RECONFIGURABLE CNN ACCELERATOR DESIGN USING DATAFLOW ANALYSIS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

By
Alperen Kalay
September 2024

RECONFIGURABLE CNN ACCELERATOR DESIGN USING DATAFLOW ANALYSIS
By Alperen Kalay
September 2024

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Uğur Güdükbay(Advisor)

Özcan Öztürk(Co-Advisor)

Hamdi Dibeklioğlu

Süleyman Tosun

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

RECONFIGURABLE CNN ACCELERATOR DESIGN USING DATAFLOW ANALYSIS

Alperen Kalay
M.S. in Computer Engineering
Advisor: Uğur Güdükbay
Co-Advisor: Özcan Öztürk
September 2024

Dataflow reconfigurability plays a crucial role in Convolutional Neural Network (CNN) acceleration by determining the optimal dataflow pattern for convolution operations. Fully reconfigurable architectures provide versatility and high resource utilization by supporting multiple dataflow options, but this comes with increased design complexity and operational overhead. On the other hand, non-reconfigurable architectures, optimized for a single dataflow pattern, deliver high efficiency for specific tasks but lack adaptability. This thesis introduces a novel intermediate dataflow reconfigurable CNN accelerator that balances flexibility and efficiency by integrating key dataflow patterns, enhancing adaptability and performance across diverse CNN applications. Through a detailed analysis, key dataflows are identified, and a unique architectural unit is developed for dataflow selection, with an average of 0.15% excess latency compared to the optimal scenario. Our specialized systolic array architecture accommodates various kernel sizes, providing an additional layer of reconfigurability. Our architecture requires 39% less area and 35% less power than fully reconfigurable designs. Additionally, it delivers an average of 33% better performance compared to non-reconfigurable architectures. In terms of efficiency, it provides a 7% increase over fully reconfigurable designs and outperforms non-reconfigurable options by up to 3.57X.

Keywords: CNN accelerator, dataflow, reconfigurability.

ÖZET

VERİ AKIŞI ANALİZİ KULLANARAK YENİDEN YAPILANDIRILABİLİR CNN HIZLANDIRICI TASARIMI

Alperen Kalay
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Uğur Güdükbay
İkinci Tez Danışmanı: Özcan Öztürk
Eylül 2024

Veri akışının yeniden yapılandırılabilirliği, evrişim işlemleri için en uygun veri akışı modelini belirleyerek Evrişimsel Sinir Ağı (CNN) hızlandırmasında önemli bir rol oynar. Tamamen yeniden yapılandırılabilir mimariler, birden fazla veri akışı seçeneğini destekleyerek çok yönlülük ve yüksek kaynak kullanımı sağlar, ancak bu durum artan tasarım karmaşıklığı ve işlem yükünü beraberinde getirir. Öte yandan, tek bir veri akışı modeli için optimize edilmiş, yeniden yapılandırılamayan mimariler, belirli görevler için yüksek verimlilik sunar ancak uyarlanabilirlikten yoksundur. Bu tez, esneklik ve verimliliği dengeleyen, çeşitli CNN uygulamalarında uyarlanabilirliği ve performansı artıran, anahtar veri akışı seçeneklerini birleştiren yenilikçi bir ara seviye yeniden yapılandırılabilir veri akışlı CNN hızlandırıcısı sunmaktadır. Ayrıntılı bir analizle, anahtar veri akışları belirlenmiş ve veri akışı seçiminde kullanılmak üzere özel bir mimari birim geliştirilmiştir; bu birim, optimum senaryoya kıyasla sadece ortalama %0.15 gecikme ile çalışmaktadır. Özel tasarlanmış sistolik dizi mimarimiz, çeşitli filtre boyutlarını destekleyerek yeniden yapılandırılabilirliğe ek bir katman eklemektedir. Mimarimiz, tamamen yeniden yapılandırılabilir tasarımlara kıyasla %39 daha az alan ve %35 daha az güç gerektirmektedir. Ayrıca, yeniden yapılandırılamayan mimarilere kıyasla ortalama %33 daha iyi performans sağlamaktadır. Verimlilik açısından, tamamen yeniden yapılandırılabilir tasarımlara göre %7'lik bir artış sağlamakta ve yeniden yapılandırılamayan seçenekleri 3.57 kata kadar daha yüksek verimlilikle geride bırakmaktadır.

Anahtar sözcükler: CNN hızlandırıcı, veri akışı, yeniden yapılandırılabilirlik.

Acknowledgement

I would like to thank my supervisors, Prof. Dr. Uğur Güdükbay and Prof. Dr. Özcan Öztürk, for their support. I am grateful to Prof. Dr. Özcan Öztürk for his guidance, valuable comments, and motivation.

I am also thankful for my jury members, Asst. Prof. Hamdi Dibeklioğlu and Prof. Dr. Süleyman Tosun for kindly accepting to evaluate my thesis.

Finally, most of my gratitude goes to my dearest family. Their endless love gave me strength throughout this journey. To them, I dedicate this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Convolutional Neural Networks (CNNs) are widely used in various applications such as image and video recognition, object detection, and other visual data processing tasks due to their ability to learn and extract features from raw data automatically. Their effectiveness in capturing spatial hierarchies and patterns makes them ideal for tasks requiring a high-level visual input understanding. The growing demand for real-time processing in these applications has led to significant interest in developing hardware accelerators specifically designed to optimize CNN performance, addressing the need for faster, more power-efficient implementations to handle modern neural networks' computational demands.

CNN accelerators are specialized hardware architectures that significantly enhance the performance and efficiency of CNNs, particularly in resource-constrained environments. These accelerators, often implemented on platforms such as FPGAs, ASICs, and GPUs, optimize CNN operations by leveraging different techniques, including dataflow optimization, quantization, and sparsity. Dataflow accelerators use customized dataflows to maximize data reuse and minimize memory access, achieving high energy efficiency and throughput [1]. Quantization-focused accelerators reduce computational complexity by converting weights and activations to lower-precision formats, thereby speeding up inference while maintaining accuracy [2]. Sparsity-based accelerators exploit the

inherent sparsity in neural networks by skipping zero-valued computations, significantly reducing both computational load and memory usage [3]. These categories of accelerators address the growing demand for high-performance, low-power solutions, enabling advanced CNN models to be deployed effectively in real-world applications.

Dataflow architectures in CNN accelerators are essential for optimizing performance and energy efficiency by defining how data is processed and moved through the system. Fully reconfigurable dataflow architectures offer flexibility by dynamically adapting to various dataflow patterns based on the specific needs of each layer [1, 4]. This adaptability allows these architectures to efficiently handle different dataflow options without hardware modifications, optimizing resource utilization and minimizing memory access. In contrast, non-reconfigurable dataflow architectures are fixed to a specific dataflow pattern [5, 6, 7]. These architectures, while simpler, are limited to the particular dataflow patterns they support, which can constrain their versatility across various CNN workloads.

The fully reconfigurable architecture's flexibility allows a single accelerator to support multiple dataflow options without hardware changes, enhancing versatility and utilization. However, the complexity of reconfigurable systems can lead to increased design and operational overhead, potentially impacting performance if not managed effectively. On the other hand, non-reconfigurable architectures, such as those optimized for a single dataflow pattern, provide simpler design and implementation with lower overhead. These systems can achieve high efficiency for specific tasks but lack the adaptability to handle diverse convolution operations, which may limit their overall utility. This thesis presents a new approach involving intermediate reconfigurability that combines some of the dataflow patterns. This architecture balances flexibility and efficiency by offering a compromise between fully reconfigurable and non-reconfigurable designs. Exploring the intermediate reconfigurability of dataflows could provide a solution that leverages the strengths of both approaches, offering improved adaptability and performance for a range of CNN applications.

Therefore, we propose an intermediate dataflow reconfigurable CNN accelerator, which is a systolic-array-based architecture to accelerate convolution layers in CNNs. Our key contributions are as follows:

- We comprehensively analyzed exploitable dataflow dimensions, identifying the most suitable CNN parameters for each dataflow option in processing these CNNs. This enabled us to search for the optimal combination of dataflows.
- We developed a systolic array architecture that supports intermediate reconfigurable dataflows. Our exploitable dimension analysis guided the selection of dataflows, and the systolic array's Processing Elements (PEs) are tailored to the chosen dataflow combinations.
- Cooperation of systolic arrays is done with Collector Units. These units are designed to work with various kernel sizes without losing efficiency. They provide another type of reconfigurability on top of dataflow.
- The Dataflow Selection Unit, integrated within the convolution engine, dynamically selects the optimal dataflow for convolution tasks. A selection algorithm is proposed to facilitate this on-the-fly dataflow selection.

In the rest of this thesis, we first discuss the background and the related work in Chapters 2 and 3, respectively. We then provide a detailed analysis of dataflow and reconfigurability in Chapter 4. Chapter 5 covers the proposed architecture's design and implementation. Our experimental setup is provided in Chapter 6, while Chapter 7 presents our experimental results. We finally conclude in Chapter 8.

Chapter 2

Background

2.1 Convolutional Neural Networks

CNNs are a class of deep learning models that have revolutionized the field of computer vision and have applications in various domains, such as image and video recognition, natural language processing, and autonomous driving. Unlike traditional neural networks, CNNs are specifically designed to process data with a grid-like topology, such as images, by exploiting spatial hierarchies. The foundational idea behind CNNs is the convolution operation, which allows the model to automatically and adaptively learn spatial hierarchies and extract features from raw input images.

CNNs have shown remarkable success in various benchmarks and real-world applications due to their ability to learn directly from data without requiring manual feature extraction. This capability has led to their widespread adoption and continuous development in both academic research and industry applications. A typical CNN architecture consists of several layers: convolutional, pooling, and fully connected. These layers work together to progressively transform the input data into features that can be used for classification or regression tasks.

- **Convolutional Layers:** These layers apply the convolution operation to the input data, producing feature maps. Each convolutional layer is defined by the number of filters it uses, the size of these filters, the stride, and the padding.
- **Pooling Layers:** These layers perform a down-sampling operation on the feature maps produced by the convolutional layers. Pooling helps reduce the spatial dimensions of the feature maps, thereby decreasing the computational load and aiding in extracting dominant features. Common types of pooling include max pooling and average pooling.
- **Fully Connected Layers:** These layers are similar to traditional neural network layers where each neuron is connected to the previous layer's neurons. They combine the features extracted by the convolutional and pooling layers to make final predictions.

The convolution operation is central to the functioning of CNNs. It involves sliding a filter (also known as a kernel) over the input data to produce a feature map. The filter is a small matrix of weights applied to a local region of the input data. Mathematically, the convolution operation can be expressed as:

$$(I * K)(i, j) = \sum_{m} \sum_{n} I(i + m, j + n) \times K(m, n), \tag{2.1}$$

where I is the input matrix (e.g., an image), K is the kernel, and (i,j) are the coordinates of the current position of the kernel on the input matrix. The sums are taken over the dimensions of the kernel.

Convolution can be performed on 1D, 2D, and 3D matrices. In 3D convolutions, the additional dimension is often called the channel, which allows the network to process color images with multiple channels (e.g., RGB) or volumetric data. Its illustration on a 2D matrix is given in Figure 2.1.

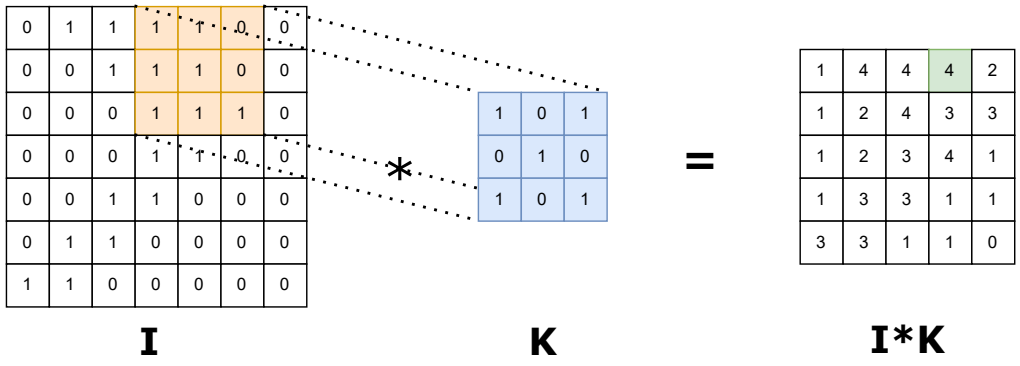

Figure 2.1: Convolution operation on 2D matrix.

The convolution operation in CNNs exhibits several fundamental properties that make it particularly effective for tasks involving spatial data. First, local connectivity ensures that each neuron in a convolutional layer is connected only to a local input region. This allows the network to learn spatial hierarchies and patterns, such as edges and textures in earlier layers and more complex structures in deeper layers. Second, parameter sharing uses the same set of weights (the kernel) across different spatial locations of the input, significantly reducing the number of parameters and enabling the network to learn translation-invariant features to recognize objects regardless of their position in the input image. Third, the stride and padding properties control the movement and spatial dimensions of the kernel over the input matrix; larger strides reduce the output dimensions and computational complexity, while padding adds extra pixels (usually zeros) around the border of the input to preserve spatial dimensions and allow the kernel to cover border regions. Finally, after the convolution operation, a non-linear activation function (such as ReLU [8], sigmoid [9], or tanh [10]) is typically applied to the feature map, enabling the network to learn complex, non-linear representations of the input data.

The convolution operation can be broken down into several steps:

- **Kernel Initialization:** The kernels in the convolutional layers are initially assigned random values. These kernels are small matrices (e.g., 3×3 or 5×5) and can be considered feature detectors.
- **Sliding the Kernel:** The kernel slides over the input matrix from left to right, top to bottom, applying the dot product between the kernel and local patches of the input matrix.
- **Generating the Feature Map:** The result of each dot product operation is a single value in the output feature map. This process is repeated for every kernel position on the input matrix, generating a complete feature map highlighting the presence of the features detected by the kernel.
- **Applying Non-linearity:** After generating the feature map, a non-linear activation function is applied to introduce non-linearities into the model, enabling it to learn more complex patterns.

In the context of a convolutional neural network, bias is an additional parameter used in the convolution operation. The purpose of the bias term is to allow the activation of the convolutional layer to be shifted by a particular value, which helps in better modeling the data and adds flexibility to the model. Formally, if K represents the kernel weights and b represents the bias, then the convolution operation can be expressed as:

$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n) \times K(m, n) + b. \tag{2.2}$$

The bias term b is added to every element of the resulting feature map after the convolution with the kernel K. It can help the model learn better by shifting the activation function.

Stride refers to the number of pixels by which the kernel moves (or "strides") across the input feature map during the convolution operation. The stride determines how much the kernel shifts at each step and thus affects the spatial

dimensions of the output feature map. The stride is denoted as s and can be represented as s_x for horizontal and s_y vertical movements, respectively. If the stride is s, the kernel's position on the input feature map moves by s pixels at each step.

For example, if the stride is 1, the kernel moves one pixel at a time, producing an output larger than a stride of 2, where the kernel moves two pixels at a time. The dimensions of the output feature map can be calculated as follows:

$$Output\,width = \left\lfloor \frac{Input\,width - Kernel\,width}{Stride} \right\rfloor + 1 \tag{2.3}$$

$$Output\,height = \left\lfloor \frac{Input\,height - Kernel\,height}{Stride} \right\rfloor + 1 \tag{2.4}$$

, where:

- $\lfloor . \rfloor$ denotes the floor operation, which rounds down to the nearest integer.
- Input width and height are the dimensions of the input feature map.
- Kernel width and height are the dimensions of the kernel.
- Stride is the step size for moving the kernel across the input feature map.

So, the bias allows for shifting the activation function, giving the model more flexibility. At the same time, the stride controls the movement of the kernel across the input, affecting the size of the output feature map.

The convolution operation generally enables CNNs to automatically and adaptively learn relevant features from the input data. The network typically learns low-level features such as edges, corners, and textures in the initial layers. As the depth of the network increases, the layers learn more abstract and high-level features, such as shapes, objects, and scenes. This hierarchical feature extraction is crucial for the success of CNNs in various computer vision tasks.

2.2 Toeplitz Matrices and Convolution

Toeplitz matrices and convolution are fundamental concepts in various fields, such as signal processing, systems theory, and applied mathematics. Their interplay is particularly significant in linear systems and digital signal processing, where they provide efficient computational frameworks for a wide range of applications.

A Toeplitz matrix, named after the German mathematician Otto Toeplitz [11], is a matrix in which each descending diagonal from left to right is constant. Formally, an $n \times n$ Toeplitz matrix T is defined as:

$$T = \begin{bmatrix} t_0 & t_{-1} & t_{-2} \dots & t_{n-1} \\ t_1 & t_0 & t_{-1} \dots & t_{n-2} \\ t_2 & t_1 & t_0 \dots & t_{n-3} \\ \vdots & \vdots & \ddots & \vdots \\ t_{n-1} & t_{n-2} & t_{n-3} \dots & t_0 \end{bmatrix} \tag{2.5}$$

Each element t_{ij} in the matrix is determined by $t_{ij} = t_{i-j}$. This structure leads to the unique property that its first row and first column entirely determine a Toeplitz matrix.

Toeplitz matrices are crucial in various computational problems due to their structured nature, allowing for efficient matrix-vector multiplication algorithms and linear system solving. This efficiency is paramount in large-scale applications like image processing and time series analysis.

The relationship between Toeplitz matrices and convolution arises because the convolution operation can be represented as a matrix-vector multiplication involving a Toeplitz-like matrix. Specifically, if we denote the convolution of a signal x with a kernel h as $y{=}x{*}h$, this operation can be written in matrix form as:

$$y = T(x)h, \tag{2.6}$$

where $T(x)$ is a Toeplitz matrix constructed from the sequence x. The matrix $T(x)$ encapsulates the convolution operation, with its structure ensuring that the convolution can be performed efficiently, especially for large-scale problems. An illustration is given in Figure 2.2.

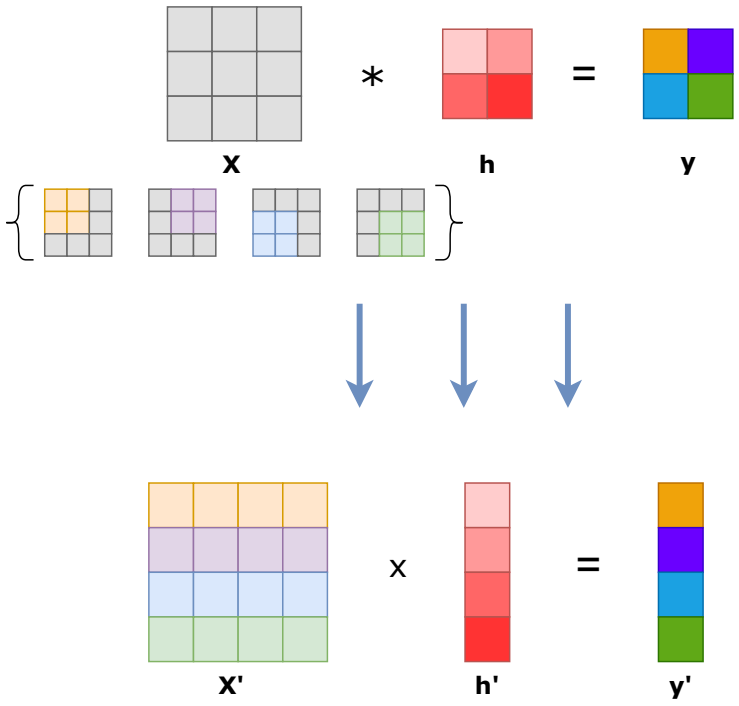

Figure 2.2: Convolution operation as matrix multiplication.

2.3 Systolic Arrays and Dataflows

2.3.1 Systolic Arrays

Systolic arrays are a class of parallel computing architectures designed for high-throughput data processing. They consist of PEs that rhythmically compute and pass data through the system. The term "systolic" derives from how data flows through the array, akin to blood pulsing through the body.

The key characteristics of systolic arrays include the following:

- **Regular and Repetitive Structure:** Systolic arrays comprise a grid of identical PEs, each performing simple operations. This regular structure allows for efficient hardware implementation.
- **Local Communication:** Each PE communicates only with its immediate neighbors, minimizing the need for complex and power-hungry global interconnects.
- **Data Pipelining:** Data flows in a pipelined fashion through the array, enabling high throughput and efficient utilization of hardware resources.

Systolic arrays are particularly well-suited for tasks that require repetitive and regular computations, such as matrix multiplication, convolution operations in neural networks, and digital signal processing.

2.3.2 Dataflows in Systolic Arrays

The efficiency of systolic arrays is heavily influenced by the dataflow patterns used to move data through the array. Different dataflow strategies can be employed to optimize performance, memory usage, and power consumption. The three primary dataflow patterns are Input Stationary (IS), Weight Stationary (WS), and Output Stationary (OS).

2.3.2.1 Input Stationary (IS):

In the IS dataflow, input data elements remain stationary within the PEs while weights and partial sums (psum) are moved through the array. This approach reduces the need to repeatedly access input data from memory, thereby saving memory bandwidth and energy. It is particularly effective when the input data is reused multiple times across different operations, as is common in CNNs. The illustration for IS is given in Figure 2.3.

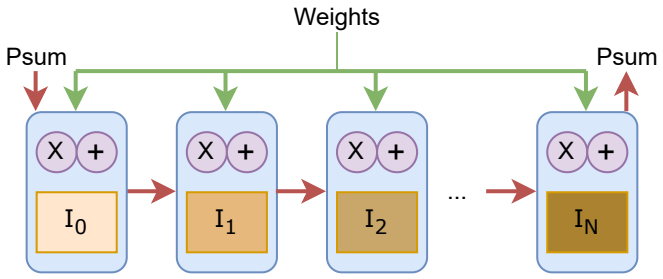

Figure 2.3: Dataflow of IS on PEs. Psum indicates partial sums, while I_i indicates the input.

- **Advantage:** Reduces memory access for input data, suitable when input datas are reused.
- **Disadvantage:** Requires efficient weight and partial sum movement management.

2.3.2.2 Weight Stationary (WS):

In the WS dataflow, weights remain fixed within the PEs while input data and partial sums flow through the array. This method minimizes the overhead of repeatedly fetching weights from memory, which can be beneficial when weights are reused across multiple input data elements. Figure 2.4 shows the detailed computational flow for WS.

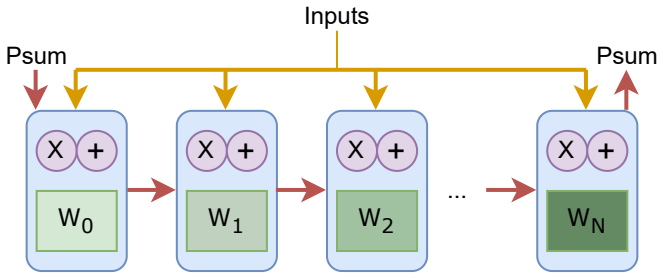

Figure 2.4: Dataflow of WS on PEs. Psum indicates partial sums, while W_i indicates the weight.

- **Advantage:** Reduces memory access for weights, advantageous when weights are reused.
- **Disadvantage:** May require more complex management of input data and partial sums.

2.3.2.3 Output Stationary (OS):

In the OS dataflow, partial sums (intermediate results) remain stationary within the PEs until the final result is computed. Input and weights flow through the array to update these partial sums. This approach is beneficial for applications where partial sums are frequently updated, as it minimizes the movement of intermediate results. The details of OS execution are given in Figure 2.5.

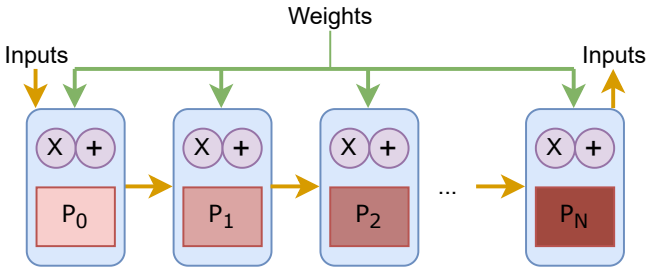

Figure 2.5: Dataflow of OS on PEs. P_i denotes partial sum.

- **Advantage:** Reduces memory access for partial sums, suitable for operations with frequent updates.
- **Disadvantage:** Requires efficient input data handling and weight movement.

Each dataflow pattern offers unique trade-offs regarding computational efficiency, memory bandwidth, and energy consumption. The choice of dataflow depends on the specific characteristics of the application and the target hardware architecture. Figure 2.6 shows different implementations of these dataflows on a systolic array. In IS and WS, either input or weight is stored in PEs, while in OS, accumulation is stored in PEs. Depending on dataflow, PEs store different data.

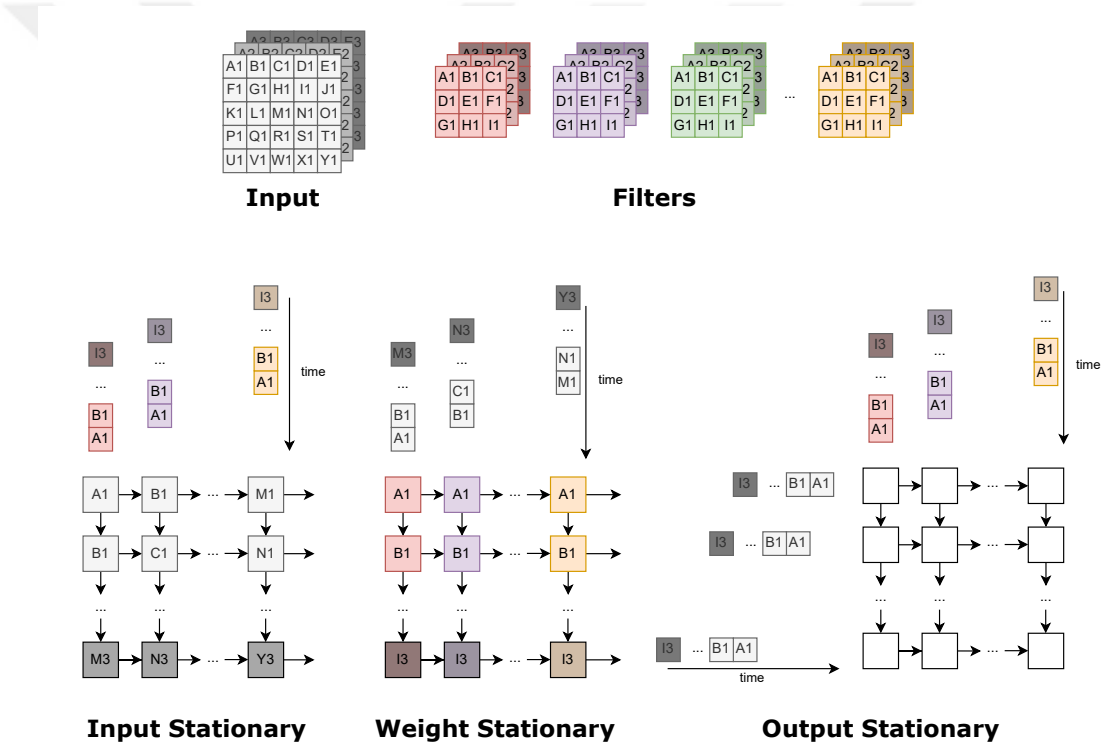

Figure 2.6: Different dataflows on a Systolic Array.

In IS, inputs are stored in PEs, and kernel weights are fed vertically to PEs. Multiplication is computed and transferred horizontally in each cycle. In WS, the structure is similar. The only difference is that weights are stored in PEs, and inputs are fed to PEs. In OS, accumulation is not transferred. It is stored in PEs. Inputs are fed horizontally, and weights are fed vertically.

2.3.3 Applications of Systolic Arrays and Dataflows

Systolic arrays and the associated dataflow patterns are widely used in high-performance computing applications. In particular, they are integral to accelerating machine learning and artificial intelligence workloads, enabling efficient implementation of deep learning models. Based on the characteristic discussed earlier in Sections 2.1 and 2.2, convolution is a very suitable operation for systolic arrays due to its repetitive and regular computational patterns, which align well with the structure and dataflow capabilities of systolic architectures.

Chapter 3

Related Work

The increasing demand for faster and more energy-efficient implementations of deep learning applications has sparked significant research and development into various methods to tackle this challenge. This has driven researchers and engineers to innovate and optimize hardware designs to meet modern deep learning tasks' performance and energy efficiency requirements. This section highlights some notable contributions in this field. Initially, some pioneering works are discussed. Subsequently, accelerators that prioritize different dataflow options are explored. A discussion on reconfigurable architectures follows this. Next, architectures focusing on quantization and sparsity are examined. Then, approaches utilizing specialized memories are mentioned. Finally, a discussion regarding our contribution is made.

In 1989, Treleaven and colleagues started the efforts on custom VLSI architectures for neural networks [12]. Later in 1992, Asanovic and his team created a microprocessor for artificial neural networks called SPERT [13], and they improved it in later years [14, 15]. More recently, systolic array-based accelerators have replaced coprocessors and specialized processors. This trend grew after Chen T. and his team developed the DianNao architecture family [5, 6, 16, 17].

Since then, new methods have focused on essential concepts such as dataflow-reconfigurability, quantization, and sparsity. Given the diverse nature of accelerator designs, Table 3.1 divides these efforts into categories in two dimensions. The first dimension is about optimization used in the design. The optimization approaches explored in the literature include dataflow, quantization, and sparsity. Since quantization and sparsity are highly related to the model's training, the focus turns to other strategies, like quantization-aware training for quantization and activation function selection for sparsity. Therefore, architecture becomes more model-specific. On the other hand, dataflow optimizations are closer to hardware design, leading to more development in this area. The second dimension is about the reconfigurability of the implementation. Our approach falls into the Reconfigurable-Dataflow category in the top left corner.

Optimization\Dynamicality	**Reconfigurable**	**Non-Reconfigurable**
Dataflow	FlexFlow [4], NeuFlow [18] Origami [19], Eyeriss v2 [20], MAERI [21], AdaptFlow [22]	ShiDianNao [6], nn-X [23], SCNN [24], INCA [25], UCNN [26]
Quantization	BitFusion [27], Stripes [28], UNPU [29], Loom [30], DRQ [31]	UCNN [26], YodaNN [32], OLAccel [33]
Sparsity	Eyeriss [1], EIE [3], Cambricon-X [34]	SCNN [24],Cnvlutin [35]

Table 3.1: Categorization of related efforts. They are categorized in two dimensions: optimization and reconfigurability.

Reconfigurability represents an innovative approach to empowering hardware systems to adapt to various data reuse patterns as the need arises. These architectures manage some design choices on-the-fly. Non-reconfigurable architectures refer to pre-defined and implemented optimizations. Most literature uses three optimization techniques: dataflow, quantization, and sparsity. Reconfigurable dataflow works focus on using different dataflows depending on convolution parameters, while non-reconfigurable cases try to select the best dataflow for their architecture. In quantization, reconfigurable works aimed to work with varying bit widths in the same architecture. On the other hand, non-reconfigurable ones define the best possible bitwidth to quantize. Similarly, reconfigurable sparsity focuses on on-the-fly compression of sparse data, while non-reconfigurable case adapts a compression algorithm through the architecture.

Regarding dataflow, each architectural design pursues the optimization of data reuse in convolution operations, aiming to reduce the need for memory transfers. This optimization is achieved through three distinct forms of reuse: input feature map reuse, convolutional reuse, and filter reuse, each corresponding to specific approaches. These primary approaches include IS, WS, and OS dataflows. In IS dataflow, input feature maps are stored locally within PEs, while weights are shared across PEs. For example, SCNN [24] focuses on accelerating convolution with sparse inputs, showing speedups above 17% sparsity. INCA [25], another IS dataflow accelerator, focuses on the processing in memory (PIM) approach to reach inputs faster. WS dataflow involves storing filter weights in PEs while input values are distributed across PEs. nn-X [23] operates as a low-power co-processor, achieving 227 Giga Operations Per Second (GOPS) with less than 4W power consumption. In OS dataflow, PEs store partial sums instead of weight data, eliminating the need for an accumulation tree, as seen in WS dataflows. Accelerators adhering to this paradigm can effectively leverage convolutional reuse. ShiDianNao [6] directly acquires input data from sensors, thus avoiding DRAM reads. It employs a similar Neural Functional Unit (NFU) architecture as in DianNao, enhancing performance through inter-PE communication and improving energy efficiency by up to 60x. UCNN [26] introduces factorization and memoization techniques to expedite the computation of repetitive weight results, thereby addressing sparsity issues. Gupta et al. [7] also adopt OS dataflow, focusing on energy-efficient design and utilizing low-precision fixed-point arithmetic with stochastic rounding. Considering all dataflow techniques, each has its advantages and disadvantages. Different layers within neural networks may be more suited to specific dataflow approaches. Research conducted by Hyonkjun Kwon and Xianhong Hu explores the behavior of various dataflows, shedding light on their relative performance characteristics [36, 37].

In addition to the mentioned dataflows, Chen Yu-Hsin and his colleagues have developed a novel reconfigurable accelerator with a unique row-stationary dataflow, as seen in Eyeriss [1] and Eyeriss v2 [20]. In this approach, input data is distributed diagonally, filter weights are distributed horizontally, and output values are collected diagonally using 1D convolutions. It stands out in terms

of efficiency, aiming to minimize idle PEs across various convolution operations. One of the other noteworthy examples in this regard is FlexFlow [4]. FlexFlow introduces a complementary parallelism technique and determines loop parameters through a utilization function. This function tailors the data to the specific characteristics of a convolutional layer and utilizes a 2D mesh topology. On the other hand, MAERI [21] employs a specialized tree topology known as the augmented reduction tree. This approach mitigates the issues of excessive area occupation and power inefficiency often associated with crossbar or mesh topologies. The flexible interconnect in MAERI enables it to support various layer types such as LSTM, pooling, fully connected, and convolution. Another example, AdaptFlow [22], features a flexible interconnect, a dataflow selection algorithm, and a dataflow mapping technique, collectively improving performance and energy efficiency. NeuFlow [18] is optimized for computer vision tasks like optical flow, showcasing real-time performance and efficiency. Origami [19] exemplifies WS architecture focusing on scalability and adaptability to larger and smaller architectures. However, considering all reconfigurable architectures, it is essential to note that the downside of reconfigurability is the overhead incurred when reconfiguring the system for each layer operation. While beneficial for adapting to different reuse patterns, this additional step introduces a computational cost that must be carefully managed.

Quantization and variable bit-width computation are essential concepts in neural network design. Variable bit-width computation allows for different precision levels in neural network layers, enhancing accuracy in critical layers. Bitfusion [27] uses decomposed multiplication, supporting 8, 4, and 2-bit precisions. Stripes [28] employs bit-serial multiplication, allowing real-time tradeoffs between latency and accuracy. UNPU [29] and Loom [30] also use spatio-temporal and fully temporal designs, respectively, for variable bit-width computation. However, in such approaches, high precision can increase latency. YodaNN [32] focuses on binary neural network acceleration, eliminating the need for multiplication units and benefiting from channel tiling. DRQ [31] and OLAccel [33] propose techniques for handling quantization sensitivity. DRQ adjusts precision dynamically for sensitive regions, while OLAccel includes specialized PEs for outlier values.

Sparse accelerators are essential since some activation functions, like ReLU in neural networks, create sparse activations. This sparseness increases as activations get closer to the last layers. Besides activations, sparse weights can also be present. Pruning can be used to create these sparse networks in deep learning applications. To exploit sparseness, several designs are proposed from a hardware point of view. Generally, sparse designs aim to compress data using various compression methods like Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Compressed Image Size (CIS), and Run Length Coding (RLC). Cnvlutin [35] eliminates computations with zero activations, while Cambricon-X [34] skips multiplications with zero weights. SCNN [24] uses a tiled approach with compressed sparse formats for both activations and weights, and EIE [3] broadcasts non-zero values to all PEs for multiplication.

Other accelerators try to look at the problem from different aspects. TETRIS [38] and PRIME [39] aims to reduce memory transfer overhead by performing computations near or within memory. TETRIS uses a 3D Hybrid Memory Cube (HMC) for in-memory processing and PRIME employs the PIM technique with Resistive RAM (ReRAM) for computation and storage. SC-DCNN [40] uses stochastic computing for convolution operations, mapping data range [-1,1] and utilizing basic logic operations for multiplication and addition, offering power and area savings at some accuracy cost. ESCALATE [41] uses kernel decomposition to filter weights, reducing computation in convolution and enabling more prunable computation schemes.

In addition to all these efforts, we propose a custom accelerator in reconfigurable dataflow category. Previously proposed approaches in this category focus on either applying one dataflow or making a reconfigurable accelerator to apply these dataflows. However, intermediate reconfigurability is overlooked in such approaches. Therefore, we concentrated on intermediate reconfigurability and analyzed architecture overheads for different configurations. Our main contribution is to exploit reconfigurability in dataflow and propose a custom hardware with a detailed exploitable dimension analysis. We implemented a dataflow selection unit in the convolution engine. Furthermore, we proposed and verified that combining two dataflows is enough to provide efficiency and performance.

Chapter 4

Dataflow and Reconfigurability Analysis

4.1 Dataflow Analysis

CNN accelerators generally exploit data reuse in one of the dataflow forms: IS, WS, or OS. As mentioned in the literature review, most previous designs adopt OS and WS dataflow since IS performs poorly in the proposed hardware topologies. However, our analysis show that a custom hardware accelerator can benefit from IS if designed accordingly.

Algorithmically, a convolution operation consists of 7 nested loops for 3D convolution, where each loop is responsible for a dimension. The number of loops changes for 1D or 2D cases, but computer vision tasks generally deal with 3D images where the third dimension is the channel. Table 4.1 gives possible parameters used in different dataflows. These loops can be reordered, and each dataflow offers a different ordering. Since outer loops are reached at long intervals, they are considered stationary. As also mentioned in the Background section, dataflows have various stationary parts, and these stationary parts correspond to outer loops in that specific ordering. An accelerator tries to parallelize the inner loops

and reuse the outer loops as much as possible. Selecting an outer loop is not ideal, as there can be a convolution operation with various parameters. Algorithms 1, 2, and 3 show loop orderings for IS, WS, and OS dataflows, respectively. Some layers are suitable for exploitation with parallel computing. Red boxes indicate a stationary section, and blue boxes represent a parallelizable section.

Batch Number (BN)	Number of inputs for one inference
Input Channel (IC)	Depth of input matrix
Input Height (IH)	Height of input matrix
Input Weight (IW)	Width of input matrix
Kernel Number (KN)	Number of kernels to be applied
Kernel Channel (KC)	Depth of kernel matrix (must be same with IC)
Kernel Height (KH)	Height of kernel matrix
Kernel Width (KW)	Width of kernel matrix
Output Channel (OC)	Depth of output matrix (must be same with KN)
Output Height (OH)	Height of output matrix
Output Width (OW)	Width of output matrix

Table 4.1: Possible parameters used in different dataflows in convolution.

IS

```
for bn ∈ batch do
    for ic ∈ input channel do
        for ih ∈ input height do
            for iw ∈ input width do
                for kn ∈ kernel number do
                    for kh ∈ kernel height do
                        for kw ∈ kernel width do
                            output[bn][kn][(ih-kh)//stride][(iw-kw)//stride]+=weights[kn][ic][kh][kw]*inp[bn][ic][ih][iw]
```

Algorithm 1: Loop ordering of convolution operation for IS dataflow.

WS

```
for bn ∈ batch do
    for kn ∈ kernel number do
        for kc ∈ kernel channel do
            for kh ∈ kernel height do
                for kw ∈ kernel width do
                    for oh ∈ output height do
                        for ow ∈ output width do
                            output[bn][kn][oh][ow]+=weights[kn][kc][kh][kw]*inp[bn][kc][stride*oh+kh][stride*ow+kw]
```

Algorithm 2: Loop ordering of convolution operation for WS dataflow.

OS

```
for bn ∈ batch do
    for oc ∈ output channel do
        for oh ∈ output height do
            for ow ∈ output width do
                for kc ∈ kernel channel do
                    for kh ∈ kernel height do
                        for kw ∈ kernel width do
                            output[bn][oc][oh][ow]+=weights[oc][kc][kh][kw]*inp[bn][kc][stride*oh+kh][stride*ow+kw]
```

Algorithm 3: Loop ordering of convolution operation for OS dataflow.

Selecting which dataflow to use is one of the main concerns in accelerator design. From this algorithmic perspective, it can be seen that OS can be used to parallelize computations where some kernel dimensions change. Similarly, WS may be suitable for large iterations related to output dimensions and can reduce computing time over kernel dimensions. In other words, each dataflow exploits convolutional dimensions in their corresponding inner loops. Exploitable dimensions of dataflows are represented in Table 4.2.

Dataflow	**Reconfigurability**
IS	Output Channel, Kernel Height, Kernel Width
WS	Output Height, Output Width
OS	Kernel Channel, Kernel Height, Kernel Width

Table 4.2: Exploitable dimensions for different dataflow techniques. Note that they are inner loop dimensions in the corresponding algorithms.

To clarify Table 4.2, it is best to use;

- OS if input with a large channel size is convolved with large filters because the parallelizable part in the OS algorithm involves dimensions related to kernels.
- WS if the resulting output height and width are large because the parallelizable part in the WS algorithm involves dimensions of output height and width.
- IS if several filters are to be applied and filter sizes are large because the parallelizable part in IS dataflow involves dimensions of kernel height, kernel width, and number of kernels.

4.2 Reconfigurability Analysis

A designer can merge different dataflows to not stick with one dataflow only. Reconfiguration can be implemented to decide the dataflow on the fly. In our case, the accelerator takes convolution parameters as inputs, and a deciding unit selects one option best suited to the given parameters. Then, the convolution operation is performed with this selected dataflow since different convolutions demand different dataflows to be applied.

Reconfigurable architectures come with the advantage of flexibility, where different configurations can be used on the same hardware. In our hardware architecture, different dataflows can be applied for different convolutions in the same accelerator. However, the overhead of area and power is inevitable. Because hardware should be capable of working with all dataflows, the accelerator must include all the necessary units, even if they are not used. Some parts of the hardware will be idle for a convolution operation. For example, when working with large input sizes, the accelerator selects WS dataflow as a suitable approach for the required computation. Still, some specific units will be idle during that computation, which is only related to IS. So, the overhead of combining similar dataflows will be lower than combining very different structures.

4.2.1 Toeplitz Matrices and Similarity of WS ⊎ IS

Based on precvious discussion in the Background section, convolution can be represented as matrix multiplication of Toeplitz matrices. As shown in Figure 2.6, WS and IS share similar structures: either inputs or weights are stored in PEs, while the other is obtained from a matrix. Swapping the roles of inputs and weights makes it easy to switch between WS and IS. The critical difference between WS and IS lies in handling inputs and weights, allowing easy combination by choosing which matrix to store or feed. However, this approach does not apply to OS, which requires both inputs and weights simultaneously for computation.

Partial sums in WS and IS move along the systolic array, updating the result without additional memory. In contrast, OS generates results in PEs, requiring an extra accumulator rather than a simple register, unlike WS and IS, where stationary data is kept in registers without accumulation. Thus, OS necessitates an additional unit for all PEs. While this is a trade-off, it highlights that WS and IS are similar unlike OS.

4.2.2 Combining Exploitable Dimensions

In Section 4.1, Table 4.2 shows exploitable dimensions. If we examine them, it can be seen that OS and IS have same two dimensions to exploit. Since output height and width dimensions are only exploitable in WS, it would be wise to use WS. On the other hand, combining both OS and IS may not be worthwhile since two of the dimensions are same. Besides, OS is not similar to WS or IS in terms of their similarity, as discussed in the previous section. So, adding extra accumulators to every PE in return for exploiting only the kernel channel dimension may not be a good design choice. By leaving OS out of our dataflow options, we can exploit 6/7 of the dimensions, requiring only minor adjustments to combine WS and IS. Based on these observations, if we want to design a reconfigurable architecture, combining WS and IS may not be as costly as combining WS or IS with OS.

Chapter 5

Accelerator Architecture

The high-level architecture of our accelerator is illustrated in Figure 5.1. This architecture comprises a convolution engine, memory blocks, buffers, Collector Units, and PE arrays as core components. Initially, we will discuss the hardware's workflow and the finite state machine (FSM). Subsequently, the architectural components will be examined, focusing on their reconfigurability.

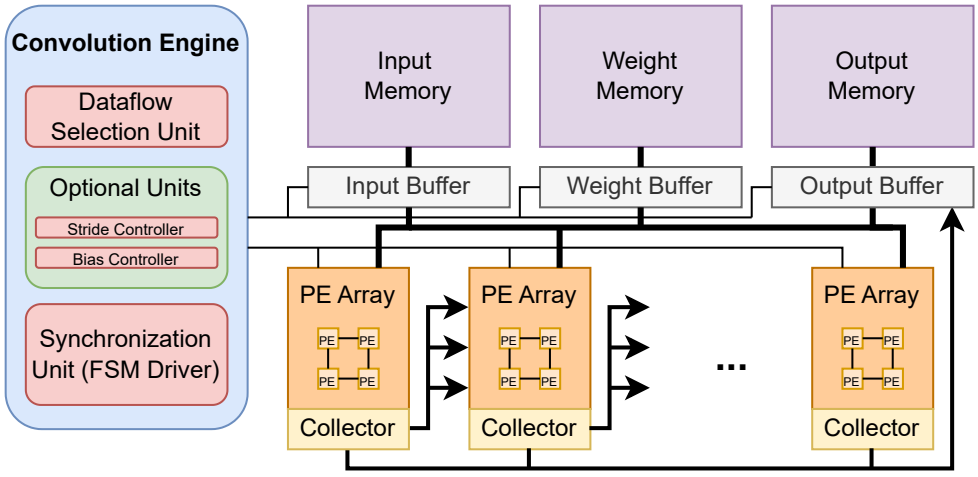

Figure 5.1: High-level architecture of the accelerator.

As discussed in Sections 2 and 4, reconfigurability comes with overheads due to the extra hardware needed for executing different dataflows. Moreover, it is analyzed that combining WS and IS might be more efficient in hardware compared to any other combination involving OS. So, our proposed architecture has reconfigurability of WS⊎IS. The similarity between these dataflows is a critical aspect of the architectural design. However, other reconfigurable architectures will also be implemented and evaluated for comparison, highlighting the additional requirements necessary for integration. In addition to that, non-reconfigurable (only IS, WS, or OS) options will also be evaluated.

The accelerator primarily takes convolution operation parameters as input, computes the results, and stores them in the output memory. Inputs and weights are stored in SRAMs using a Toeplitz matrix format, while the output SRAM collects results in the standard format, not using the Toeplitz structure. The Dataflow Selection Unit inside the convolution engine determines which dataflow to apply based on the given convolution parameters. It is the place where dataflow reconfigurability is handled. Stride and bias controllers are used in convolution operations for different stride and bias options. The synchronization unit manages communication between arrays and generates signals indicating where to retrieve the results. The PE arrays consist of distinct systolic arrays connected by small Collector Units that either collect results or pass the current accumulation to the following PE array. Each PE array contains a mesh topology of $k \times 9$ processing elements, where k stands for the number of kernels processed simultaneously. A detailed description of these units will be provided in the subsequent sections.

As discussed in previous sections, one of the critical concerns is selecting the appropriate dataflow. A reconfigurable architecture has been designed to compute using WS and IS dataflows. Other combinations of these dataflows have been developed and will be compared in the experimental results section. It is crucial to co-design the architecture and dataflow to achieve optimal performance.

5.1 Workflow and FSM structure

Our accelerator operates using a finite state machine (FSM) model, as illustrated in Figure 5.2. The FSM structure comprises three states: idle, load, and execute.

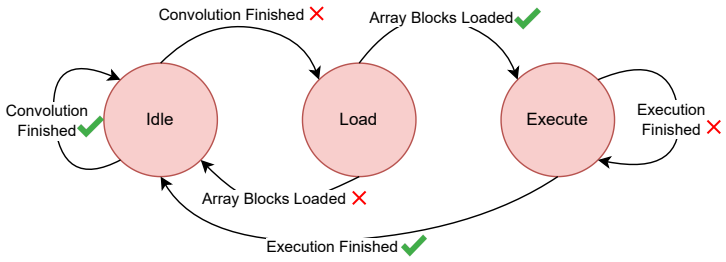

Figure 5.2: FSM of the architecture.

The idle state is used to synchronize PE arrays during the loading state. In this state, a buffer fetches a portion of data from either the inputs or weights, depending on the selected dataflow. Subsequently, one PE array is provided with the input data during each cycle. The FSM alternates between the load and idle states throughout this process. The idle state is also employed after the computation has finished.

The load state loads stationary data into the PE registers. It begins by checking the dataflow selection and determines how many PE arrays will be loaded. Depending on the active PE arrays, stationary data is loaded sequentially using the weight or input buffer during each cycle. For the WS dataflow, a portion of the channels is handled during each execution. The cursor for the channel portion is updated after every load state. If the last channel portion is reached, the kernel cursor is updated, and the procedure continues until all kernels are processed. For the IS dataflow, a portion of the spatial locations in the input are handled. The cursor for the spatial location portion is updated after every load state. If the last spatial location portion is reached, the channel cursor is updated, and the procedure continues until all channels are traversed. The pseudo-code for the

loading procedure for both WS and IS dataflows is provided in Algorithms 4 and 5, along with their corresponding traversals on input and weight matrices, which are shown in Figures 5.3 and 5.4.

Load State - WS

repeat
  if *channel cursor* $\neq$ last channel **then**
    Load Stationary Data
    Update(*channel cursor*)
    State $\leftarrow$ Execute
  else if *kernel cursor* $\neq$ last_kernel **then**
    Load Stationary Data
    Update(*kernel cursor*)
    Update(*channel cursor*)
    State $\leftarrow$ Execute
  else
    Finished
  end if
until Convolution Finished

Algorithm 4: Load state of WS dataflow.

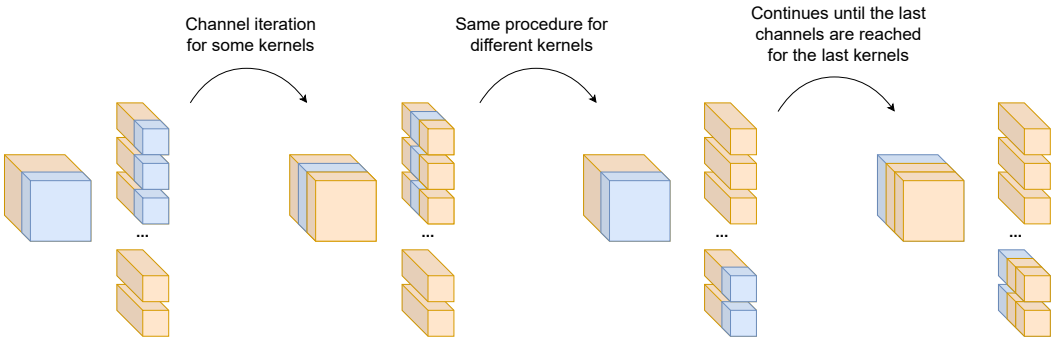

Figure 5.3: Load state of WS on matrices. Blue parts represent the input and kernel parts currently used for convolution.

Considering Algorithm 4 and Figure 5.3, channels are updated for both input and a group of kernels as the first step. After reaching the last channel, different kernels are selected. The channel updating procedure holds for these new kernels as well. The convolution ends when all the channels are reached for the last group of kernels.

Load State - IS

repeat
  if *spatial cursor* $\neq$ last location **then**
    Load Stationary Data
    Update(*spatial cursor*)
    State $\leftarrow$ Execute
  else if *channel cursor* $\neq$ last channel **then**
    Load Stationary Data
    Update(*channel cursor*)
    Update(*spatial cursor*)
    State $\leftarrow$ Execute
  else
    Finished
  end if
until Convolution Finished

Algorithm 5: Load state of IS dataflow.

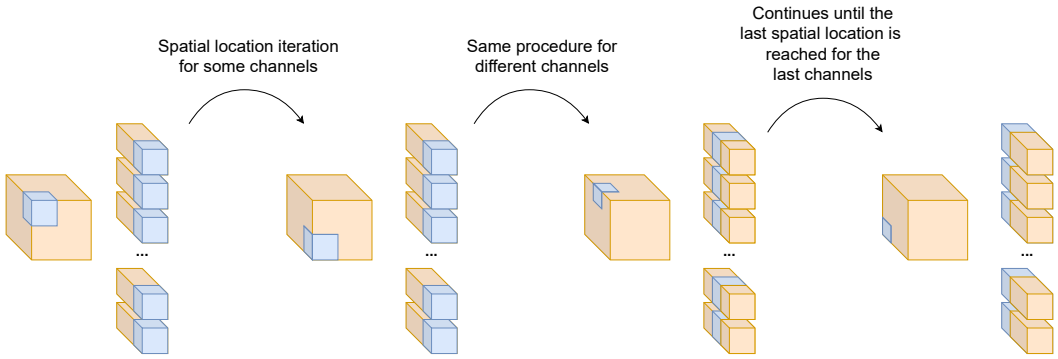

Figure 5.4: Load state of IS on matrices. Blue parts represent the input and kernel parts currently used for convolution.

Considering Algorithm 5 and Figure 5.4, spatial location in input is updated as the first step. After reaching the last spatial location, different channels are selected. All kernels are handled together, so there is no need to choose a distinct group of kernels. The spatial location updating procedure holds for these new channels as well. The convolution ends when the last spatial location in input is reached for all the channels of input and kernels.

The execute state is performed after the arrays are loaded with stationary data, as illustrated in Figure 5.5. Colorful boxes represent PEs; the data above are non-stationary data provided at each cycle. Each PE in the PE arrays carries out multiplication and accumulating operations. During each cycle, new non-stationary data is fetched, multiplied by the stationary data at the PE, and accumulated with the result from the previous PE. For example, at cycle 3, the last sum is collected with the new multiplication of $k3 \times i3$. This execution is performed on a portion of the stationary data. Once all the stationary data has been processed with the help of the loading state, as discussed above, the convolution operation is complete.

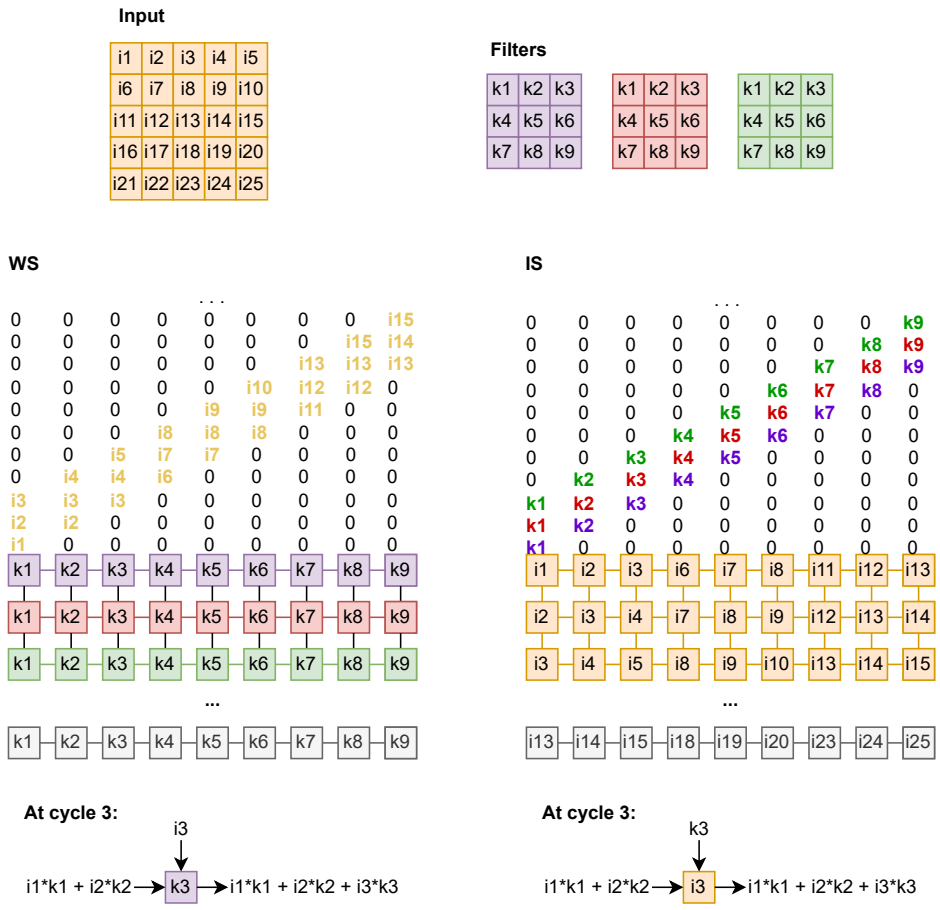

Figure 5.5: The execution details of convolution in WS and IS.

5.2 Incorporating OS in the Accelerator

We have discussed the general workflow from the perspectives of WS and IS dataflows due to the reconfigurability analysis in Section 4.2. Next, we examine the necessary changes for implementing an OS-capable accelerator. Figure 5.6 shows the FSM of the OS-capable architecture. As can be seen, it requires an additional store state because the result is accumulated within the PE. The results are collected and stored in the output memory during the store state.

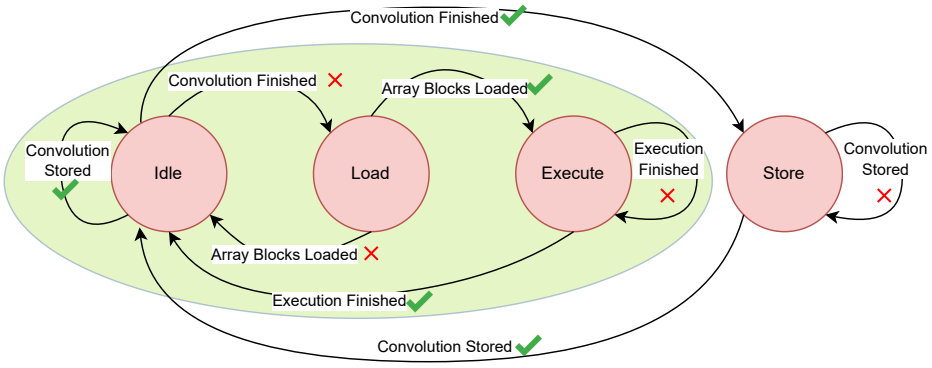

Figure 5.6: FSM of the OS-capable architecture. Note that green part is original FSM.

Algorithm 6 provides the respective pseudo-code for the load state. It traverses along the input spatial locations. When the last portion of the spatial location is reached, the kernel cursor is updated. Figure 5.7 shows its corresponding traversal on input and weight matrices.

Load State - OS

repeat
  if *spatial cursor* $\neq$ last location **then**
    Update(*spatial cursor*)
    State $\leftarrow$ Execute
  else if *kernel cursor* $\neq$ last kernel **then**
    Update(*kernel cursor*)
    Update(*spatial cursor*)
    State $\leftarrow$ Execute
  else
    Finished
  end if
until Convolution Finished

Algorithm 6: Load state of OS dataflow.

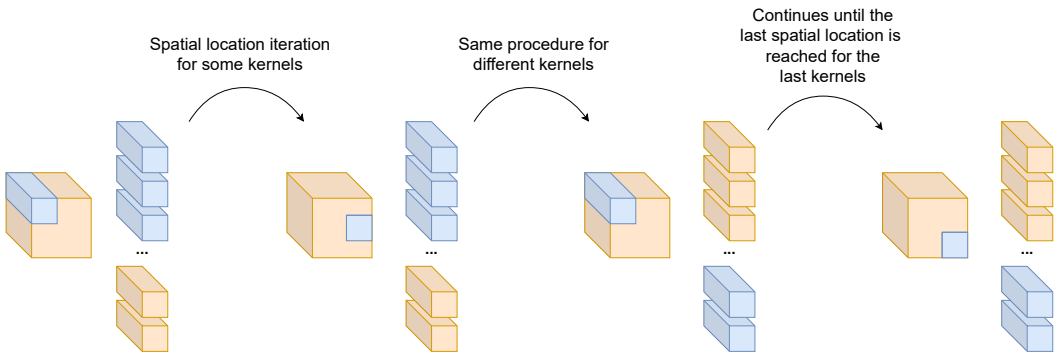

Figure 5.7: Load state of OS on matrices. Blue parts represent the input and kernel parts currently used for convolution.

Considering Algorithm 6 and Figure 5.7, spatial location in input is updated as the first step, using all the channels in that spatial location for a portion of the kernels. After reaching the last spatial location, different kernels are selected. The spatial location updating procedure holds for these new kernels as well. The convolution ends when all the spatial location in the input is reached for the last kernel portion.

During the execution state, Toeplitz inputs are received from both directions, as illustrated in Figure 5.8. The MAC operation is performed within each PE without transferring the operation to other PEs. At cycle 3, the accumulator is updated by adding $k3 \times i3$ to the previous accumulation.

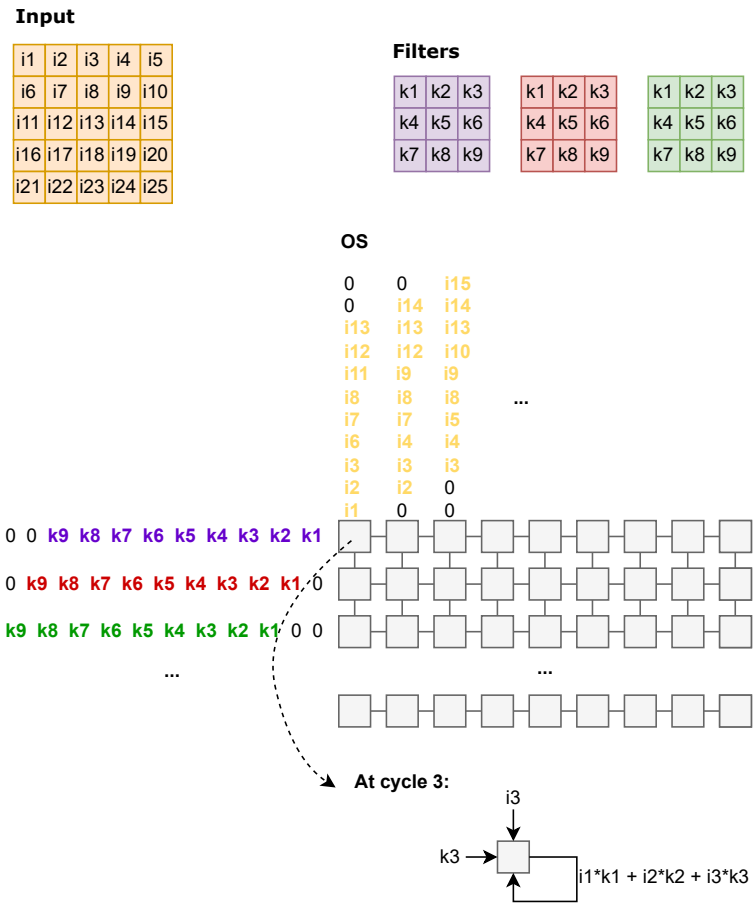

Figure 5.8: The execution steps of convolution in OS.

Transitioning to an OS paradigm for our accelerator involves significant changes. Unlike WS and IS modes, where data synchronization and multiply-accumulate (MAC) operations are tailored similarly, OS requires accumulating results within each PE and storing them directly in output memory. This shift necessitates modifying the FSM to include a dedicated store state for result collection.

5.3 PE Arrays and Collector Units

The centerpiece of our architecture is PE arrays, which are responsible for computation. There are a total of 18 PE arrays in the architecture, each containing its own $k \times 9$ systolic arrays. Collector Units are special units that are responsible for PE array connections. PE arrays and Collector Units are discussed together since they are highly coupled and collaborate. Figure 5.9 shows a PE inside a PE array and a Collector Unit. A 2D systolic array resides in the PE array. PEs at the last column of the systolic array are connected to the Collector Unit. Depending on the signal from the synchronization unit in the convolution engine, the Collector Unit decides to continue the computation in the following PE array or collect the results from this PE array.

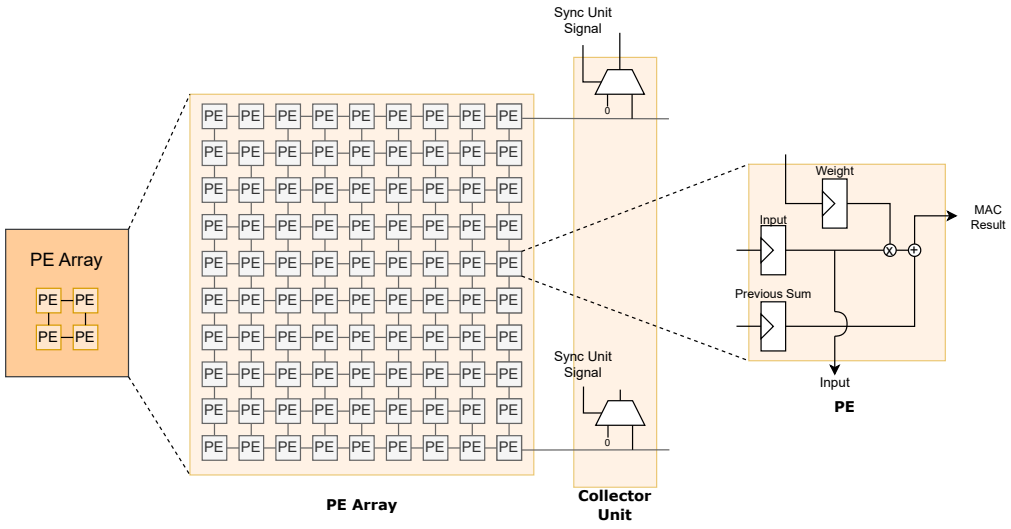

Figure 5.9: Details of PE array, PE, and Collector Unit.

The following section explains the reasons behind architectural parameters in the design and clarifies how PE arrays and Collector Units operate together. We examine PEs inside the PE arrays and discuss their internal structure. We also explain the changes and additional requirements resulting from incorporating OS in the architecture.

5.4 PE Array and Collector Unit Design

5.4.1 PE Array Height and Width

Considering Figure 5.9, the height of the systolic array is defined according to the need for a maximum number of kernels that are used in parallel. It helps to perform convolution of a matrix with multiple kernels at the same time. In this context, k in the array dimension of $k \times 9$ is a variable that aims to exploit different kernels. The effect of this variable is examined in the experimental results section.

The width of the systolic array is set according to the minimum kernel size of 9. It is the base number for array width because larger kernels can be stored with multiples of 9. Due to the nature of convolution, as also discussed in the Background section, the convolution of a point in an input matrix is computed when the center of a kernel matrix resides on that point. So, the relationship of the input matrix element with its neighbor elements is concerned. Since the minimum kernel size, larger than 1×1 and has a center point, is 3×3, nine is selected as systolic array width. The architecture can store elements of 3×3 kernels along a row of one PE array.

5.4.2 Number of PE Arrays

The number of PE arrays and Collector Units is determined according to the maximum kernel size and utilization of PE arrays together. The PE array width is set according to the minimum kernel size. The Collector Unit comes into play when working with larger kernels. Connecting multiple PE arrays allows the architecture to work with larger kernels. So, duplicate rows of different PE arrays can be used together, in a sense, extending rows. This extension can be done with the Collector Unit up to a point determined by the maximum kernel size and utilization of PE arrays.

An essential part of network design is selecting kernel sizes for convolution. Investigating various CNN models like LeNet [42], AlexNet [43], VGGNet [44], GoogleNet [45], ResNet [46], DenseNet [47] and EfficientNet [48]; there is not a convolution with larger than 11×11 kernels. Moreover, mostly 3×3 kernels are preferred because using smaller kernels consecutively rather than a larger one is more efficient [46]. Therefore, there is a general tendency towards using smaller kernels in network designs. Considering this trend, our accelerator's maximum kernel size is 11×11. It requires 121 elements to store, which 14 PE arrays can handle since 14 PE arrays can store up to $14 \times 9 = 126$ elements in one row. Next, we analyze the utilization of this base configuration and try to find the sweet spot. Table 5.1 shows the utilization of the architecture by indicating how many of the PEs are active considering different numbers of kernel sizes. Both static and reconfigurable architectures can compute convolutions with 11×11 kernels. Therefore, the PE array number starts from 14. The static case refers to the scenario where there is no extension, and the connections of the PE arrays cannot be reconfigured. The reconfigurable case refers to the architecture in which Collector Units are used. Kernel sizes are commonly used sizes in previously mentioned CNNs. As discussed earlier, even-numbered kernel dimensions are not used due to the lack of center points. From minimum 3×3 to maximum 11×11, all kernels are inspected.

Considering Table 5.1, utilization is better in reconfigurable architecture compared to static one as expected. Moreover, if we want a static architecture to work with bigger kernels, we lose utilization in smaller kernels. In reconfigurable architecture, the utilization drop is due to excess PEs and PE arrays. However, there is not a specific pattern. For example, a 7×7 kernel has 49 elements; at least 49 PE should be connected to accumulate the result. So, the architecture uses 6 PE arrays to process one channel of 7×7 kernel with the help of Collector Units. One row of 6 PE arrays has 54 elements, enough for this kernel size. Therefore, a group of 6 PE arrays are used for one channel. If the architecture has 14 PE arrays, 2 of the PE arrays will be idle. Also, 49 of the 54 PEs in a group will be used. 5 PEs in every 6 PE array groups will be idle.

Number of PE Array \Kernel Size		**3x3**	**5x5**	**7x7**	**9x9**	**11x11**
14	Static	7.1%	19.8%	38.8%	64.2%	96%
	Reconfigurable	100%	79.3%	77.7%	64.2%	96%
15	Static	6.6%	18.5%	36.3%	60%	89.6%
	Reconfigurable	100%	92.6%	72.5%	60%	89.6%
16	Static	6.3%	17.4%	34%	56.3%	84%
	Reconfigurable	100%	86.8%	68%	56.3%	84%
17	Static	5.9%	16.3%	32%	52.9%	79%
	Reconfigurable	100%	81.7%	64%	52.9%	79%
18	Static	5.6%	15.4%	30.2%	50%	74.7%
	Reconfigurable	**100%**	**92.6%**	**90.7%**	**100%**	**74.7%**
19	Static	5.3%	14.6%	28.7%	47.4%	70.8%
	Reconfigurable	100%	87.7%	86%	94.7%	70.8%
20	Static	5%	13.9%	27.3%	45%	67.3%
	Reconfigurable	100%	83.4%	81.7%	90%	67.3%

Table 5.1: Utilization of static and reconfigurable architectures with different numbers of PE arrays for different kernel sizes.

Considering these observations, the 18-PE array case is the best option. Utilization is at least 90.7% for all kernel sizes except 11×11. Because 3×3 kernel requires 1 PE array, 5×5 kernel requires 3 PE arrays, 7×7 kernel requires 6 PE arrays, and 9×9 kernel requires 9 PE arrays for processing one channel. More specifically, all these required PE array numbers (1, 3, 6, 9) are divisors of 18 leading to high utilization. If we take 11×11 into account, it needs 14 PE arrays, and the least common multiple of 1, 3, 6, 9, and 14 is 126. However, an architecture with 126 PE arrays would be drastically inefficient. The Toeplitz matrices for input and weight would require lots of zeros, leading to enormous sizes for these matrices. Besides, it would consume high power due to its size. Moreover, usage of 11×11 kernel is rare and encountered in older networks, as mentioned in the previous section. Our trade-off analysis indicates that enlarging the architecture in return for active PE utilization does not pay off. Therefore, we opt to work with 18 PE arrays in the default configuration.

5.4.3 Cooperation of PE Arrays and Collector Units

The cooperation of PE arrays and Collector Units provides a different form of reconfigurability on top of dataflow reconfigurability. The primary role of Collector Units is to manage various kernel sizes efficiently without compromising performance. A PE array group includes multiple numbers of PE arrays according to kernel size. A different group of these PE arrays works on other channels. Figure 5.10 illustrates the focus of the dimensions of the accelerator.

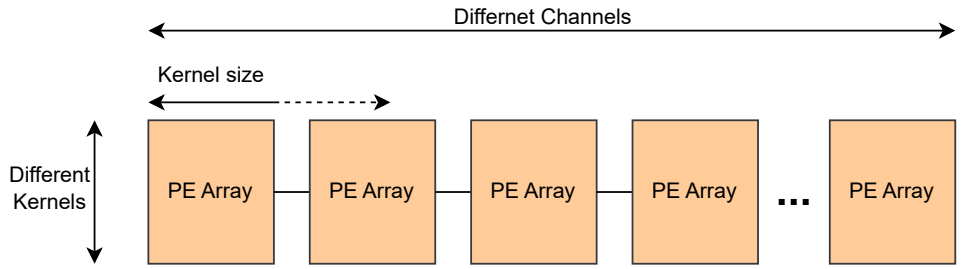

Figure 5.10: Accelerator reconfiguration using different PE array dimensions.

As discussed, PE array height k is responsible for different kernels. PE array width is responsible for kernel size and is set as 9. Kernel size can be adjusted by combining multiple PE arrays with the help of Collector Units. Several PE arrays for this kernel size form a PE array group and process one convolution channel. Multiple PE groups are responsible for different channels.

Figure 5.11 illustrates how different channels can be processed for various kernel sizes, where channels of the same color represent identical processing tasks. The arrows near PE arrays represent results. We can collect results after different PE arrays depending on the kernel size. Collector Units either collect or pass the accumulations. If the computation is done, the result is collected. Otherwise, accumulation continues. A PE array group, which includes a different number of PE arrays according to kernel size, is responsible for one channel. Another number of channels can be processed in parallel depending on the kernel size. Essentially, this Collector Unit extension allows for dynamic adjustment of the channels processed by the underlying hardware, ensuring adaptability to different computational requirements.

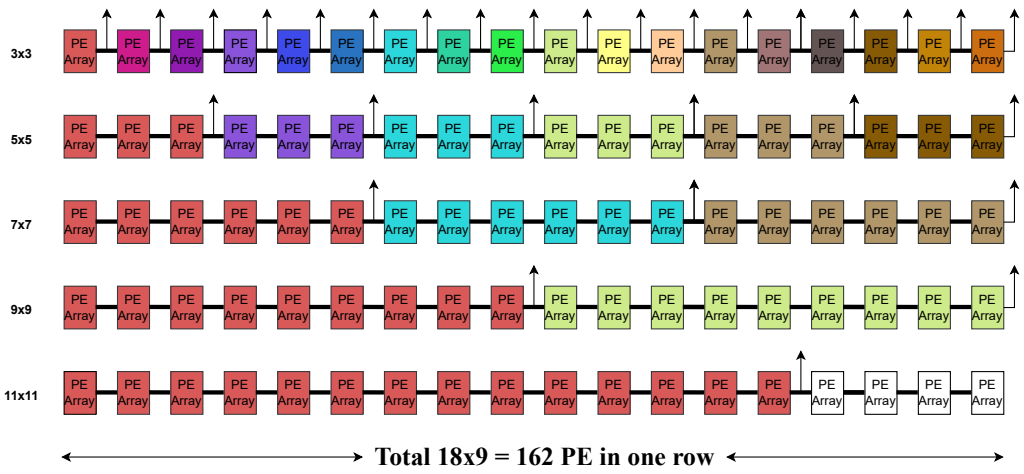

Figure 5.11: PE array adjustment for different kernel sizes.

An example is given in Figure 5.12. Green PEs and arrows represent active PEs and paths, respectively. Red ones represent idle PEs and paths. A 5×5 kernel requiring 25 weights can be accommodated within a minimum of 3 PE arrays containing 27 PEs (9×3). Results are extracted from every 3 PE arrays; the last 2 PEs in every 3 PE arrays do not make a computation and pass the result, which is red. In this setting, there is no excess PE array that remains idle. This setup essentially allows simultaneous processing of k kernels and their six channels within the architecture. Since a PE array group responsible for one channel includes 3 PE arrays, 18 PE arrays are utilized. This approach enables efficient utilization of the architecture across various kernel sizes.

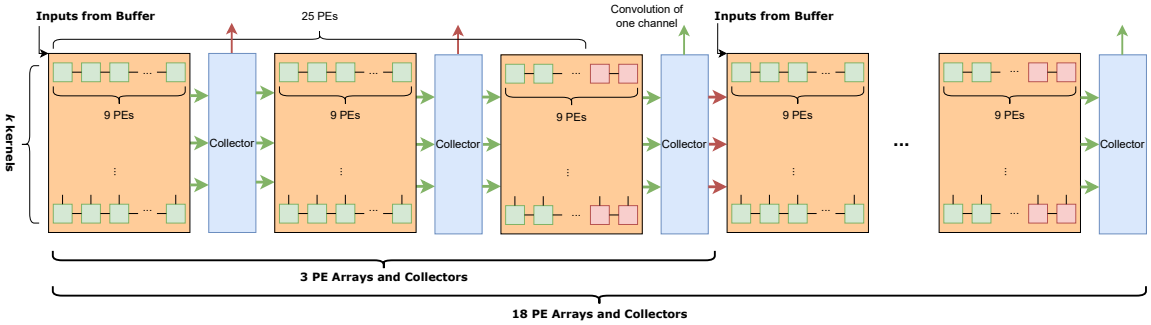

Figure 5.12: Example of a convolution with 5×5 kernel. 18/18 of the PE arrays are used. 25/27 of the PEs in every 3 PE arrays are used. So, six channels are processed.

5.4.4 Processing Elements

Figure 5.9 provides a detailed view of a single PE array. The extension on the right side of each PE array manages output collection. Depending on the filter size, connections can be adjusted to gather results from various PE arrays. Moreover, the PEs within these systolic arrays can adapt to different dataflow configurations with minor adjustments. In the upcoming section, we will analyze these structures in the context of WS and IS configurations and then discuss additional OS requirements. This examination will highlight the advantage of integrating WS and IS dataflows versus introducing OS into the architecture.

Each PE has a dedicated logic for MAC operations. Additionally, there are three registers with details explained below.

- The **weight register** is used for multiplication and can be updated during the load state when new weights are fetched.
- The **input register** facilitates multiplication and is updated every clock cycle. Inputs flow downward to the lower PEs within the PE array.
- The **previous sum** register is employed for accumulation. It enables the transfer of results from one PE to the next within the systolic array.

These registers are essential for executing MAC operations efficiently across the PE array. For WS and IS cases, the architecture remains unchanged structurally. The only difference lies in the inputs and weights managed by the registers. In WS mode, the weight registers are loaded with filter weights, while in IS mode, they receive input data. Likewise, the input registers are supplied with input data in WS mode, whereas they store filter weights in IS mode. This distinction is illustrated in the execution cycle diagram in the previous section, demonstrating the architecture's flexibility in accommodating different dataflow configurations without requiring modifications to its core structure.

5.4.5 Incorporating OS

While WS and IS configurations can directly be implemented in our architecture, integrating OS such as WS⊎OS, IS⊎OS, or WS⊎IS⊎OS requires a different look. In the OS case, the dimensions of the PE arrays are reevaluated because kernel size becomes irrelevant. Since accumulation occurs within PEs, inputs, and filters flow simultaneously in each cycle. Figure 5.13 illustrates the dimensional impact in the OS configuration. The extension part of each PE array solely passes results. All arrays collectively function as a $k \times 162$ systolic array, where 162 comes from 9 PEs for 18 PE arrays (18×9). It emphasizes that kernel size does not dictate the storage of inputs or weights. Each PE in this setup corresponds to different spatial locations in the output, and each row is dedicated to other kernels.

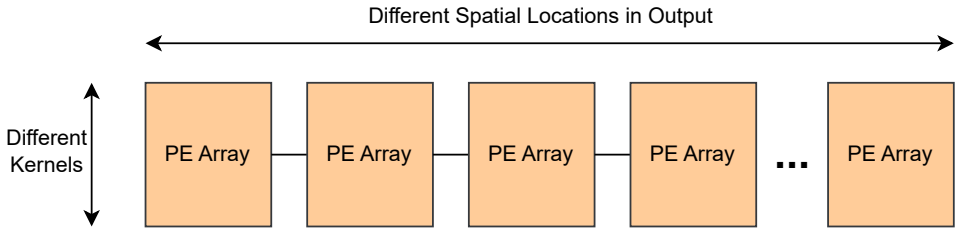

Figure 5.13: Accelerator reconfiguration using different PE array dimensions for OS case.

Figure 5.14 depicts the PE suitable for OS, featuring an accumulator instead of a previous sum register, where results are stored inside the PE in each cycle. Figure 5.9 shows that the PE structure suits both WS and IS configurations. However, combining OS with WS, IS, or both necessitates modifying the PE structure, as shown in Figure 5.15. This modification clearly shows that integrating OS into the architecture complicates the design compared to WS and IS configurations.

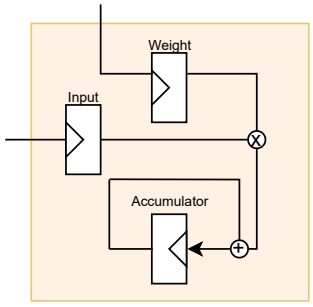

Figure 5.14: PE for OS case.

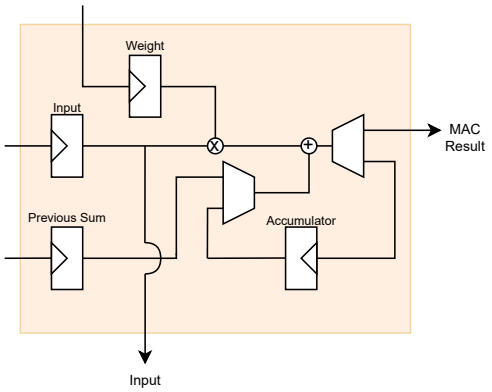

Figure 5.15: PE for WS⊎IS⊎OS case.

5.5 Convolution Engine

Convolution Engine is responsible for

- selecting dataflow depending on the convolution parameters,
- producing signals to handle synchronization among PE arrays and maintaining computation,
- managing stride and bias features in convolution.

The below subsections explain respective units and discuss possible architectural changes for different reconfigurable implementations.

5.5.1 Dataflow Selection Unit

The Dataflow Selection Unit within the convolution engine is required due to dataflow reconfigurability. Depending on the convolution parameters, it selects the dataflow to be applied. The algorithm's parameters and meanings are given in Table 5.2. The algorithm behind dataflow selection is given in Algorithm 7. The performance of the Dataflow Selection Unit is discussed in detail in the experimental results section.

Parameters	Meaning
IA	Input area
KV	Kernel volume
KA	Kernel area
KN	Kernel number
KN/KC	Ratio of kernel number to kernel channel (Larger kernel number means larger output channel, larger kernel channel means larger input channel)
AVG(OH,OW)	Average of output height and output width dimensions
thr	Threshold value for deciding between IS and OS dataflows

Table 5.2: Dataflow selection algorithm parameters.

Dataflow Selection Unit

if IA $>$ KV & IA $>$ KN $\times$ KA **then**
    WS $\rightarrow$ Dataflow
else if KN/KC $>$ AVG(OH,OW) $\times$ KA $\times$ *thr* **then**
    IS $\rightarrow$ Dataflow
else
    OS $\rightarrow$ Dataflow
end if

Algorithm 7: Dataflow selection algorithm.

Considering both the above algorithm and Chapter 4, WS is prioritized because there is not a WS dimension that can also be effectively exploited in IS or OS. If the input area is much larger than the kernel volume, we know it produces an output with a significant height and width, which WS exploits. So, one comparison is made between input area and kernel volume. Another comparison is made between the input area and the number of kernels because a large kernel number should imply the IS option. Suppose the input area is smaller in one of the comparisons, leading to a non-WS option. In the next step, a comparison between IS and OS is made. If the comparison is adjusted to set the threshold alone, the algorithm's 'else if' comparison turns into Expression 5.1. In this open form, the numerator focuses on distinctive features of IS and OS. In contrast, the denominator focuses on standard features of IS and OS, lowering the effect of unique features. If the standard part is dominant, it is hard to see a drastic change in results between IS and OS. A ratio of kernel number to kernel channel is made. Because IS exploits kernel numbers, and OS exploits kernel channels.

This ratio is divided by multiplying the average of the output area dimensions and the kernel area. We added the other commonly exploited dimensions to the denominator because these dimensions are exploited commonly. Kernel height (KH) and kernel width (KW) are exploited directly in both data, adding them as multiplication. Output height (OH) and output width (OW) values depend on input height (IH), input width (IW), kernel height (KH), and kernel width (KW) due to the nature of convolution. So, we added information from output that was not as strong as directly exploited dimensions by simply averaging output height (OH) and output width (OW). If this multiplication is more significant than a threshold *thr*, IS dataflow is selected; otherwise, OS. It is reasonable because the number of kernels makes this result significant, forcing it to select IS. If the kernel channel is much larger than the number of kernels, the ratio would be small, causing the Dataflow Selection Unit to choose OS. The reason for the threshold is to avoid IS or OS-biased decisions. Since kernels are smaller than inputs and outputs, the threshold behaves like a normalization factor and helps to prevent biased selections. If our dataflow is not WS, then *thr* behaves as a switch between IS and OS. The default *thr* value is set as 0.004 experimentally. The sensitivity analysis of this variable is further discussed in the experimental results section.

$$thr < \frac{KN/KC}{\left(\frac{OH+OW}{2}\right) \times (KH \times KW)} \tag{5.1}$$

While the Dataflow Selection Unit considers all dataflows, unnecessary comparisons are ignored for intermediate reconfigurabilities like WS⊎IS, WS⊎OS, or IS⊎OS. For example, if WS⊎IS reconfigurability is implemented, only WS checks would be enough. The selection unit does not require any modification when comparing intermediate configurations and WS⊎IS⊎OS architecture.

5.5.2 Synchronization Unit

The Synchronization Unit primarily generates signals to facilitate continuous computation within the accelerator. It incorporates cursors and counters to manage essential control signals that drive Collector Units. Table 5.3 shows signals generated by the Synchronization Unit and their purpose. On the other hand, Table 5.4 shows required signals for different dataflow options.

Signals	**Purpose**
Channel Iterator (CI)	Iterations over channel groups
Channel Cursor (CC)	Iterations inside a channel group
Kernel Iterator (KI)	Iterations over kernel groups
Kernel Cursor (KC)	Iterations inside a kernel group
Input Spatial Iterator (ISI)	Iterations over spatial groups of input
Input Spatial Cursor (ISC)	Iterations inside a spatial group of input
Output Spatial Iterator (OSI)	Iterations over spatial groups of output
Output Spatial Cursor (OSC)	Iterations inside a spatial group of output
PE Reset (PER)	Reset accumulation register (only in OS capable architectures)
Start Accumulate (SA)	Start accumulation (only in OS capable architectures)
Write Ready (WR)	Write results to output memory
Load Finished (LF)	Loading state is finished
Execute Finished (EF)	Execution state is finished
Store Finished (SF)	Store state is finished (only in OS capable architectures)

Table 5.3: Signals generated by the Synchronization Unit.

Dataflows	**Required Signals**
WS	CI, CC, KI, KC, ISC, OSC, WR, LF, EF
IS	CI, CC, KC, ISI, ISC, OSI, OSC, WR, LF, EF
OS	CC, KI, KC, ISI, ISC, OSI, OSC, PER, SA, WR, LF, EF, SF

Table 5.4: Required signals for different dataflow options. Note that the required signals are selected according to the loading schemes mentioned in the FSM discussion.

It can be seen that WS needs fewer signals than other dataflows. On the other hand, OS requires extra signals to control accumulation registers in PEs in OS-capable architectures.

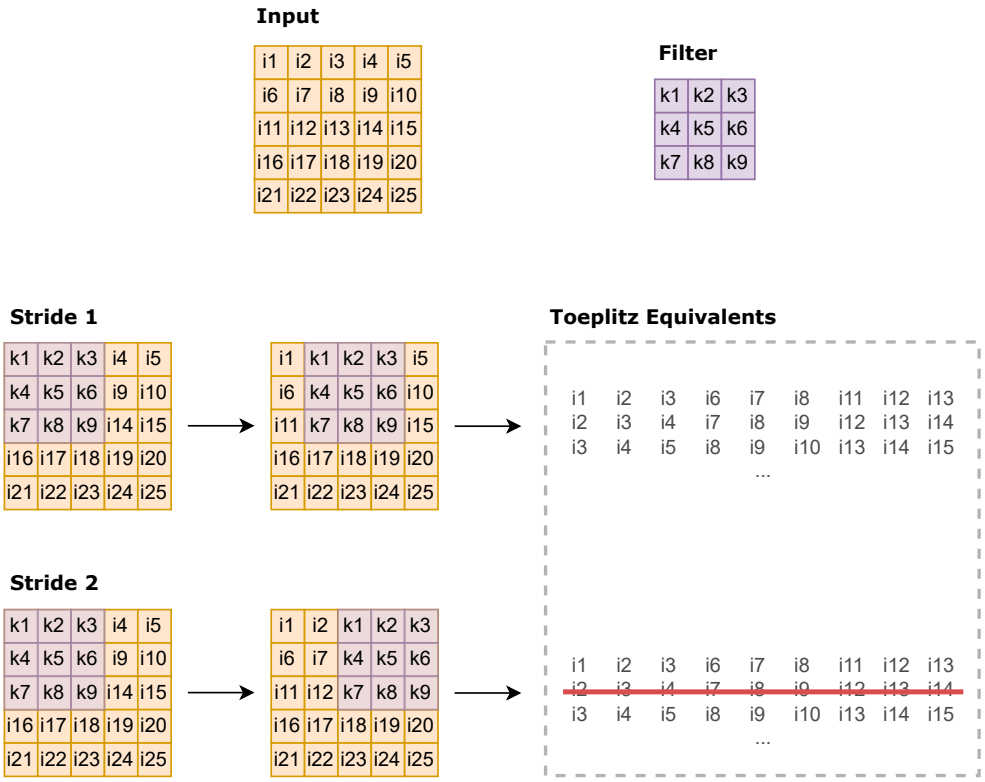

Figure 5.16: Stride effect in computation.

Combining different dataflows, such as WS⊎IS, WS⊎OS, IS⊎OS, or WS⊎IS⊎OS, alter the Synchronization Unit's design. These combinations typically require the addition of extra cursors and counters. The integration of OS significantly impacts the unit. The general purpose nature of this unit is the same among all architectures. However, generating required signals for OS-capable architectures can be costly. We will evaluate the difference in synchronization units regarding area and power between various dataflow configurations.

5.5.3 Convolution Options

This unit is responsible for how stride and bias options, explained in the Background section, are managed within the architecture. Regarding stride, adjustments are made to the Toeplitz matrix to ensure that the architecture processes related inputs and weights accordingly, effectively skipping over sections of the matrix. The optional units' cursors and counters are synchronized with the stride information to facilitate this operation. Figure 5.16 shows the effect of stride in

computation. As can be seen, some of the rows are skipped during loading. Regarding bias, input memory is used to store bias values and inputs. In both WS and IS configurations, bias is added to the result during the final accumulation of an element in the output matrix. This involves traversing all channels for that specific output location and concluding the accumulation process by incorporating the bias value. The store state collaborates with bias incorporation for OS scenarios during the storing process. Each result is stored after bias addition, ensuring consistency across different computational modes.

5.6 Memories

Efficient data access is crucial for accelerated computation, making memory management a critical component. The architecture, depicted in Figure 5.1, features three SRAM memories: inputs, weights, and outputs. Inputs and weights are stored in a Toeplitz matrix format optimized for convolution operations, while the output memory stores results in a standard 2D format. Data retrieval involves two main steps facilitated by dedicated buffers for each memory type. The input buffer fetches 18×9 data segments from the input memory, feeding these to the uppermost PE line across all PE arrays. Similarly, the weight buffer retrieves $k \times 9$ data segments from the weight memory and distributes them to the respective PE array during the load state. During the execute state, each cycle processes convolution operations across PE arrays. For instance, as discussed previously, in a 5×5 filter convolution, every three PE arrays handle one channel. The resulting outputs undergo a reduction operation within the PE arrays, aggregating results from different channels. This aggregation yields k results for each kernel. After reduction, the results from every three PE arrays are merged and written to the output buffer. The buffer temporarily holds k data points, which are transferred to the output memory. In the subsequent execute state, results from the next channel undergo a similar reduction process and accumulate with existing output locations. This accumulation ensures that results from all channels are progressively stored in the output memory.

All IS, WS, and OS configurations require essential storage elements within the architecture. The critical distinction for OS is the simultaneous input and weight data requirement in every cycle, as PEs do not retain either. From an architectural perspective, each dataflow necessitates the generation of distinct control signals but fundamentally relies on the same storage elements. Therefore, combining different dataflows does not impact memory requirements, as the essential storage elements remain the same across all configurations.

Chapter 6

Experimental Evaluation

This chapter gives details about the accelerator design environment and experimental setup. We used three tools as design and test environments. To evaluate the performance of our accelerator, various convolutional neural networks and the parameters of their convolutional layers are also discussed.

6.1 Design and Test Environment

The accelerator is designed using Chisel [49]. Chisel, which stands for Constructing Hardware in a Scala Embedded Language, is a hardware description language developed to streamline the design of digital hardware circuits. Embedded in Scala, Chisel offers tools to describe complex hardware structures, allowing designers to leverage high-level programming constructs and functional paradigms to create flexible and reusable hardware components. Cycle-accurate simulations are performed on the RTL, which Chisel generates, ensuring that the design meets the required timing and functionality before it moves to the synthesis stage.

The designed RTL is subsequently synthesized using the Cadence Genus [50] tool with the TSMC 40nm CMOS technology library, optimizing the design for area, power, and timing constraints. Genus translates the RTL description into

a gate-level netlist during synthesis, applying various optimization techniques to ensure that the hardware meets the specified performance criteria. Performance analyses, including power consumption, maximum clock frequency, and area utilization, are reported post-synthesis to verify that the design goals are achieved.

However, because the physical structure of memories extends beyond the Boolean logic that most synthesis tools are designed to handle, CACTI [51] is used to model the power and area of SRAM modules. CACTI provides a detailed analysis of the memory's characteristics, such as access time, power dissipation, and area footprint, allowing for more accurate results in the overall performance evaluation. The accelerator's design is refined by integrating these tools to balance high performance with efficient resource usage.

6.2 Implementation Alternatives

In our experimental evaluations, we used seven different alternatives for comparison. In addition to our accelerator representing a WS⊎IS scenario, we have implemented the non-reconfigurable WS (Origami [19]), IS (SCNN [24]), and OS (ShiDianNao [6]) cases. Furthermore, we have implemented the reconfigurable WS⊎OS, IS⊎OS, and WS⊎IS⊎OS (FlexFlow [4]). We have created a setup that can be modified to represent these different implementations. Therefore, we perform experiments using the same settings and the same parameters to obtain a fair comparison with the literature.

Considering other dataflows, Table 6.1 presents area and power consumption results for different combinations: WS, IS, OS, WS⊎IS, WS⊎OS, IS⊎OS, and WS⊎IS⊎OS. Note that referenced architectures are not exact implementations. Instead, they are our implementations of the representative dataflows in our framework. To support such variations, we customized the same architecture and implemented the different dataflow options in the literature.

Architectures	Area (mm^2)	Power (mW)
WS (Origami [19])	1.98	66.76
IS (SCNN [24])	1.99	67.49
OS (ShiDianNao [6])	2.07	51.00
Accelerator (WS⊎IS)	2.02	68.09
WS⊎OS	2.63	79.74
IS⊎OS	2.68	81.02
WS⊎IS⊎OS (FlexFlow [4])	2.80	92.51

Table 6.1: Properties of different implementations used in our experimental evaluation. Area and power results for each implementation are provided here for reference. Note that referenced architectures are not exact implementations. They are representative examples of the corresponding dataflow.

Each implementation's area and power results are provided here for reference but will be explained in more detail in the experimental evaluation chapter. As seen from the second column of Table 6.1, the area required by each architecture is different, ranging from 1.98 mm^2 to 2.80 mm^2. The third column of this table shows the power consumption of each alternative. Depending on the complexity of the design and used hardware units, the power consumption ranges from 51 mW to 92 mW. This indicates a possible trade-off analysis that we will explore in our experiments.

6.3 Benchmarks

Three different convolutional neural networks are tested throughout the experiments. AlexNet [43], LeNet [42], and VGG-16 [52] are landmark CNN architectures that have significantly influenced the development of deep learning. LeNet, developed by Yann LeCun in the late 1980s and early 1990s, was one of the first CNNs designed for handwritten digit recognition (e.g., MNIST dataset [53]). It consists of a few convolutional and pooling layers, followed by fully connected layers, demonstrating the effectiveness of CNNs in image classification. AlexNet, introduced by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton in 2012, won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) and brought

deep learning into the mainstream. It featured deeper layers, ReLU activation functions, and GPU acceleration, setting new performance standards. VGG-16, proposed by the Visual Geometry Group (VGG) at the University of Oxford, extended the depth of CNNs with 16 weight layers. It employed tiny (3x3) convolution filters, proving that deep networks with a uniform architecture could achieve high performance. These architectures are foundational in computer vision, influencing subsequent models' design and advancing neural network capabilities in image recognition tasks. Tables 6.2, 6.3, and 6.4 give the convolution layers and parameters for AlexNet, VGG-16, and LeNet, respectively.

Layers	**Input Size**	**Filter Size**	**Number of Kernels**	**Stride**	**Padding**
Conv1	227x227x3	11x11	96	4	-
Conv2	27x27x96	5x5	256	1	2
Conv3	13x13x256	3x3	384	1	1
Conv4	13x13x384	3x3	384	1	1
Conv5	13x13x384	3x3	256	1	1

Table 6.2: AlexNet convolution layers and their parameters.

Layers	**Input Size**	**Filter Size**	**Number of Kernels**	**Stride**	**Padding**
Conv1	224x224x3	3x3	64	1	1
Conv2	224x224x64	3x3	64	1	1
Conv3	112x112x64	3x3	128	1	1
Conv4	112x112x128	3x3	128	1	1
Conv5	56x56x128	3x3	256	1	1
Conv6	56x56x256	3x3	256	1	1
Conv7	28x28x256	3x3	512	1	1
Conv8	28x28x512	3x3	512	1	1
Conv9	14x14x512	3x3	512	1	1

Table 6.3: VGG-16 convolution layers and their parameters.

Layers	**Input Size**	**Filter Size**	**Number of Kernels**	**Stride**	**Padding**
Conv1	32x32x1	5x5	6	1	-
Conv2	14x14x6	5x5	16	1	-

Table 6.4: LeNet convolution layers and their parameters.

For the above tables, input sizes are given as *input height* $\times$ *input width* $\times$ *input channel*. On the other hand, kernels are represented by *kernel height* $\times$ *kernel width*. It can be seen that smaller kernels are preferred. The reason lies in the *receptive field* [54] concept. Applying two consecutive 3×3 convolutions

to a matrix corresponds to a computation over 25 elements; the same applies to a single 5×5 convolution. So, using two 3×3 convolution can be preferred instead of one convolution with a 5×5 kernel because it has fewer parameters. Some larger kernels are also used, where 11×11 is the largest. We should also emphasize that our accelerator has a 100% utilization rate for 3×3 kernels. It can also perform convolution up to 11×11 kernels.

6.4 Default Parameters

Default parameters for the architecture are given in Table 6.5. Due to the reconfigurable nature of our accelerator, we performed tests with different PE array heights and Dataflow Selection Unit threshold values. The variable k for kernel number is selected as 10 throughout the evaluations. This is a design choice determined according to filter numbers in convolutions. If it is too small, too many iterations on the load state would be required. If it is too large, the utilization at the last iteration could be too low due to the processing of excess data. In Section 7.9, we give a detailed sensitivity analysis based on this parameter. The threshold *thr* in the Dataflow Selection Unit is another variable for the architecture. Its default value is set as *0.004*, determined experimentally to avoid any bias towards OS or IS. Above this value, it starts to make OS-biased decisions. Similarly, below this default threshold value, IS-biased choices are made. Sensitivity analysis of threshold value will also be examined.

Parameter	**Value**
Dataflow reconfigurability	WS⊎IS
PE array width	9
PE array height (k)	10
Number of PE arrays	18
Total number of PEs	1620
Dataflow Selection Unit threshold (thr)	0.004
On-Chip SRAM sizes	224 KB
Arithmetic precision	8-bits

Table 6.5: Default parameters used in our experiments.

The design specifications for the architecture according to default parameters are given in Table 6.6. It can be seen that core area and power values form a small fraction of total chip area and power values. This is due to memory blocks required for reuse in convolution. In terms of the number of PEs, there are a total of 18 PE arrays each of which has 90 PEs. Therefore, there are 1620 PEs (18×90) with 20ns clock.

Technology	TSMC 40 nm
Clock Rate	50 MHz
Chip Area	17.883 mm^2
Core Area	2.02 mm^2
Chip Power Consumption	824.05 mW
Core Power Consumption	68.09 mW

Table 6.6: Design specifications of the hardware implementation.

6.5 Accuracy

CNN accelerators are primarily designed to accelerate inference rather than training [55, 56]. The inference is usually performed online and does not involve complex operations like backpropagation. However, these kinds of complex operations are necessary during training. Due to these complex operations required for the training, accelerators are typically built to work with pre-trained networks [57].

A vital training process step is quantization, as Chapter 3 highlights. Quantization allows models to perform inference using lower precision data, often reducing bit-width to 8 bits. Due to its efficiency and performance, this 8-bit inference has been widely adopted in numerous applications, particularly in edge computing scenarios [58, 59, 60].

Quantization enables faster computations, lower power consumption, and reduced storage requirements, which are ideal for resource-constrained environments like edge devices. In most cases, CNN accelerators are optimized for inference on pre-trained 8-bit models, ensuring that the models can run without requiring changes. The conversion from 32-bit to 8-bit is an earlier step before

inference. So, the reduction in accuracy resulting from this conversion is independent of the software or hardware architecture used to run the model. Since these accelerators do not alter the pre-trained 8-bit models, their accuracy remains unaffected. Therefore, accuracy analysis is often unnecessary [1, 4, 5, 19, 27, 39].

We also opted to use 8-bit model inference for our architecture. Several tests are done on different images from different datasets to verify the accelerator's convolution operations. ImageNet [61], CIFAR-10 [62], and MNIST [53] datasets are used for this purpose. They are popular datasets in the field of computer vision. ImageNet contains over 14 million images labeled across 20,000 categories, while CIFAR-10 consists of 60,000 32x32 color images in 10 classes. MNIST features 70,000 grayscale images of handwritten digits. Images are cropped if necessary to work as input to AlexNet, VGG16, and LeNet models. These images are convolved with random kernels that are suitable for convolution shapes in Tables 6.2,6.3,6.4. We selected random images for each dataset and convolved with different kernels. The results obtained from the accelerator are compared and verified against CPU results. All computations and convolution operations are correctly performed, verifying the accelerator as expected.

Chapter 7

Experimental Results

7.1 Dataflow Results

We constructed an accelerator operating in all configurations to compare all possible dataflows. As outlined in previous sections, the necessary registers and blocks are incorporated for each dataflow, resulting in an accelerator compatible with all three configurations. This section presents the performance of the dataflow under various convolution parameters.

Initially, our focus will not be on reconfiguration. However, we will explore the optimal reconfigurability options, considering both the dataflows' cost and performance. In the following sections, we aim to address questions such as: "Is it worthwhile to combine a dataflow?" and "Do we achieve a reasonable speedup in return for the additional units required to construct the accelerator?"

The accelerator discussed in this section can perform computations in WS, IS, and OS dataflows. As mentioned in the previous section, the convolution parameters of three different CNNs are utilized. Figures 7.1, 7.2, and 7.3 display the number of cycles required to complete the convolution operation for the given parameters. Different convolutions are presented on the x-axis, whereas the y-axis shows the number of cycles needed to finish the convolution.

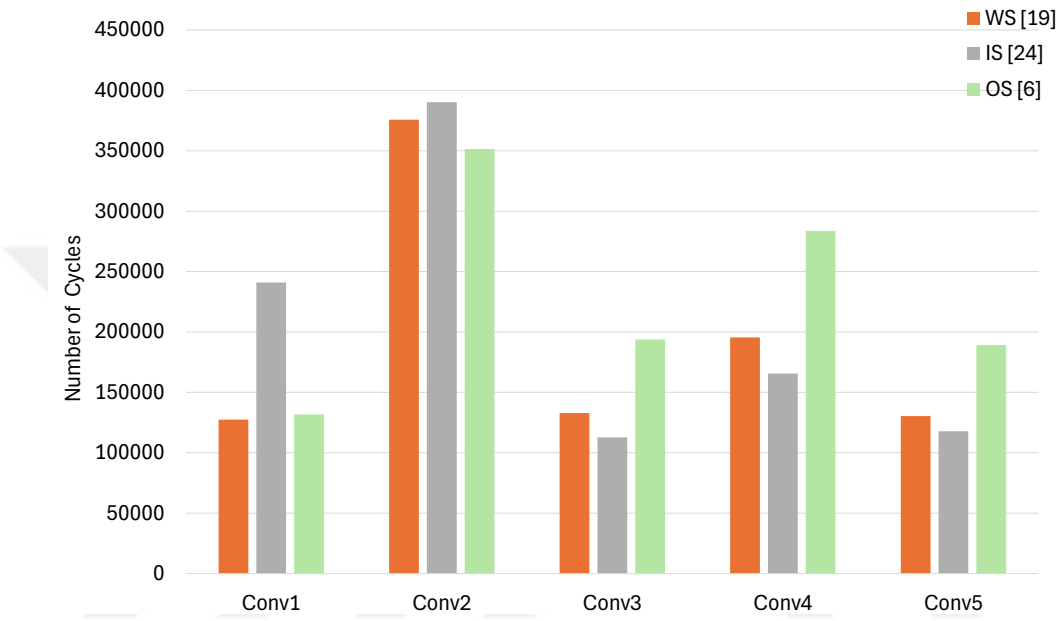

Figure 7.1: Computation cycles of AlexNet for different dataflows.

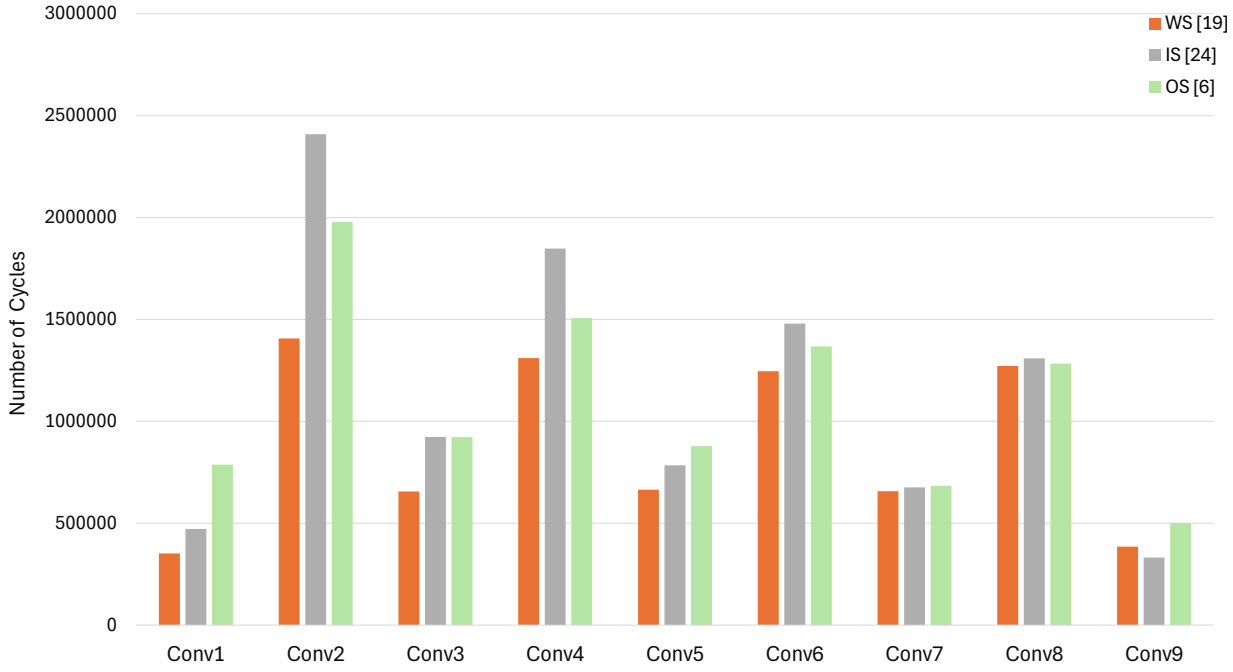

Figure 7.2: Computation cycles of VGG-16 for different dataflows.

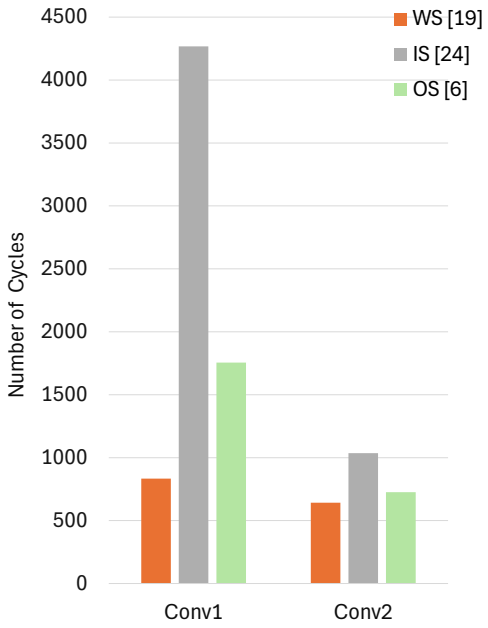

Figure 7.3: Computation cycles of LeNet for different dataflows.

As illustrated in the figures, different dataflows perform better with varying parameters. This observation aligns with the exploitable dimensions discussed in the dataflow and reconfigurability analysis section.

In AlexNet, the IS performs better for the last three convolutions, which is expected since the number of kernels (OC) provides opportunities for exploitation. For the other convolutions, the WS is better in the first case, and the OS is better in the second case, which is also coherent with exploitable dimension analysis in Chapter 4. In VGG-16, the WS is the optimal choice in most cases, although the IS occasionally performs better. In LeNet, WS consistently performs best, while IS has a significant overhead. OS is comparable to WS but outperforms IS. For all three networks, the performance of WS and IS is similar and often better than OS in most cases. There is only one convolution parameter where OS is the best; otherwise, WS or IS are preferable. Up to this point, it has been observed that combining WS and IS dataflows is architecturally straightforward. Furthermore, both WS and IS generally exhibit strong performance across various scenarios.

7.2 Performance Results

Performance results of different architectures are given in Figure 7.4. This figure compares computation latencies for the average of all convolutions. Considering the previous section, reconfigurable architectures try to select the best option for each convolution. As expected, WS⊎IS⊎OS architecture performs best as it can utilize all dataflows. The second best option with a slight difference is our accelerator architecture (with a WS⊎IS setup). It is better than all non-reconfigurable architectures and only 0.2% worse than the optimal case (WS⊎IS⊎OS). We observe that all alternative implementations with WS dataflow generate good results. Therefore, it is the most critical dataflow and explains why it is widely adopted in the literature. We can also conclude that integrating OS into the accelerator does not improve the convolution latencies significantly, as expected and indicated in previous sections. In the following two sections, we will discuss different architectures' area and power results and explain why adding an OS option to the architecture is unnecessary.

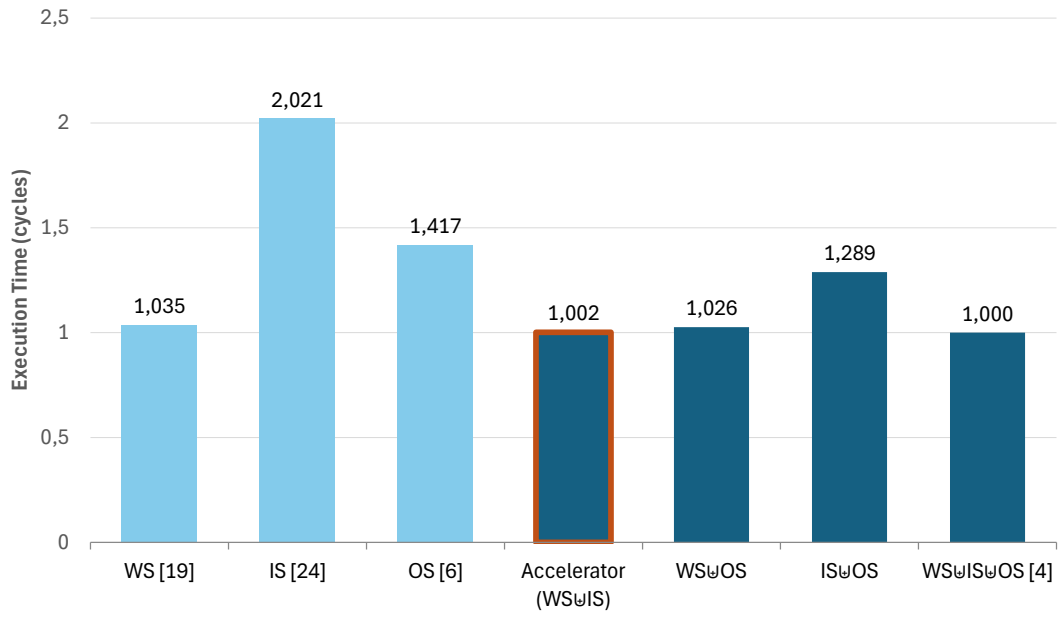

Figure 7.4: The average execution times of all convolutions on different architectures. Results are normalized with respect to WS⊎IS⊎OS architecture. Lighter shades of blue represent non-reconfigurable architectures, while darker shades indicate reconfigurable architectures.

7.3 Power Results

Power results of different architectures are given in Figure 7.5. These results are independent of CNN models and their convolutions. We can see from this figure that a non-reconfigurable OS is the best option for power with 51 mW. The reason for lower power consumption lies in the accumulator inside the PE. Even though the accumulator increases area, OS dataflow keeps the whole computation inside the PE without requiring any data transfer during the computation, contrary to other dataflows. The communication reduction is the key to the reduced power consumption. On the other hand, our accelerator consumes 68 mW, which is better than all other reconfigurable architectures and is very close to non-reconfigurable ones. Combining OS with other dataflows significantly impacts the power consumption of the architecture. However, WS and IS are not costly to merge, which verifies our claim in previous sections.

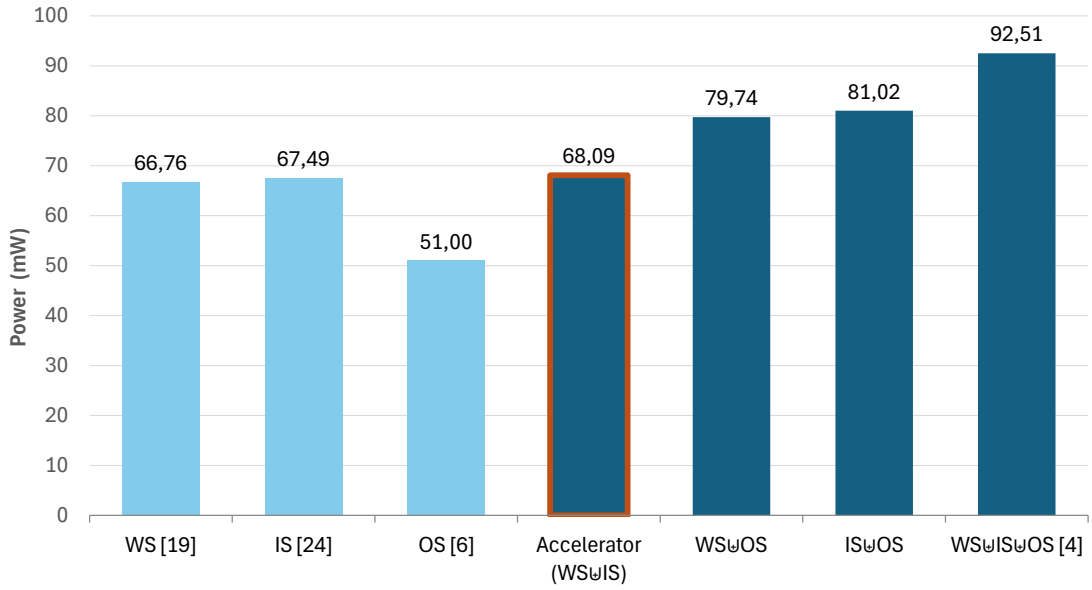

Figure 7.5: Power consumption (mW) of different architectures. Lighter shades of blue represent non-reconfigurable architectures, while darker shades indicate reconfigurable architectures.

7.4 Area Results

Figure 7.6 shows area results of different architectures. Similar to power results, it is also independent of CNN models and their convolutions. As expected, non-reconfigurable WS and IS options are the least area-demanding architectures with close results. Specifically, they both require almost $2mm^2$ area. If we further investigate the figure, our accelerator stands out with an outstanding result, much lower than reconfigurable architectures with a $2.02mm^2$. It is only 1% more than the minimum area and even better than the non-reconfigurable OS option. Furthermore, OS-capable reconfigurable accelerators (WS⊎OS, IS⊎OS, and WS⊎IS⊎OS) increase the area by at least 30%. It clearly shows that designing an OS-capable accelerator leads to higher area requirements. On the other hand, our accelerator requires only 2% and 1.5% more area compared to WS and IS options, respectively.

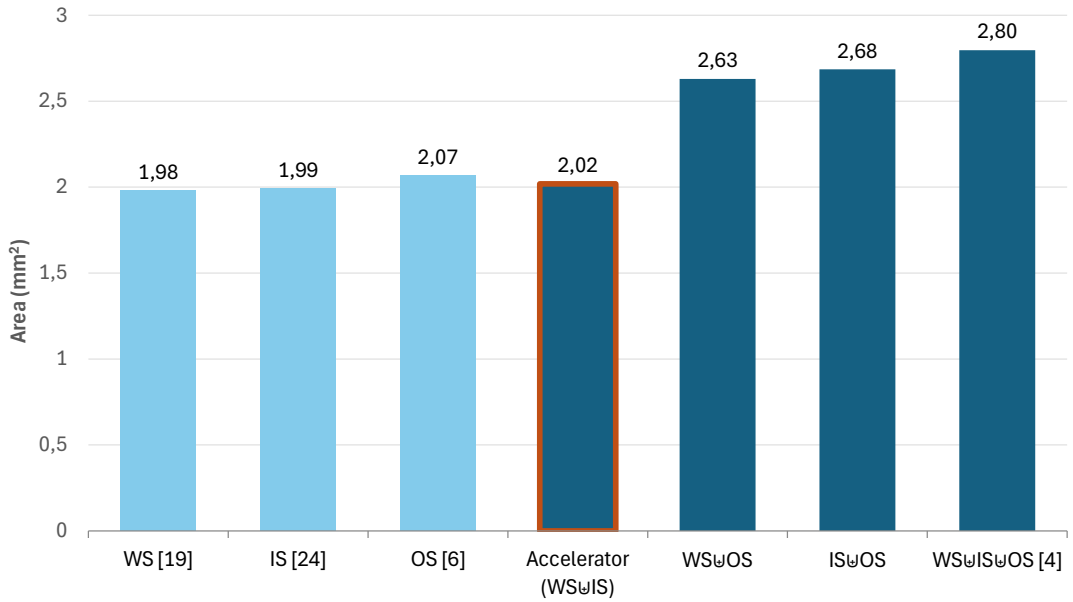

Figure 7.6: Area (mm^2) of different architectures. Lighter shades of blue represent non-reconfigurable architectures, while darker shades indicate reconfigurable architectures.

7.5 Dataflow Selection Unit Performance

Dataflow decision performance is crucial because we build a reconfigurable architecture and must benefit from reconfigurability. It is meaningful if only the Dataflow Selection Unit selects the most suitable dataflow, which is computed faster than others. Table 7.1 shows the performance of our dataflow. The second column represents the ideal dataflow that should be selected, i.e., the fastest one in the dataflow results above. The third column shows the choice of our Dataflow Selection Unit. Considering 16 convolution layers from 3 different CNNs, our decision-maker successfully selects 15 dataflows. The only exception is Conv8 in the VGG-16 model. This convolution's total cycle difference between the WS and IS dataflows is only 2.4%

Convolutions	**Ideal**	**Dataflow Selection Unit**	**Excess Latency**
AlexNet-Conv1	WS	WS	-
AlexNet-Conv2	OS	OS	-
AlexNet-Conv3	IS	IS	-
AlexNet-Conv4	IS	IS	-
AlexNet-Conv5	IS	IS	-
VGG-16-Conv1	WS	WS	-
VGG-16-Conv2	WS	WS	-
VGG-16-Conv3	WS	WS	-
VGG-16-Conv4	WS	WS	-
VGG-16-Conv5	WS	WS	-
VGG-16-Conv6	WS	WS	-
VGG-16-Conv7	WS	WS	-
VGG-16-Conv8	**WS**	**OS**	**2.4%**
VGG-16-Conv9	IS	IS	-
LeNet-Conv1	WS	WS	-
LeNet-Conv2	WS	WS	-

Table 7.1: Dataflow Selection Unit performance results.

7.6 Synchronization Unit Evaluation

As discussed in Chapter 5, the synchronization unit produces different signals for different dataflow options. Considering signals, WS and IS dataflows require similar signals, while OS needs extra signals to control the accumulation register in PEs. Figure 7.7 shows synchronization unit areas, and Figure 7.8 shows synchronization unit power consumption for different architectures. It can be seen that WS⊎IS is the best among reconfigurable architectures. It has the least area and power consumption compared to reconfigurable architectures. Moreover, it is very close to results from non-reconfigurable architectures. This also shows that combining WS with IS is less costly than making an OS-capable architecture.

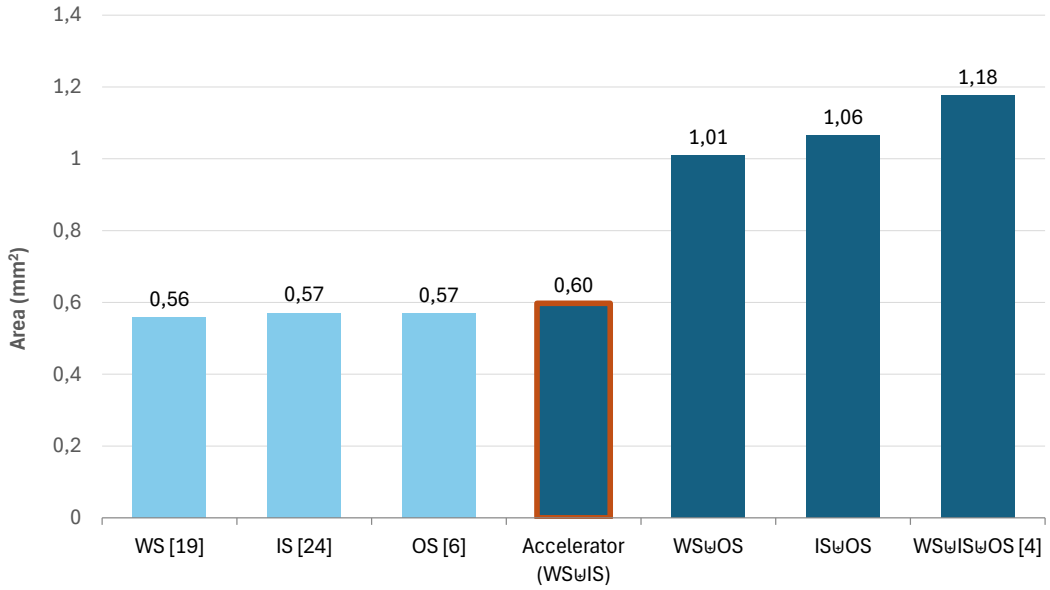

Figure 7.7: Area (mm^2) of synchronization unit for different architectures. Lighter shades of blue represent non-reconfigurable architectures, while darker shades indicate reconfigurable architectures.

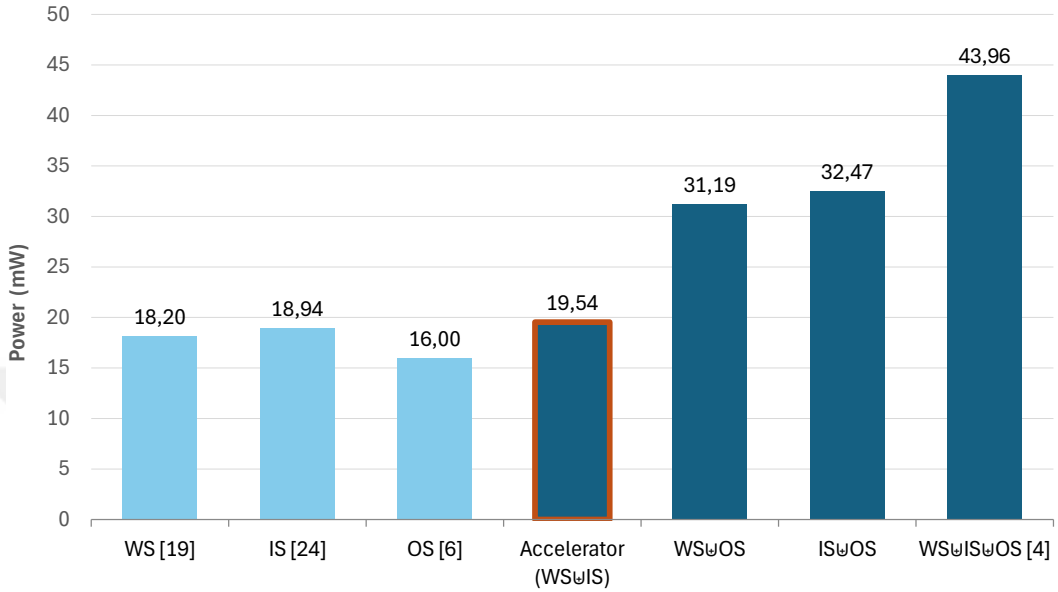

Figure 7.8: Power (mW) of synchronization unit for different architectures. Lighter shades of blue represent non-reconfigurable architectures, while darker shades indicate reconfigurable architectures.

7.7 Efficiency of Dataflow Reconfigurability

In hardware design, the efficient utilization of components is a critical consideration. This section compares various accelerators and underscores the significance of dataflow combination suitability in constructing hardware. Among many possibilities examined, we observed that the WS⊎IS option performs best compared to existing methods.

We evaluated reconfigurability efficiency using the most critical parameters of the design. Expression 7.1 considers speed, power, and area as explained in the literature [63, 64, 65]. More specifically, efficiency evaluation considers the combined effect of speed, power, and area. It is similar to the PPA (Power, Performance, Area) metric used in semiconductor applications.

$$Efficiency = \frac{1}{n} \sum_{n=1}^{layers} \frac{1}{cycles \times power \times area} \tag{7.1}$$

Figures 7.9, 7.10, and 7.11 illustrate the efficiency of different architectures, taking into account speed across various networks. The mean average cycle counts for all convolution layers within each network are also provided. For example, in Figure 7.2, the average cycle of each convolution in VGG-16 is calculated for different architectures. This analysis provides insights into the value of incorporating reconfigurability into our architecture for different dataflows. It highlights potential trade-offs between the overheads associated with reconfigurability and the advantages of speedup.

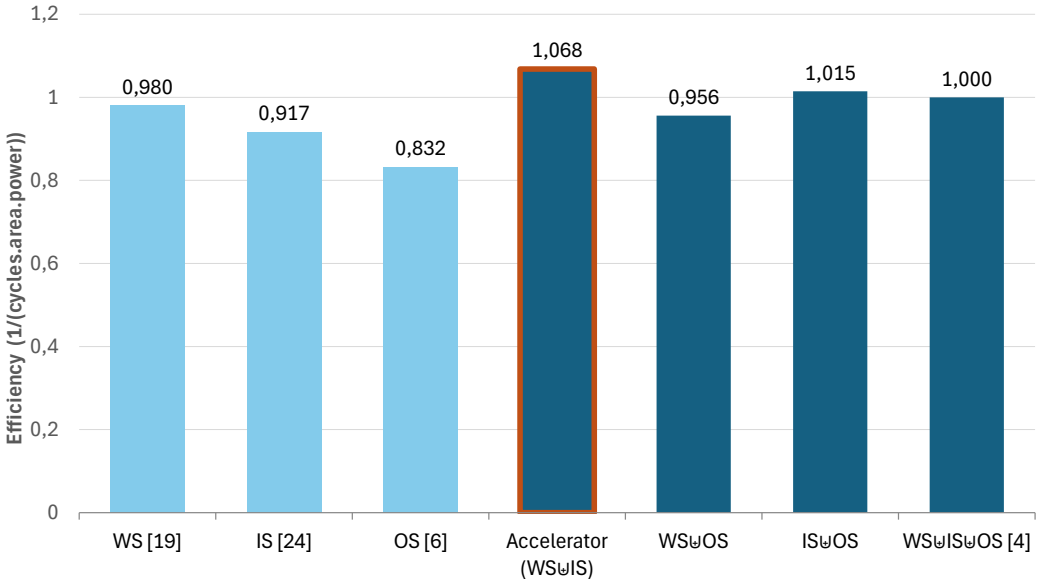

Figure 7.9: Efficiency of different architectures for AlexNet. Results are normalized to WS⊎IS⊎OS architecture. Lighter shades of blue represent non-reconfigurable architectures, while darker shades indicate reconfigurable architectures.

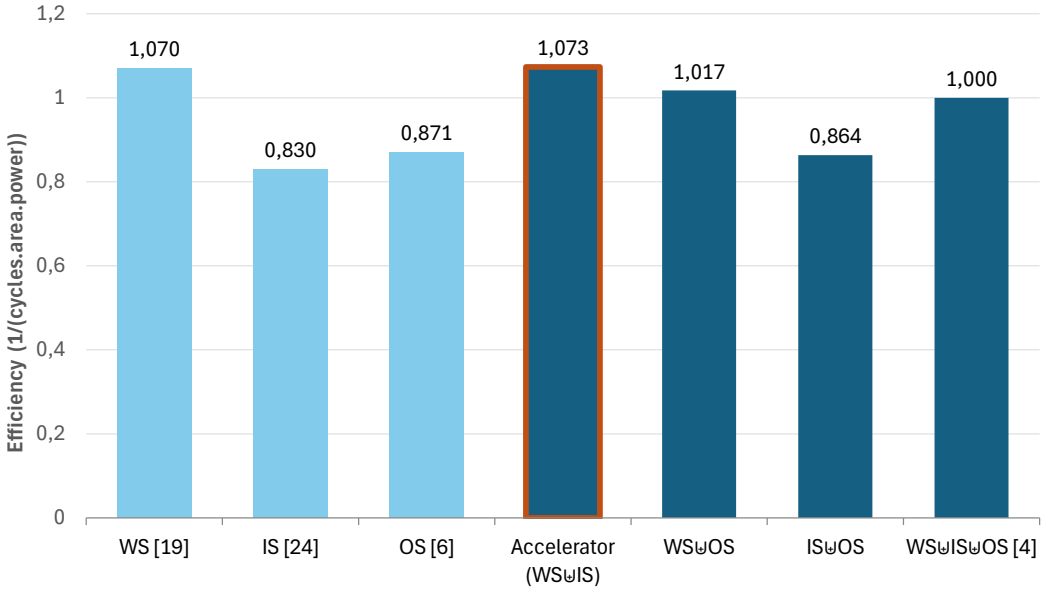

Figure 7.10: Efficiency of different architectures for VGG-16. Results are normalized to WSɯISɯOS architecture. Lighter shades of blue represent non-reconfigurable architectures, while darker shades indicate reconfigurable architectures.

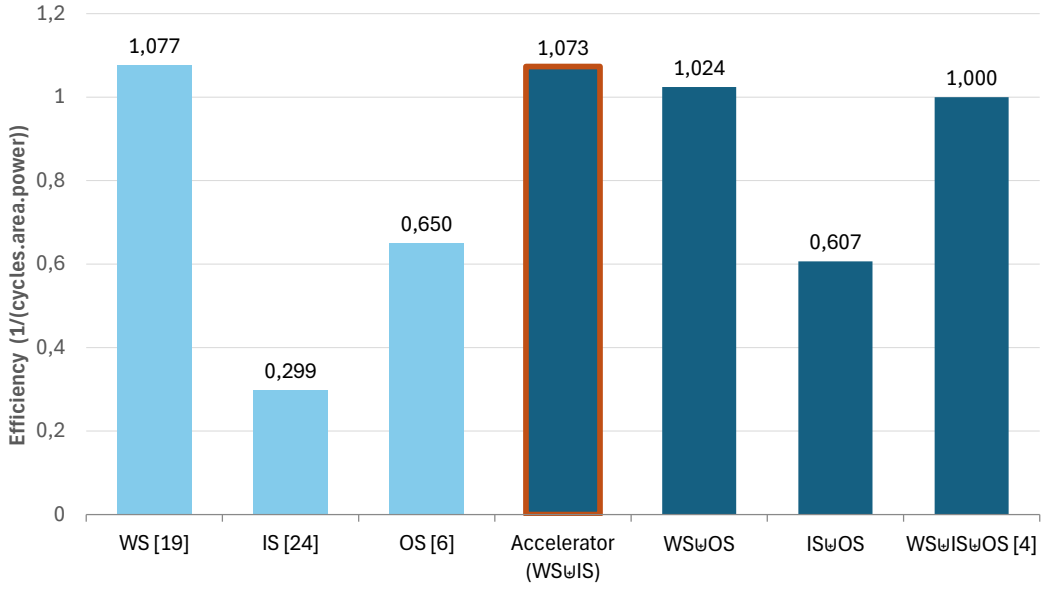

Figure 7.11: Efficiency of different architectures for LeNet. Results are normalized to WSɯISɯOS architecture. Lighter shades of blue represent non-reconfigurable architectures, while darker shades indicate reconfigurable architectures.

Based on our comparisons of different architectures, WS⊎IS emerges as the optimal choice when considering speedup, area, and power efficiency together. In AlexNet, WS⊎IS is significantly better than all other options. For VGG-16, WS is second best and closer to WS⊎IS. Only in LeNet, WS is the best option, and WS⊎IS is the second best with a slight difference. Because LeNet includes only two convolutions and is best suited to WS dataflow. As convolution layers increase, the best option for dataflow may change. Moreover, LeNet takes an input of 32×32 pixels, which is relatively lower than today's deep learning models. As a result, WS⊎IS stands out as the best option considering various CNN models. Besides, among reconfigurable architectures, WS⊎IS performs better than others significantly. As anticipated and discussed, combining WS⊎IS results in minimal additional area and power consumption, yet it delivers significant speedup through reconfigurability. Conversely, adding OS requires notable hardware differences and additional units, increasing area and power consumption. Despite OS's speed advantage through reconfigurability, it does not sufficiently offset the area and power overheads.

Therefore, it can be concluded that reconfigurability should be judiciously employed, considering the underlying hardware constraints. One needs to carefully apply reconfiguration to achieve the optimal balance of performance and efficiency. Based on our observations and experiments, WS⊎IS performs best overall.

7.8 Power and Area Breakdown

We analyzed the architecture regarding area and power, where breakdown results are given in Figures 7.12 and 7.13. As we discussed, most of the consumption comes from memories, both in terms of area and power. Besides the high memory requirement of the convolution operation, working with Toeplitz matrices increases the need for data storage. In addition, output memory has a smaller area and power demand because the result is stored in a standard 2D array, not in a Toeplitz way. A Toeplitz matrix engine can be constructed to transform some portion of the 2D array into Toeplitz form, which provides on-the-fly data. This may be critical for future work in this architecture to reduce memories' area and power demand. For the core of the architecture, most of the area and power consumption come from PE arrays, as expected. The convolution engine is the least demanding part of the architecture because it works as a controller that creates required signals.

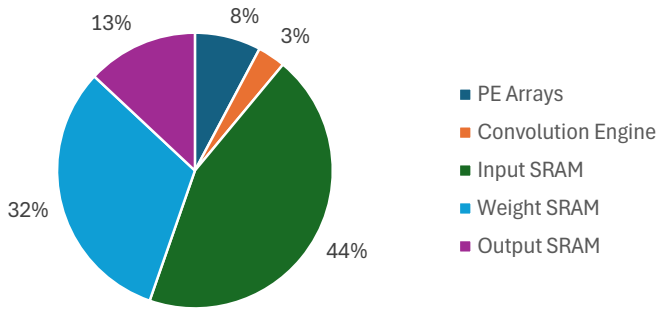

Figure 7.12: Area breakdown of our architecture.

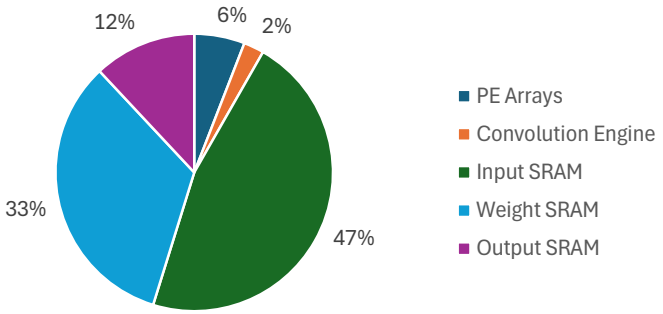

Figure 7.13: Power breakdown of our architecture.

7.9 Sensitivity Analysis

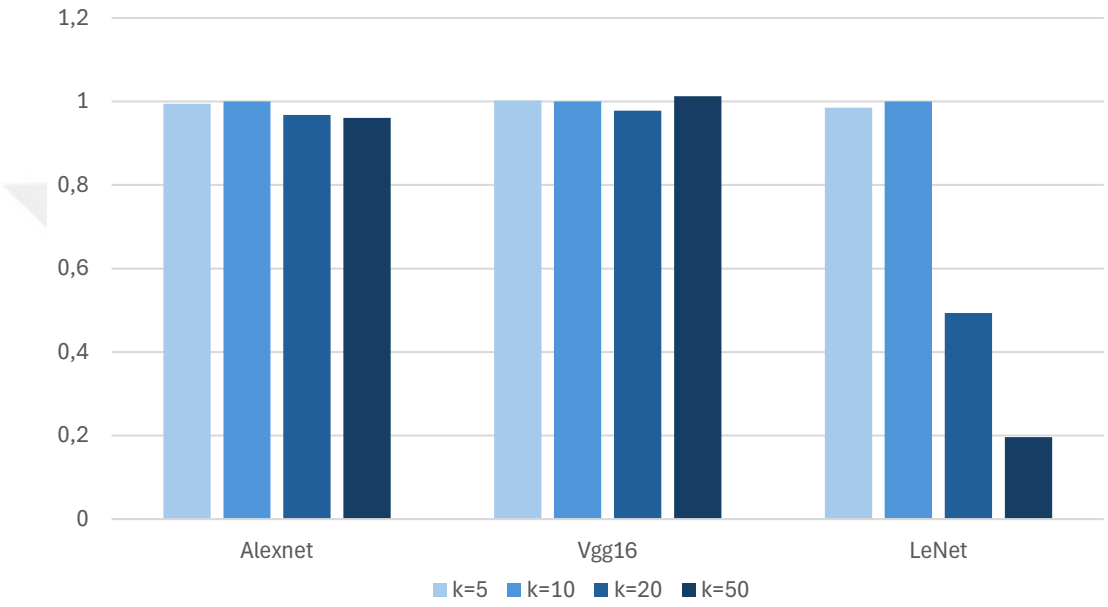

Figure 7.14: Efficiency of architectures with different array height (k) values. Results are normalized with respect to k=10.

We showed that WS⊎IS is the optimal choice of dataflow reconfigurability. Based on this architecture, some sensitivity analyses are made. As indicated, array height k is a customizable parameter set to 10 as the default value. For different numbers of k values, Table 7.2 shows the average convolution cycle in a network, area, and power values for corresponding architecture. As expected, the scaled k affects cycles, power, and area. For example, when k=10 is dropped to k=5, cycles nearly get doubled, area nearly get halved, and power nearly get halved. Similarly, when k=20, cycles nearly get halved, area nearly doubled, and power nearly doubled. This behavior shows the scalability of our architecture. The only exception occurs on LeNet. This can be explained by the kernel number for the first convolution of LeNet being just 6. Since 6 is smaller than 10, 20, and 50, latency would be the same for the first convolution. Smaller convolutions lead to less efficiency for larger architectures. Figure 7.14 shows efficiency scaled with k value. Expression 7.1 is used again for efficiency, and it is multiplied by k. It shows that results do not change considerably among different k values

Models	Average Latency (cycles)				Core Area (mm^2)				Power (mW)			
	k=5	k=10	k=20	k=50	k=5	k=10	k=20	k=50	k=5	k=10	k=20	k=50
AlexNet	347892	175883	89741	37198	1,02	2,01	4,09	10,18	34,09	68,08	135,45	342,84
VGG16	1722004	877015	442841	169835								
LeNet	1474	738	738	738								

Table 7.2: Average latency, area, and power results for different array height (k) values.

for AlexNet and VGG as expected—the difference in LeNet results from working with very small kernels. In general, scalability is preserved for larger kernels.

Another sensitivity analysis is made for Dataflow Selection Unit threshold, *thr*. Figure 7.15 shows the average excess latency of the Dataflow Selection Unit for different threshold values. Similar to Table 7.1, the excess latency of each convolution is noted and then averaged. Based on these results, 0.004 leads to a non-biased Dataflow Selection Unit. Below it, architecture tends to select IS; above it, OS is selected with a biased tendency. Therefore, 0.004 is the sweet spot for *thr*, where we observe results with the least excess latency.

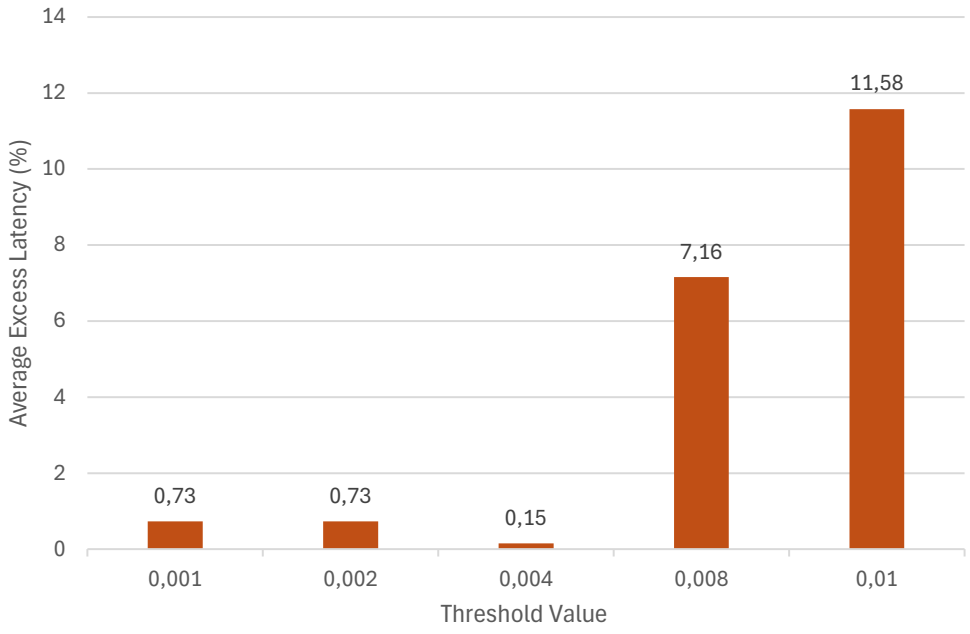

Figure 7.15: Average excess latency of the Dataflow Selection Unit for different threshold values.

Chapter 8

Conclusion and Future Work

This thesis proposes a reconfigurable architecture capable of working with both WS and IS dataflows. During in-depth analysis of dataflow reconfigurability, we showed that intermediate reconfigurabilities, which are mostly overlooked, should be considered in addition to fully reconfigurable (WS⊎IS⊎OS) and non-reconfigurable (WS, IS, OS) architectures.

This approach allows for more granular control over the data processing pipeline, enabling a finer balance between performance and resource utilization. By exploring and implementing intermediate reconfigurability, the architecture can adapt to varying computational loads more effectively, offering a more versatile solution than traditional, rigid configurations.

Our proposed accelerator architecture integrates critical components such as convolution engine, memory blocks, Collector Units, and PE arrays. It is designed to focus on reconfigurability to support selected dataflows by leveraging the similarities between WS and IS. The architecture efficiently handles the selection of dataflows through the Dataflow Selection Unit. The careful co-design of architecture and dataflows underscores the importance of selecting the most efficient dataflow for optimal performance. This approach provides a flexible yet high-performance solution to convolution operations.

While our architecture is 39% less area and 35% less power demanding option than fully reconfigurable architecture, it has also 3%, 57%, and 39% better performances compared to WS, IS, and OS options, respectively. It achieves a 7% improvement in efficiency over fully reconfigurable architecture, and its efficiency outperforms non-reconfigurable options by up to 3.57X. Besides its efficiency, its Dataflow Selection Unit works with an average of 0.15% excess latency for finding the fastest dataflow option. Moreover, our architecture can work with different kernel sizes by dynamically adjusting array blocks without losing efficiency, on top of dataflow reconfigurability.

As future work, our accelerator could be further enhanced by incorporating an additional Toeplitz matrix creation engine, which would help reduce the required data transfer size, and an activation function engine to offload more of the computational workload of CNNs onto the accelerator. Moreover, implementing application-specific reconfigurability can optimize the accelerator for various tasks. For instance, image segmentation models produce outputs in the form of images, whereas classification models give one-dimensional outputs. Analyzing the commonly used convolutions and their exploitable dimensions specific to these domains could lead to designing accelerators tailored to application-specific needs.

Bibliography

[1] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, p. 1–12, jun 2017.

[3] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA, p. 243–254, IEEE Press, 2016.

[4] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible

dataflow accelerator architecture for convolutional neural networks," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 553–564, 2017.

[5] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014.

[6] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *Proceedings of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 92–104, 2015.

[7] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," *CoRR*, vol. abs/1502.02551, 2015.

[8] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pp. 807–814, 2010.

[9] X.-P. Han, T.-S. Lee, and J.-H. Kim, "The influence of the sigmoid function parameters on the speed of backpropagation learning," *International Journal of Computer Mathematics*, vol. 55, no. 3-4, pp. 297–307, 1995.

[10] Y. A. LeCun, Y. Bengio, and G. Hinton, *Efficient BackProp*. Springer, 2012.

[11] R. M. Gray, *Toeplitz and circulant matrices: A review*, vol. 1 of *Foundations and Trends in Communications and Information Theory*. Hanover, MA: Now Publishers Inc., 2006.

[12] P. C. Treleaven, M. A. C. Pacheco, and M. M. B. R. Vellasco, "VLSI architectures for neural networks," *IEEE Micro*, vol. 9, no. 6, pp. 8–27, 1989.

[13] K. Asanovic, J. Beck, B. Kingsbury, P. Kohn, N. Morgan, and J. Wawrzynek, "SPERT: a VLIW/SIMD microprocessor for artificial neural network computations," in *Application Specific Array Processors, ASAP, Proceedings of the*

International Conference on, Berkeley, CA, USA, 4-7 August, pp. 178–190, IEEE, 1992.

[14] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan, "SPERT-II: A vector microprocessor system and its application to large problems in backpropagation training," in *Proceedings of the Advances in Neural Information Processing Systems 8, NIPS, Denver, CO, USA, November 27-30* (D. S. Touretzky, M. Mozer, and M. E. Hasselmo, eds.), pp. 619–625, MIT Press, 1995.

[15] J. Wawrzynek, K. Asanovic, B. Kingsbury, D. Johnson, J. Beck, and N. Morgan, "Spert-ii: A vector microprocessor system," *Computer*, vol. 29, no. 3, pp. 79–86, 1996.

[16] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," *47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[17] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," *SIGARCH Comput. Archit. News*, vol. 43, p. 369–381, mar 2015.

[18] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 109–116, 2011.

[19] L. Cavigelli and L. Benini, "Origami: A 803-gop/s/w convolutional network accelerator," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 11, pp. 2461–2475, 2017.

[20] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[21] H. Kwon, A. Samajdar, and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, (New York, NY, USA), p. 461–475, Association for Computing Machinery, 2018.

[22] G. S. Aujla, A. Singh, and N. Kumar, "Adaptflow: Adaptive flow forwarding scheme for software-defined industrial networks," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 5843–5851, 2020.

[23] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 696–701, 2014.

[24] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH computer architecture news*, vol. 45, no. 2, pp. 27–40, 2017.

[25] B. Kim, S. Li, and H. Li, "Inca: Input-stationary dataflow at outside-the-box thinking about deep learning accelerators," in *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 29–41, 2023.

[26] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 674–687, IEEE, 2018.

[27] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775, 2018.

[28] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[29] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, 2019.

[30] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, "Loom: Exploiting weight and activation precisions to accelerate convolutional neural networks," in *Proceedings of the 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2018.

[31] Z. Song, B. Fu, F. Wu, Z. Jiang, L. Jiang, N. Jing, and X. Liang, "Drq: Dynamic region-based quantization for deep neural network acceleration," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1010–1021, 2020.

[32] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Yodann: An ultra-low power convolutional neural network accelerator based on binary weights," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 236–241, 2016.

[33] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 688–698, 2018.

[34] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

[35] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–13, 2016.

[36] X. Hu, Y. Zeng, Z. Li, X. Zheng, S. Cai, and X. Xiong, "A resources-efficient configurable accelerator for deep convolutional neural networks," *IEEE Access*, vol. 7, pp. 72113–72124, 2019.

[37] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.

[38] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, (New York, NY, USA), p. 751–764, Association for Computing Machinery, 2017.

[39] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–39, 2016.

[40] A. Ren, Z. Li, C. Ding, Q. Qiu, Y. Wang, J. Li, X. Qian, and B. Yuan, "Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing," *SIGARCH Comput. Archit. News*, vol. 45, p. 405–418, apr 2017.

[41] S. Li, E. Hanson, X. Qian, H. H. Li, and Y. Chen, "Escalate: Boosting the efficiency of sparse cnn accelerator with kernel decomposition," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, (New York, NY, USA), p. 992–1004, Association for Computing Machinery, 2021.

[42] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[44] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[45] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9, 2015.

[46] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[47] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4700–4708, 2017.

[48] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the International Conference on Machine Learning*, pp. 6105–6114, PMLR, 2019.

[49] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Technical Report*, DAC, (New York, NY, USA), p. 1216–1225, Association for Computing Machinery, 2012.

[50] Cadence Design Systems, *Genus Synthesis Solution*. Available at: `https://www.cadence.com`. Accessed: August 2024.

[51] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," in *Hewlett Packard*, 2009.

[52] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of the 3rd International Conference on Learning Representations, ICLR San Diego, CA, USA, May 7-9, Conference Track Proceedings*, 2015.

[53] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[54] W. Luo, Y. Li, R. Urtasun, and R. Zemel, "Understanding the effective receptive field in deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 29, 2016.

[55] D. Moolchandani, A. Kumar, and S. R. Sarangi, "Accelerating cnn inference on asics: A survey," *Journal of Systems Architecture*, vol. 113, p. 101887, 2021.

[56] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on fpgas: A survey," *arXiv preprint arXiv:1806.01683*, 2018.

[57] Y. Hu, Y. Liu, and Z. Liu, "A survey on convolutional neural network accelerators: Gpu, fpga and asic," in *Proceedings of the 14th International Conference on Computer Research and Development (ICCRD)*, pp. 100–107, 2022.

[58] Y. Toyama, K. Yoshioka, K. Ban, S. Maya, A. Sai, and K. Onizuka, "An 8 bit 12.4 tops/w phase-domain mac circuit for energy-constrained deep learning accelerators," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 10, pp. 2730–2742, 2019.

[59] D. J. Pagliari, E. Macii, and M. Poncino, "Dynamic bit-width reconfiguration for energy-efficient deep learning hardware," in *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 1–6, 2018.

[60] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey of machine learning accelerators," in *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–12, IEEE, 2020.

[61] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009.

[62] A. Krizhevsky, "Learning multiple layers of features from tiny images," *Technical Report, University of Toronto*, 05 2012.

[63] S. Bose, A. De, and I. Chakrabarti, "Area-delay-power efficient vlsi architecture of fir filter for processing seismic signal," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 11, pp. 3451–3455, 2021.

[64] C.-T. Kuo and Y.-C. Wu, "Area-power-delay-efficient multi-modulus multiplier based on area-saving hard multiple generator using radix-8 booth-encoding scheme on field programmable gate array," *Electronics*, vol. 13, no. 2, 2024.

[65] M. Jung, Y. Jin, and M. Shihab, "Area, power and latency considerations of STT-MRAM to substitute for main memory," in *The Memory Forum*, 06 2014.