *IEEE Transactions on Computers* 69.12 (2020), pp. 1795–1808. DOI: 10.1109/TC.2020.2978817.

[27] Eriko Nurvitadhi et al. “Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?” In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 5–14. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021740.

[28] Kai Li et al. “A Precision-Scalable Energy-Efficient Bit-Split-and-Combination Vector Systolic Accelerator for NAS-optimized DNNs on Edge”. In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022, pp. 730–735. DOI: 10.23919/DATE54114.2022.9774679.

[29] Wenjie Li et al. “Low-Complexity Precision-Scalable Multiply-Accumulate Unit Architectures for Deep Neural Network Accelerators”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* (2022). DOI: 10.1109/TCSII.2022.3231418.

[30] Yaman Umuroglu et al. “BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing”. In: *2018 28th International Conference on Field-Programmable Logic and Applications (FPL)*. 2018, pp. 307–3077. DOI: 10.1109/FPL.2018.00059.

[31] Hardik Sharma et al. “Bit Fusion: Bit-level Dynamically Composable Architecture for Accelerating Deep Neural Network”. In: *ACM/IEEE ISCA*. 2018, pp. 764–775. DOI: 10.1109/ISCA.2018.00069.

[32] Reena Elangovan et al. “Ax-BxP: Approximate Blocked Computation for Precision-Reconfigurable Deep Neural Network Acceleration”. In: *ACM Transactions on Design Automation of Electronic Systems* 27.3 (2022), pp. 1–20. DOI: 10.1145/3492733.

[33] Ehab M. Ibrahim et al. “Taxonomy and Benchmarking of Precision-Scalable MAC Arrays under Enhanced DNN Dataflow Representation”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 69.5 (2022), pp. 2013–2024. DOI: 10.1109/TCSI.2022.3141519.

[34] Patrick Judd et al. “Stripes: Bit-serial Deep Neural Network Computing”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783722.

[35] Mengshu Sun et al. “FILM-QNN: Efficient FPGA Acceleration of Deep Neural Networks with Intra-Layer, Mixed-Precision Quantization”. In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2022, pp. 134–145. DOI: 10.1145/3490422.3502364.

[36] Chen Wu et al. “MP-OPU: A Mixed Precision FPGA-based Overlay Processor for Convolutional Neural Networks”. In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 2021, pp. 33–37. DOI: 10.1109/FPL53798.2021.00014.

[37] Vincent Camus et al. “Review and Benchmarking of Precision-Scalable Multiply-Accumulate Unit Architectures for Embedded Neural-Network Processing”. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.4 (2019), pp. 697–711. DOI: 10.1109/JETCAS.2019.2950386.

[38] Olexa Bilaniuk et al. “Bit-Slicing FPGA Accelerator for Quantized Neural Networks”. In: *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2019, pp. 1–5. DOI: 10.1109/ISCAS.2019.8702332.

[39] Jinmook Lee et al. “UNPU: An Energy-Efficient Deep Neural Network Accelerator with Fully Variable Weight Bit Precision”. In: *IEEE Journal of Solid-State Circuits* 54.1 (2018), pp. 173–185. DOI: 10.1109/JSSC.2018.2865489.

[40] Sayeh Sharify et al. “Loom: Exploiting Weight and Activation Precisions to Accelerate Convolutional Neural Networks”. In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6. DOI: 10.1145/3195970.3196072.

[41] Paolo Ienne and Marc A. Viredaz. “GENES IV: A Bit-Serial Processing Element for a Multi-Model Neural-Network Accelerator”. In: *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology* 9.3 (1995), pp. 257–273. DOI: 10.1007/BF02407088.

[42] Song Han et al. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. DOI: 10.48550/arXiv.1510.00149. arXiv: 1510.00149 [cs.CV].

[43] Angshuman Parashar et al. “SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 27–40. DOI: 10.1145/3140659.3080254.

[44] Shijin Zhang et al. “Cambricon-X: An Accelerator for Sparse Neural Networks”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783723.

[45] 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*. 2017, pp. 75–84. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021745.

[46] Jiajun Li et al. “SmartShuttle: Optimizing off-Chip Memory Accesses for Deep Learning Accelerators”. In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2018, pp. 343–348. DOI: 10.23919/DATE.2018.8342033.

[47] Yong Zheng et al. “Optimizing Off-Chip Memory Access for Deep Neural Network Accelerator”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 69.4 (2022), pp. 2316–2320. DOI: 10.1109/TCSII.2022.3150030.

[48] Adam Paszke et al. “Pytorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[49] Mohamed Abdelfattah et al. “DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration”. In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 411–4117. DOI: 10.1109/FPL.2018.00077.

[50] Ying Wang et al. “DeepBurning: Automatic Generation of FPGA-based Learning Accelerators for the Neural Network Family”. In: *Proceedings of the 53rd Annual Design Automation Conference*. 2016, 110:1–110:6. ISBN: 978-1-4503-4236-0. DOI: 10.1145/2897937.2898003.

[51] Xiaofan Zhang et al. “DNNBuilder: An Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs”. In: *Proceedings of the International Conference on Computer-Aided Design*. 2018, 56:1–56:8. ISBN: 978-1-4503-5950-4. DOI: 10.1145/3240765.3240801.

[52] Y. Guan et al. “FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates”. In: *2017 IEEE 25th*

Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Apr. 2017, pp. 152–159. DOI: 10.1109/FCCM.2017.25.

[53] 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*. 2015, pp. 161–170. ISBN: 978-1-4503-3315-3. DOI: 10.1145/2684746.2689060.

[54] K. Guo et al. "Angel-Eye: A Complete Design Flow for Mapping CNN onto Embedded FPGA". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.1 (Jan. 2018), pp. 35–47. ISSN: 1937-4151. DOI: 10.1109/TCAD.2017.2705069.

[55] Xuechao Wei et al. "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017. ISBN: 978-1-4503-4927-7. DOI: 10.1145/3061639.3062207.

[56] Siqi Wang et al. "High-Throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (2020), pp. 2254–2267. DOI: 10.1109/TCAD.2019.2944584.

[57] Christian Szegedy et al. "Going Deeper with Convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1–9.

[58] Paul Feautrier. “Some Efficient Solutions to the Affine Scheduling Problem. I. One-dimensional Time”. In: *International Journal of Parallel Programming* 21.5 (1992), pp. 313–347. DOI: 10.1007/BF01407835.

[59] Paul Feautrier. “Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time”. In: *International Journal of Parallel Programming* 21.6 (1992), pp. 389–420. DOI: 10.1007/BF01379404.

[60] Richard M. Stallman. “GNU Compiler Collection Internals”. In: *Free Software Foundation* 46 (2002).

[61] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[62] Weiwen Jiang et al. “Hardware/Software Co-Exploration of Neural Architectures”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.12 (2020), pp. 4805–4815. DOI: 10.1109/TCAD.2020.2986127.

[63] Mohamed S. Abdelfattah et al. “Best of Both Worlds: AutoML Codesign of a CNN and Its Hardware Accelerator”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218596.

[64] Shmuel Winograd. *Arithmetic Complexity of Computations*. Vol. 33. SIAM, 1980.

[65] Andrew Lavin and Scott Gray. “Fast Algorithms for Convolutional Neural Networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 4013–4021.

[66] Juan Yepez and Seok-Bum Ko. "Stride 2 1-D, 2-D, and 3-D Winograd for Convolutional Neural Networks". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.4 (2020), pp. 853–863. DOI: 10.1109/TVLSI.2019.2961602.

[67] Yun Liang et al. "Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.4 (2020), pp. 857–870. DOI: 10.1109/TCAD.2019.2897701.

[68] Utku Aydonat et al. "An OpenCL™ Deep Learning Accelerator on Arria 10". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 55–64. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021738.

[69] Adi Fuchs and David Wentzlaff. "The Accelerator Wall: Limits of Chip Specialization". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 1–14. DOI: 10.1109/HPCA.2019.00023.

[70] Xinheng Liu et al. "WinoCNN: Kernel Sharing Winograd Systolic Array for Efficient Convolutional Neural Network Acceleration on FPGAs". In: *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2021, pp. 258–265. DOI: 10.1109/ASAP52443.2021.00045.

[71] Vijaya Lakshmi et al. "A Novel In-Memory Wallace Tree Multiplier Architecture Using Majority Logic". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 69.3 (2022), pp. 1148–1158. DOI: 10.1109/TCSI.2021.3129827.

[72] K. Z. Pekmestzi. "Multiplexer-Based Array Multipliers". In: *IEEE Transactions on Computers* 48.1 (1999), pp. 15–23. DOI: 10.1109/12.8739.

[73] Oscar Gustafsson and Andreas Ehliar. "Low-Complexity General FIR Filters Based on Winograd's Inner Product Algorithm". In: *2013 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2013, pp. 93–96. DOI: 10.1109/ISCAS.2013.6571790.

[74] L. D. Elfimova and Yu V. Kapitonova. "A Fast Algorithm for Matrix Multiplication and Its Efficient Realization on Systolic Arrays". In: 37.1 (Jan. 2001), pp. 109–121. DOI: 10.1023/A:1016676318988.

[75] H.V. Jagadish and T. Kailath. "A Family of New Efficient Arrays for Matrix Multiplication". In: *IEEE Transactions on Computers* 38.1 (1989), pp. 149–155. DOI: 10.1109/12.8739.

[76] Ignacio Bravo et al. "Different Proposals to Matrix Multiplication Based on FPGAs". In: *2007 IEEE International Symposium on Industrial Electronics*. 2007, pp. 1709–1714. DOI: 10.1109/ISIE.2007.4374862.

[77] Riya Jain and Neeta Pandey. "Approximate Karatsuba Multiplier for Error-Resilient Applications". In: *AEU - International Journal of Electronics and Communications* 130 (2021), p. 153579. ISSN: 1434-8411. DOI: 10.1016/j.aeue.2020.153579.

[78] Riya Jain et al. "Booth-Encoded Karatsuba: A Novel Hardware-Efficient Multiplier". In: *Advances in Electrical and Electronic Engineering* 19.3 (2021), pp. 272–281. DOI: 10.15598/aeee.v19i3.4199.

[79] Volker Strassen. "Gaussian Elimination Is Not Optimal". In: *Numerische Mathematik* 13.4 (1969), pp. 354–356. DOI: 10.1007/BF02165411.

[80] Shmuel Winograd. "On Multiplication of 2× 2 Matrices". In: *Linear Algebra and its Applications* 4.4 (1971), pp. 381–388.

[81] Jianan Sun et al. "Accelerate Dense Matrix Multiplication on Heterogeneous-GPUs". In: *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. 2023, pp. 2726–2729. DOI: 10.1109/ICPADS60453.2023.00362.

[82] Chandan Misra et al. "Stark: Fast and Scalable Strassen's Matrix Multiplication Using Apache Spark". In: *IEEE Transactions on Big Data* 8.3 (2022), pp. 699–710. DOI: 10.1109/TBDATA.2020.2977326.

[83] Arjun Gopala Krishnan and Dhrubajyoti Goswami. "Multi-Stage Memory Efficient Strassen's Matrix Multiplication on GPU". In: *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2021, pp. 212–221. DOI: 10.1109/HiPC53243.2021.00035.

[84] Ahmed Khaled et al. "Applying Fast Matrix Multiplication to Neural Networks". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 2020, pp. 1034–1037. DOI: 10.1145/3341105.3373852.

[85] Jianyu Huang et al. "Strassen's Algorithm Reloaded on GPUs". In: *ACM Transactions on Mathematical Software (TOMS)* 46.1 (2020), pp. 1–22. DOI: 10.1145/3372419.

[86] Pai-Wei Lai et al. "Accelerating Strassen-Winograd's Matrix Multiplication Algorithm on GPUs". In: *2013 IEEE 20th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2013, pp. 139–148. DOI: 10.1109/HiPC.2013.6799109.

[87] Benjamin Lipshitz et al. "Communication-Avoiding Parallel Strassen: Implementation and Performance". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11. DOI: 10.1109/SC.2012.33.

[88] Grey Ballard et al. "Communication-Optimal Parallel Algorithm for Strassen's Matrix Multiplication". In: *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. 2012, pp. 193–204. DOI: 10.1145/2312005.2312044.

[89] Luis G. León-Vega et al. "Acceleration of Fully Connected Layers on FPGA Using the Strassen Matrix Multiplication". In: *2023 IEEE 5th International Conference on BioInspired Processing (BIP)*. 2023, pp. 1–6. DOI: 10.1109/BIP60195.2023.10379257.

[90] *Xillybus*. URL: http://xillybus.com/.

[91] Trevor E. Pogue and Nicola Nicolici. *FFIP Accelerator Implementation*. 2023. URL: https://github.com/trevorpogue/algebraic-nnhw.

[92] *Intel Arria 10 SoC Development Kit*. URL: https://www.intel.ca/content/www/ca/en/products/details/fpga/development-kits/arria/10-sx.html.

[93] *UltraScale Architecture DSP Slice*. 2021. URL: https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp.

[94] *Intel Arria 10 Native Fixed Point DSP IP Core User Guide*. 2017. URL: https://www.intel.com/content/www/us/en/docs/programmable/683583/current/intel-arria-native-fixed-point-dsp-ip.html.

[95] Shuanglong Liu et al. "Toward Full-Stack Acceleration of Deep Convolutional Neural Networks on FPGAs". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.8 (2022), pp. 3974–3987. DOI: 10.1109/TNNLS.2021.3055240.

[96] Hongxiang Fan et al. "FPGA-based Acceleration for Bayesian Convolutional Neural Networks". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.12 (2022), pp. 5343–5356. DOI: 10.1109/TCAD.2022.3160948.

[97] Jianjing An et al. "An OpenCL-based FPGA Accelerator for Faster R-CNN". In: *Entropy* 24.10 (2022), p. 1346.

[98] Yufei Ma et al. "Automatic Compilation of Diverse CNNs onto High-Performance FPGA Accelerators". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.2 (2020), pp. 424–437. DOI: 10.1109/TCAD.2018.2884972.

[99] Jianfei Jiang et al. "A CPU-FPGA Heterogeneous Acceleration System for Scene Text Detection Network". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 69.6 (2022), pp. 2947–2951. DOI: 10.1109/TCSII.2022.3167022.

[100] Suchang Kim et al. "A CNN Inference Accelerator on FPGA with Compression and Layer-Chaining Techniques for Style Transfer Applications". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 70.4 (2023), pp. 1591–1604. DOI: 10.1109/TCSI.2023.3234640.

[101] S. Kala et al. "High-Performance CNN Accelerator on FPGA Using Unified Winograd-GEMM Architecture". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.12 (2019), pp. 2816–2828. DOI: 10.1109/TVLSI.2019.2941250.

[102] Jixuan Li et al. "An FPGA-based Energy-Efficient Reconfigurable Convolutional Neural Network Accelerator for Object Recognition Applications". In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 68.9 (2021), pp. 3143–3147. DOI: 10.1109/TCSII.2021.3095283.

[103] Donghyuk Kim et al. "Agamotto: A Performance Optimization Framework for CNN Accelerator with Row Stationary Dataflow". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 70.6 (2023), pp. 2487–2496. DOI: 10.1109/TCSI.2023.3258411.

[104] Wenjin Huang et al. "FPGA-based High-Throughput CNN Hardware Accelerator with High Computing Resource Utilization Ratio". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.8 (2022), pp. 4069–4083. DOI: 10.1109/TNNLS.2021.3055814.

[105] Farshad Moradi et al. "Ultra Low Power Full Adder Topologies". In: *2009 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2009, pp. 3158–3161. DOI: 10.1109/ISCAS.2009.5118473.

[106] Natsumi Kawai et al. “A Fully Static Topologically-Compressed 21-Transistor Flip-Flop with 75% Power Saving”. In: *IEEE Journal of Solid-State Circuits* 49.11 (2014), pp. 2526–2533. DOI: 10.1109/JSSC.2014.2332532.

[107] Yunpeng Cai et al. “Ultra-Low Power 18-Transistor Fully Static Contention-Free Single-Phase Clocked Flip-Flop in 65-Nm CMOS”. In: *IEEE Journal of Solid-State Circuits* 54.2 (2019), pp. 550–559. DOI: 10.1109/JSSC.2018.2875089.

[108] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, 2010.

[109] Andrei L. Toom. “The Complexity of a Scheme of Functional Elements Realizing the Multiplication of Integers”. In: *Soviet Mathematics Doklady* 3 (1963), pp. 714–716. ISSN: 0197–6788.