POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Politecnico
di Torino

Master's Degree Thesis

Designing and Evaluating Mapping of CNN layers on an edge-CGRA

Supervisors

Prof. Daniele Jahier PAGLIARI

Dr. Alessio BURRELLO

Prof. Maurizio MARTINA

Dr. Pasquale Davide SCHIAVONE

Eng. Juan SAPRIZA

Candidate

Nicolò CARPENTIERI

Academic Year 2023-2024

Abstract

Convolutions play a crucial role in image processing and computer vision. They are extensively used for tasks like image enhancement, filtering, and feature detection. Furthermore, convolution operations are the most common computing patterns in machine learning domains. Consequently, it is essential to efficiently implement convolution operations on hardware architectures to obtain superior performance when accelerating convolutional neural networks.

The primary aim of this thesis is to explore different convolution methods for adapting convolutional neural networks to Coarse-Grained Reconfigurable Arrays (CGRAs). CGRAs represent a departure from conventional computing architectures, offering enhanced flexibility and energy efficiency. In contrast to Application-Specific Integrated Circuits (ASICs), known for their efficiency but lack of flexibility, and Graphics Processing Units (GPUs), which are versatile but consume high power, CGRAs strike a balance by enabling instruction-level programming. This approach reduces the complexity and latency associated with configuring Field-Programmable Gate Arrays (FPGAs) at the bit level, leading to a harmonious blend of performance, space optimization, and energy efficiency.

CGRAs serve as energy-efficient and high-speed accelerators in IoT processors and embedded systems to enhance the performance of demanding computational operations. These architectures comprise a grid of Processing Elements (PEs) arranged in a two-dimensional layout. Connectivity among these elements enables seamless data transfer between adjacent PEs, thereby streamlining arithmetic computations, particularly accumulation, commonly found in convolution operations.

This thesis conducts a thorough assessment of software development approaches for Convolutional Neural Networks (CNNs), with a specific emphasis on enhancing energy efficiency and minimizing latency. The study compares the conventional direct convolution method with the IM2COL technique, which reorganizes input data into a columnar format to facilitate matrix multiplication-based convolution operations. While IM2COL has the potential to enhance computational efficiency, it also leads to increased memory demands due to data replication. To tackle, on edge devices, the energy consumption related to data movement, the thesis explores two stationary dataflow methods: weight and output stationary. The

weight stationary method aims to optimize weight reuse within PEs to reduce energy consumption, whereas the output stationary approach concentrates on mitigating the energy overhead associated with managing partial sums by keeping them localized to the register file (RF) of the PEs. Additionally, the thesis exploits three types of parallelism to boost throughput and diminish latency: parallelism in output channels, parallelism in input channels, and parallelism in filter spatial dimensions. The former is dependent on output stationary, while the latter is based on weight stationary.

The findings of this study indicated that the most effective approach is the parallelization of the filter, utilizing weight stationary for optimizing both energy efficiency and latency. Coupling the CGRA accelerator with our optimized weight-stationary kernels, we achieve a performance of 0.8 $^{MAC}/_{cycle}$, 12.5× better than the ones of the baseline RISC-V processor. In terms of memory, our approach consumes as low as 34 µJ, 3.3× lower than the RISC-V processor.

Acknowledgements

I express my profound gratitude to all my mentors for their constant encouragement and support throughout my thesis experience. Special thanks go to Prof. Daniele Jahier Pagliari, Dr. Alessio Burrello, Prof. Maurizio Martina, Dr. Pasquale Davide Schiavone, and Eng. Juan Sapriza for their invaluable mentorship and guidance. Their profound technical knowledge and outstanding personal qualities have made a lasting impression on me. I also express my sincere thanks to them for their patience and insightful advice, which were instrumental in each phase of my thesis work. This valuable research project would not have been possible without their exceptional assistance. I am also thankful for their generosity in supplying me with the necessary equipment and tools vital for the successful completion of my research.

I would also like to extend my heartfelt thanks to my family and friends, whose support has been invaluable. Consentitemi di esprimere questi sentimenti con le parole più sincere: vorrei ringraziare la mia famiglia, mia mamma, mio papà, mio fratello. Vorrei ringraziare mamma perchè è e continuerà ad essere il faro della mia vita, quella calda luce che anche nei momenti più freddi e bui sa come guidarti; vorrei ringraziare mio padre, per tutti i sacrifici che ha fatto, e perchè se ho intrapreso questo percorso universitario, lo devo anche a lui; vorrei ringraziare mio fratello, perchè nonostante la distanza che ci separa, continuerai ad essere la mia principale fonte di ispirazione. Vorrei ringraziare mio nonno, per la sua energia e voglia di vivere che mi hai e mi stai continuando a trasmettere. Voglio ringraziare zio Giancarlo, zia Angela e Chiara, la mia seconda famiglia.

Last but not least, I want to thank me. To the day in which I learned how to read, an important pillar of my life. To the people who have contributed, in badness and goodness, to make me the person who I am today. To my past and future failures, where I have built and I will build myself. To my feelings, which remember us how much we are fragile but at the same time they remind us that we are human being, and we gather our strength from them.

Sapere aude.

Recognize that there will always be rocks in the road ahead of you. You have to choose if those rocks are stumbling blocks that stop you on your journey or stepping stones that create a new path. – Friedrich Nietzsche

Table of Contents

List of Tables

List of Figures

Acronyms

ALAP
As-Late-As-Possible

AI
Artificial Intelligence

ASAP
As-Soon-As-Possible

ALU
Arithmetic-Logic Unit

ASIC
Application Specific Integrated Circuit

CGRA
Coarse-Grain Reconfigurable Array

CNN
Convolutional Neural Network

CIL
Compute-Intensive Loop

CPU
Central Processing Unit

DFG
Data Flow Graph

DRAM

Dynamic Random Access Memory

DNN

Deep Neural Network

DMA

Direct Memory Access

FPGA

Field Programmable Gate Array

FU

Functional Unit

GPU

Graphics Processing Unit

ISA

Instruction Set Architecture

IoT

Internet of Thing

Im2col

Image-to-Column

IP

Input-Channel Parallelism

MAC

Multiply-and-Accumulate

ML

Machine Learning

OP

Output-Channel Parallelism

OS

Output Stationary

PC

Program Counter

PE

Processing Element

RF

Register File

RTL

Register Transfer Level

SoA

State of the Art

U

Utilization

WS

Weight Stationary

WP

Weight Parallelism

Chapter 1

Introduction

CNNs are currently widely used in various modern applications of AI [1]. Ever since their groundbreaking application in image recognition [2], CNNs have been utilized in a plethora of fields, including self-driving cars [3], cancer detection [4], and playing complex games [5]. In various fields, CNNs have achieved performance levels that exceed human accuracy. However, this enhanced accuracy comes with a trade-off: high computational complexity. Historically, general-purpose compute engines, particularly GPUs, have been the primary choice for CNN processing [6]. However, as we approach the end of Moore's law, there is a growing awareness that specialized hardware is essential to continuously improve computing performance while maintaining energy efficiency [7]. It is essential to note that in addition to providing specialized hardware, optimizing the software design for convolutional operations can also play a crucial role in further improving both accuracy and energy efficiency [8]. These software optimizations encompass various techniques such as algorithmic improvements, parallelization strategies, and hardware-software co-design, all of which can significantly contribute to achieving better overall system performance in CNN based applications. Today, applications belonging to the IoTs, and high-end embedded system domains are becoming the main targets for CNNs [9, 10]; as a consequence, speed, energy efficiency, and scalability are of high importance, while general-purpose processors (i.e., CPUs and low-end GPUs) tend to become unsuitable for these tasks. FPGA based accelerators have begun to take their place as versatile solutions [11], as well as ASIC implementations, providing significant performance gains for CNNs [12]. In addition to the already listed solutions, another reconfigurable architecture, known as Coarse Grained Reconfigurable Architecture (CGRA), is being increasingly used as accelerators for machine learning algorithms [13]. CGRAs are arrays of PEs that are connected in a 2-D network, linear, or hierarchical style. Each PE consists of a FU and a RF. FUs of PE can be multipliers, adders, shifters, and other logical operations [14]. When comparing CGRAs with other hardware accelerators, it can be stated

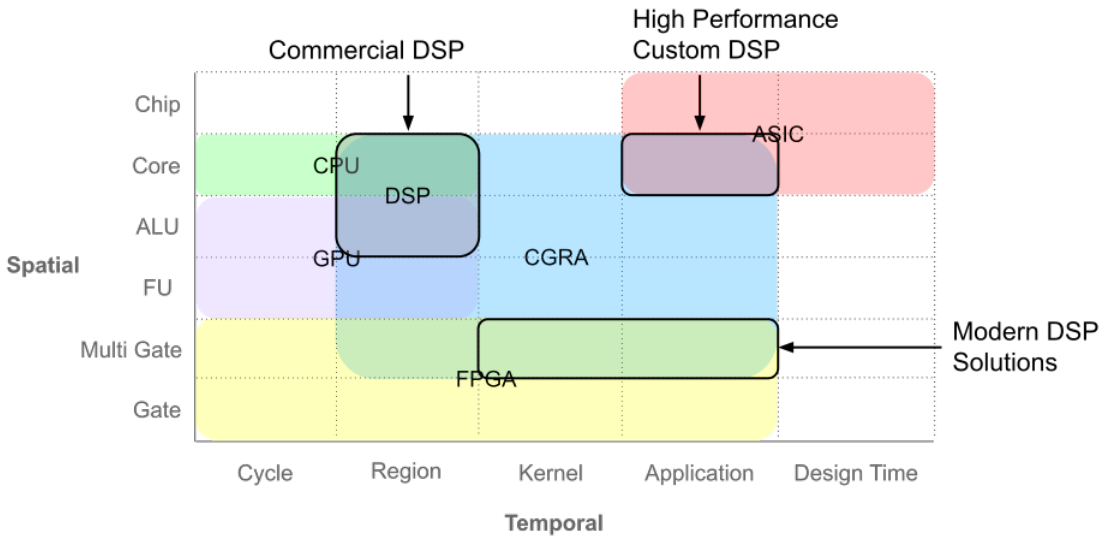


Figure 1.1: Architecture comparison, taken from [17]

that while ASICs accelerators are more performant than CGRA accelerators, they suffer from limited usability and programmability; using GPUs as accelerators could provide more benefits, but they are often limited to parallel loops, and, moreover, they have a large area and power overhead; ultimately, FPGAs are reconfigurable and general-purpose devices, but they are power hungry and have a lot of reconfiguration latency due to the large configuration bitstream [13]. CGRAs unlike FPGAs are normally programmed at the instruction level, whereas FPGAs are programmed at the bit level. As mentioned in [14] and [15], CGRAs represent a trade-off between spatial and temporal architectures. Spatial architectures, such as FPGAs and ASICs, execute many computational tasks simultaneously using a wide array of configurable logic blocks or custom-designed circuits. In contrast, temporal architectures, such as CPUs and GPUs, process instructions sequentially over time, optimizing for a series of operations to be executed by the same computing resources. CGRAs leverage the flexibility of spatial architectures with the efficiency of temporal processing, offering a balanced approach for applications that require both parallel processing capabilities and adaptability, as also shown in Figure 1.1. Research on CGRAs and their applications was in the silent phase until recent years, when they were used to speed up processor performance [16]. Numerous studies have already been carried out on harnessing CGRAs architectures to enhance the performance of CNNs [18, 19]. This thesis addresses specifically the problem of mapping convolutions, focusing on the OpenEdgeCGRA [20] architecture for edge computing applications.

The main goal is to outline efficient practices that allow to leverage this accelerator,

minimizing the impact of the overheads it imposes. For this reason, an investigation of various state-of-the-art computational and memory management strategies has done, aiming to uncover the most efficient mapping technique that balances performance and resource constraints. Specifically, a two-fold contribution is presented:

- Exploration of different implementation paradigms for convolution, different tensor parallelism axes and different memory organization;
- the results of the different implementations are benchmarked, measuring energy, latency, performance, and memory usage, and insights on the best mapping technique for low-power CGRA are provided.

This analysis highlights the predominance of direct convolution, coupled with weight parallelism, which reaches up to 3.4× and 12.5× in terms of energy and latency, respectively, compared to a plain CPU implementation, achieving an overall average performance of 0.8 $^{MAC}/_{cycle}$. In general, the thesis is organized as follows.

- Chapter 2 provides background on the context of DNNs and the most important hardware used for developing machine learning algorithms.
- Chapter 3 describes the specialized hardware used for the analysis of this thesis.
- Chapter 4 gives an overview of the CNNs. It also describes the various implementations used for this research.
- Chapter 5 focuses on visualizing and analyzing the most significant results achieved during this project.
- Chapter 6 is dedicated to providing concluding remarks and discussing future work.

Chapter 2

Background

2.1 Deep Neural Networks

DNNs mark a significant milestone in the evolution of Artificial Intelligence and Machine Learning, drawing inspiration from the architecture and capabilities of the human brain [21]. They are particularly adept at capturing complex patterns and addressing challenging problems. This thesis will explore CNNs, focusing on their effectiveness in being efficiently integrated with edge-based CGRA.

2.1.1 The Basic of Neural Networks

A neural network consists of interconnected nodes, referred to as artificial neurons or perceptrons, which collaborate to process information to make predictions or decisions. At the core of this system is the neuron, which plays a crucial role in receiving inputs, combining them through weighted sums, applying an activation function, and generating an output, as illustrated in Figure 2.1. The process of activations and adjusting weights is fundamental for the learning and adaptive capabilities of the neural network, distinguishing it from other models. Initially, neural networks, then called shallow neural networks, could handle a limited set of tasks but struggled with complex problems due to their simplistic structure with only a few hidden layers. This constraint hindered their ability to capture intricate patterns in the data. In contrast, DNNs feature multiple hidden layers that assist in processing data from input to output. These additional layers enable DNNs to learn and represent data hierarchically, capturing both fundamental details and complex abstractions. This ability to extract features at different levels is crucial for their effectiveness in addressing complex, high-dimensional data challenges.

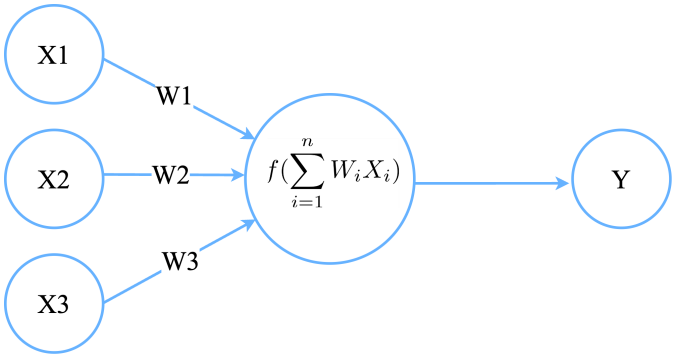


Figure 2.1: Schematic of a Neural Network Node

2.1.2 Convolutional Neural Networks

CNNs exploit the inherent structural information found in the data they analyze [22]. This is especially notable in image data, which exhibit repetitive patterns, elements that combine to form a whole, local characteristics, and self-replication within the same context. CNNs take advantage of these patterns as a fundamental assumption, capitalizing on the fact that these patterns remain consistent even with shifts—changes in the data's position do not affect its significance. This quality makes CNNs particularly efficient for tasks involving image data, as they can identify and leverage these shared traits to enhance their performance. CNNs are adept at managing image and video processing tasks due to their distinctive design, which enables them to excel in tasks such as object recognition, image classification, and segmentation. A typical CNN comprises convolutional layers that extract and handle features from the input data, as well as pooling layers that condense the dimensionality of these features to a more manageable level. This process aids in preserving crucial information while notably reducing the computational burden. Among various pooling methods, max pooling is the most common [23]. With max pooling, the top value from each segment of the feature map is preserved, ensuring that the most significant features are highlighted while minimizing the overall data volume.

2.1.3 Convolutional layers

The core component of a CNN is its convolutional layers, which are located within the hidden layers of the network and perform the essential function of convolving the input with a feature map. This operation is remarkable due to its resemblance to how the visual cortex processes stimuli [24]. In contrast, fully connected layers, although straightforward and capable of learning diverse features, encounter significant obstacles, notably their inability to inherently detect patterns

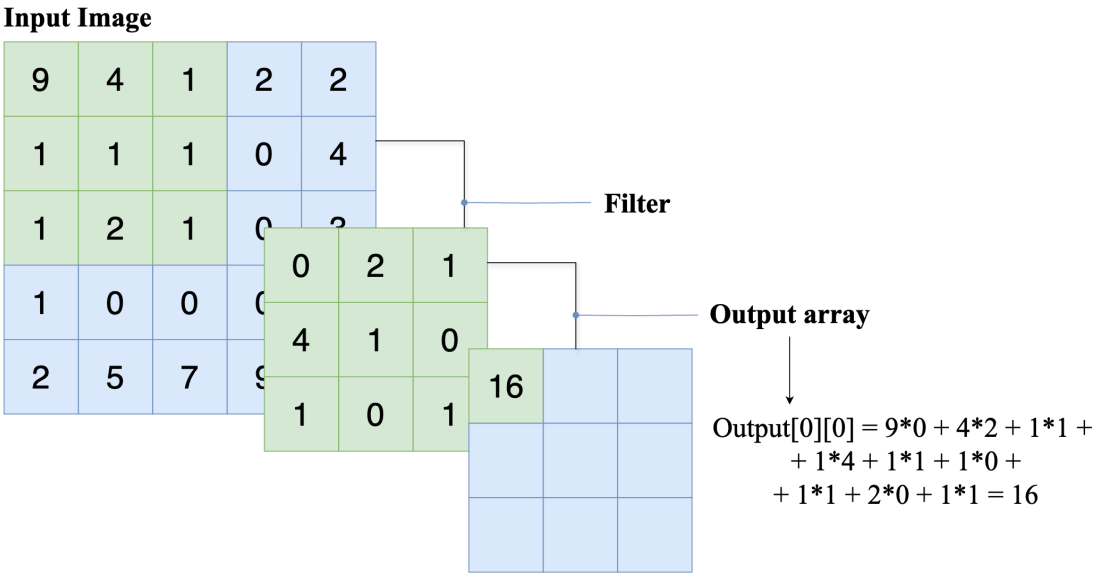


Figure 2.2: Convolution operation 1 layer

across different positions because of the absence of translation invariance. This limitation reduces their effectiveness for tasks that require recognizing patterns in various positions. Additionally, fully connected layers are burdened with a high number of parameters, resulting in elevated computational requirements and inefficiencies. Conversely, contemporary image processing applications increasingly depend on convolutional operations. Convolutional layers naturally demonstrate shift invariance, enabling more efficient and optimized pattern recognition. This characteristic makes them particularly well-suited for image processing tasks, where the capacity to identify patterns regardless of their location within the image is crucial for achieving high accuracy and efficiency. The convolution operation is characterized by the following formula:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

This process involves carefully moving a feature map over the input data incrementally, conducting multiplicative operations, as demonstrated in Figure 2.2 to illustrate a single convolution step. This method differs from fully connected layers as it significantly reduces the number of parameters by implementing parameter sharing. In contrast to fully connected networks, where each neuron connects to all neurons in the previous layer, convolutional layers utilize a set of adaptable filters. These filters are systematically applied across the input data, allowing the network to efficiently learn and detect features with fewer parameters, thus

improving computational efficiency and reducing the model's complexity. Convolution operations enable CNNs to extract and recognize these features from images, beginning with fundamental elements such as edges and textures and progressing to more intricate patterns. As the data traverses through the network, subsequent layers apply convolutions to these initial discoveries, progressively constructing more abstract representations. This hierarchical processing empowers CNNs to distinguish and identify complex patterns and objects within the data, enhancing their capacity to efficiently execute advanced image recognition tasks.
A convolutional layer is mainly composed by 2 parts:

- **Convolution**: the main component of a layer,
- **Activation Function**: an activation function is a simple function used to introduce non-linearities to the layer. It allows us to model more complex structures and data; typically this function usually can range from a simple ReLU, to a sigmoid, up to a Leaky ReLu.

In addition, to define the structure of the convolution, some paramaters must be set:

- **Stride**: it defines how much does the filter moves through the input at each step: with a stride 1 the filter moves one input data at a time, for stride 2 it is pretty much as if the filter skips one iteration each time etc.
- **Padding**: in order to not lose information by the borders of the input, usually a layer of padding is added: it is necessary to converge around the borders the input with 0 values. This parameter also avoids for border values to be computed less times than interior ones.
- **Dilatation**: it modifies the spacing between the elements of the convolutional kernel. In traditional convolutional operations, each element of the kernel is placed adjacent to each other. However, with dilation, there is a gap between the kernel elements.

In a convolutional layer, the complexity and functionality are influenced by several parameters, which are intricately related to the convolution's dimensionality. Focusing on the 2D convolution scenario, we delve into the structure of the three critical tensors involved: the output tensor, the weight tensor, and the input tensor. The process of 2D convolution transforms these interactions into a 3D space concerning the output tensor. This transformation introduces four key parameters to describe the output's dimensions. The first parameter, denoted O_X, corresponds to the width of the output, while the second parameter, O_Y, represents its height. The third dimension, denoted as K, signifies the number of output channels that the convolution operates across, highlighting the layer's capacity to handle multiple

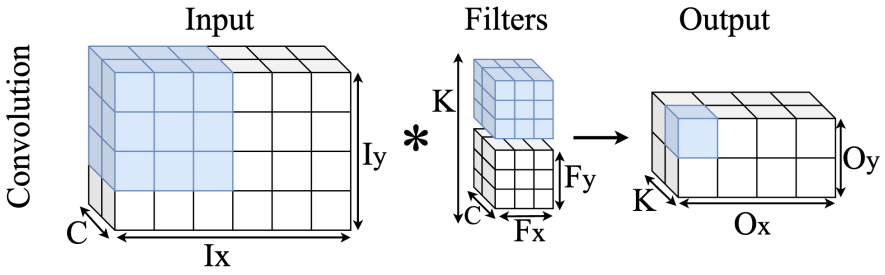


Figure 2.3: Convolution operation for the first output element.

features or filters simultaneously. The fourth dimension, B, which is optional, represents the batch size, indicating how many samples are processed together in one forward pass through the network. Similarly, the weights associated with a 2D convolution are organized along four dimensions: F_X and F_Y for the width and height of the filters, respectively, ensuring that each filter's spatial dimensions are accounted for. The dimension C synchronizes the number of channels in the weights with the input channels, facilitating the matching of feature detection across similar depth levels. The last dimension is K, representing the total number of filters. The input data are likewise structured in a four-dimensional format, with I_X and I_Y representing the width and height of the input image or feature map, ensuring that spatial information is preserved. The C dimension mirrors that of the weights, maintaining consistency in channel depth between the input and the filters applied. The B dimension, aligned with the weights and output tensors, organizes the data into batches for efficient processing. This four-dimensional approach to handling the convolution operation ensures a comprehensive framework for manipulating spatial and depth information throughout the network. An exaustive representation of the different dimensions is depicted in Figure 4.1. In this representation: $B = 1$, $I_x = 6, I_y = 4$, $F_x = F_y = 3$, $C = 2$, $K = 2$. Certain relationships among the dimensions are not completely independent, especially concerning the width and height of the input and the shape of the output, as well as the weights' shape. This interconnection becomes apparent during the convolution process, where a window of weights moves across a portion of the input to generate a single output cell. The input dimensions are thereby affected by the output and weights dimensions because of the operational mechanics of convolution. The stride and dilation parameters play a crucial role in altering how the weights move over the input, providing a key to understanding this relationship. The relationship between the input shape and the other two tensors shape can be then modeled through the following equations:

$$\begin{aligned} I_X &= StrideX * (O_X - 1) + DilatationX * (F_X - 1) + 1 \\ I_Y &= StrideY * (O_Y - 1) + DilatationY * (F_Y - 1) + 1 \end{aligned} \tag{2.1}$$

Therefore, when analyzing a 2D convolution, there are a grand total of 7 dimensions that are actually independent and needed. The computation of a single output cell is as follows:

$$O[k][oy][ox] = \sum_{c_i}^{C} \sum_{f_{yi}}^{F_Y} \sum_{f_{xi}}^{F_X} W[c_i][f_{yi}][f_{xi}] * I[c_i][o_y + f_{yi}][o_x + f_{xi}]$$

The following table recaps the list of parameters that are used for a 2D convolutional layer:

Acronym	Name
B	Number of batches
K	Number of output channels
C	Number of input channels
O_Y	Output height
O_X	Output width
I_Y	Input height
I_X	Input width
F_Y	Kernel height
F_X	Kernel width

Table 2.1: 2D convolutional layer parameters.

In conclusion, DNNs have represented a significant advancement in the field of AI and ML, showcasing exceptional abilities across a range of tasks. They are proficient at extracting complex data representations through multiple layers, leading to breakthroughs in various domains. Nonetheless, their implementation comes with its share of challenges. Issues such as the risk of memorizing training data (overfitting), the substantial computational resources required, and the extensive need for labeled datasets present significant barriers to their practical use. Innovative approaches like transfer learning, which adapts pre-trained models for new tasks, data augmentation to artificially enlarge training datasets, and the design of novel network structures provide avenues to overcome these obstacles. The evolution of DNNs is characterized by continual progress, fueled by ongoing research and innovation. As the field moves forward, it is expected that addressing current challenges will result in even more resilient and adaptable applications of deep learning, further establishing the pivotal role of DNNs in reshaping the landscape of AI and ML.

2.2 Hardware Acceleration of CNN

The acceleration of CNNs in various applications has been rapid since the start of the previous decade. This progress was made possible by the introduction of high-speed GPUs that could meet the substantial memory bandwidth and computational demands arising from the growing scale of CNNs. The deep learning solutions not only limit their applications to heavy computing machines, but they also pave the way for innovation in environments where computing resources are more constrained. However, there is also an increasing interest in utilizing CNNs on edge devices that have restricted hardware resources and energy. The hardware solutions for CNNs development and deployment range from general purpose architectures (CPUs and GPUs) to spatial architectures (FPGA and ASIC). The layer parallelization over the different hyper-parameter plays a crucial role in both fully connected and convolution layers, allowing for easy parallelization to enhance inference speed. Hardware acceleration can take the form of traditional hardware optimizations with increased computational parallelism or modern accelerators that integrate both hardware and software design capabilities. The current trend in efficient CNN applications involves CNN acceleration through a combination of hardware and software co-design. The subsequent sections will delve into a comparison of various architectures, with a specific focus on distinguishing between temporal architectures (such as CPUs and GPUs) and spatial architectures (e.g., ASICs and FPGAs), exploring the range of available hardware architectures suitable for CNNs. Spatial architectures perform multiple computational tasks at once by leveraging a broad network of configurable logic blocks or bespoke circuits. Conversely, temporal architectures handle instructions in a sequential manner, aiming to sequentially execute a series of operations with the same set of computing resources. This thesis explores particularly the CGRA architecture, which embodies characteristics of both spatial and temporal architectures. By combining the adaptability of spatial architectures with the sequential efficiency of temporal processing, CGRAs provide a versatile solution for applications demanding parallel processing power as well as flexibility.

2.2.1 Temporal Hardware Architecture

In general, CPUs and GPUs typically utilize temporal architectures characterized by a significant number of ALUs that act as processing units under centralized control. In this temporal architecture framework, ALUs do not have individual local memory and cannot communicate directly with each other. To achieve parallelism, CPUs follow the single-instruction multiple-data (SIMD) model, while GPUs adhere to the single-instruction multiple-thread (SIMT) execution model. Within these temporal platforms, convolution layers in CNNs are often carried out using matrix

multiplication. Enhancing the efficiency of matrix multiplication can be supported by software libraries such as OpenBLAS and Intel MKL for CPUs, and cuBLAS and cuDNN for GPUs. Using GPUs as accelerators could give more benefits but often limited to parallel loops, and moreover, they have a large area and power overhead [13]. Specialized hardware architectures designed specifically for CNN applications are less common in these architectures due to their general-purpose nature and adaptability to a wide range of applications.

2.2.2 Spatial Hardware Architecture

In general, designs based on ASIC and FPGA typically employ spatial architectures. Within spatial architectures, ALUs can have their own dedicated local memory and control logic, referred to as PEs. These PEs are interconnected in a processing chain, facilitating data exchange and direct communication among themselves, unlike ALUs in temporal architectures. For accelerators utilizing spatial architecture on ASIC or FPGA platforms, the primary bottleneck arises from memory access. To mitigate this, an array of PEs with small local buffers and a global buffer is utilized to minimize data retrieval from DRAM. Moreover, energy consumption in such architectures is predominantly driven by data movement between memory and processing elements due to the constrained on-chip memory and data transfer bandwidth. Various initiatives have been undertaken to address this challenge by leveraging emerging memory technologies like Dynamic Random Access Memory (DRAM), Resistive Random Access Memory (ReRAM), and Hybrid Memory Cube (HMC) to enable the direct integration of the processing engine and memory storage, known as Processing-in-Memory (PIM). The PIM approach aims to reduce data movement by executing certain computations within the memory itself, thereby mitigating the performance penalties associated with memory accesses. Noteworthy studies employing DRAM, ReRAM, and HMC-based PIM architectures for accelerating DNNs include DrAcc [25], PRIME [26], PattPIM [27], and Neurocube [28]. Subsequent subsections provide a detailed analysis of each spatial architecture.

Application Specific Integrated Circuit (ASIC)

An Application Specific Integrated Circuit (ASIC) is a type of integrated circuit (IC) designed for a specific use or application rather than for general-purpose use. In contrast to adapting the deep learning algorithm to the CPU or GPU hardware structure for acceleration, the primary method for speeding up the convolution learning algorithm using ASIC involves tailoring dedicated hardware acceleration algorithms, such as specialized designs for convolutional neural network algorithms [29, 30, 31, 32]. ASICs are specifically customized to accelerate a particular algorithm or class of algorithms, resulting in typically effective acceleration and

low power consumption. However, this specialization also leads to limited reconfigurability and high development costs. ASIC designs can deliver optimal performance, prompting the integration of custom function modules into a system on chip (SoC) to create a heterogeneous computing architecture. Nonetheless, due to the rigid nature of ASIC designs, resources cannot be repurposed, limiting them to specific tasks.

Field Programmable Gate Array(FPGA)

A Field Programmable Gate Array (FPGA), is a versatile type of programmable digital chip that comprises a large array of gates that can be programmed and reconfigured as needed. In a modern FPGA, there is an array of programmable blocks, some of which are highly flexible. These blocks may include look-up tables for basic boolean logic operations, registers for temporary storage of data, and interconnecting resources. FPGAs are commonly used to enhance the performance of applications that require intensive computations, high throughput, low latency calculations, or have strict power constraints. The use of FPGA-based accelerators has been gaining attention from researchers due to their benefits such as excellent performance, high energy efficiency, rapid development cycles, and versatility [33, 34, 35]. However, FPGAs also have limitations such as low area efficiency, high power consumption, large configuration bit-streams, and lengthy reconfiguration times, attributed to their fine-grained logic cells and static reconstruction.

2.2.3 Coarse-grained reconfigurable architecture (CGRA)

Coarse-grained reconfigurable architectures (CGRAs) are a natural coarse-grained implementation of the concept of reconfigurable computing proposed in 1960s [36]. This architectural design emerged in the 1990s [37, 38] and has been rapidly evolving since the 2000s [39, 40]. CGRAs are increasingly attracting attention due to their near-ASIC energy efficiency and performance, coupled with post-fabrication programmability similar to software [41, 42, 43]. The detailed comparison illustrated in Figure 2.4 compares CGRAs with ASICs, FPGAs, GPUs, and CPUs based on their energy efficiency, flexibility, and performance. In the academic realm, CGRAs are considered strong competitors to traditional computing architectures, as demonstrated by the substantial amount of research showcased at prestigious conferences [44, 45] and the significant support from organizations such as the Defense Advanced Research Projects Agency (DARPA)[46]. Moreover, in the industry, CGRAs are gaining attention. For example, Samsung has integrated aCGRA accelerator into its 8K high-definition television (HDTV) and Exynos System-on-Chips (SoC) [47, 48]. PACT Inc. has successfully deployedCGRA intellectual property (IP) cores in the satellite payload of Astrium [49]. Intel initiated

a project to integrate CGRAs into its Xeon processor in 2016 [50]. Despite these commercial applications, CGRAs are more prevalent in academia than in industry due to the technology's ongoing maturation process. To start, an introduction of CGRAs as the fundamental concept of this research is presented. ACGRA is a computational framework characterized by the following attributes:

Flexibility in Specific Domains CGRAs exhibit a level of flexibility after manufacturing that lies between general-purpose and fixed-function devices. Their hardware can be configured by software at runtime, but their processing elements (PEs) are not as robust as those found in general-purpose processors (CPUs), and their interconnections are less intricate compared to FPGAs [51]. This design offers just the right amount of adaptability for particular domains. In contrast to the broad flexibility of general-purpose devices like FPGAs and CPUs, domain-specific flexibility customizes the hardware for specific applications and minimizes redundant resources. Consequently, within the targeted domain, CGRAs are typically 1–2 orders of magnitude more energy-efficient than FPGAs and over 2–3 orders of magnitude more energy-efficient than CPUs [52, 53, 54]. However, the advantage of CGRAs tends to diminish for general-purpose applications [43]. Therefore, the domain-specific flexibility emerges as a crucial factor contributing to the balance that CGRAs strike between energy efficiency and adaptability.

Integrating Spatial and Temporal Computation Spatial computation in CGRAs leverages parallel computing resources and data transfer channels for processing, while temporal computation utilizes time-multiplexing resources. Consequently, the configuration of a CGRA involves determining the spatial and temporal coordinates of each node and arc in the data flow graph (DFG), a task typically handled by compilers. The combination of spatial and temporal computation provides a more flexible and powerful structure for executing applications. Compared to architectures that solely support temporal computation, such as CPUs, CGRAs eliminate the need for expensive deep pipelines and centralized communication overhead by efficiently scheduling computations over time. This temporal computation allows for a dynamic reallocation of resources based on the computational needs, reducing idle times and enhancing overall processing efficiency. In contrast to architectures focused solely on spatial computation, like traditional FPGAs, programmable array logic (PAL) architectures, and ASICs, CGRAs enhance area efficiency by allowing a more flexible and dense arrangement of computational units. This spatial computation leverages the physical layout to perform multiple operations in parallel, maximizing the use of available silicon space. By combining both temporal and spatial computation approaches, CGRAs can dynamically reconfigure the spatial layout of computational units to adapt to different tasks over time. This fusion means that a CGRA can adjust its hardware configuration to match the specific requirements of a task, blending the efficiency of spatial architectures in performing parallel operations with the flexibility of

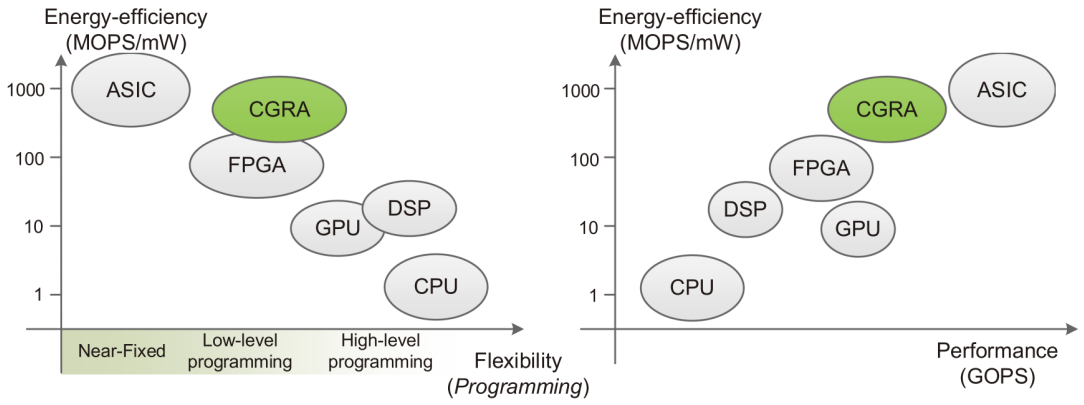


Figure 2.4: Architecture comparison in terms of flexibility, performance, and energy efficiency. Taken from [51].

temporal architectures in handling diverse computational workloads. Consequently, this dual approach significantly improves area efficiency by optimizing the utilization of the chip's real estate, ensuring that more computations can be carried out within a smaller footprint. Thus, CGRAs provide a versatile and efficient solution that leverages the best of both worlds, making them particularly suited for a wide range of applications where flexibility, performance, and area efficiency are critical. **Configuration or Data-Driven Execution** In contrast to processors that execute operations in a linear, control flow-oriented manner (as predefined by compilers), CGRAs primarily rely on configuration flow or data flow to drive their operations. The configuration of CGRAs specifies PE operations and interconnections. All PEs defined by a configuration operate synchronously under the same control flow (thread). Although configurations are influenced by control flow, the operations within each configuration run in parallel or in a pipelined fashion, leveraging compiler-driven parallelism.

Notably, configuration-based CGRAs can exploit effective explicit data flow through interconnections, a feature not supported by conventional instruction sets. A data-drivenCGRA is an instantiation of an explicit data flow machine [55], which entirely disregards control flow execution. Within a configuration, any operation with its operands ready for processing will be executed, offering candidates from all operations in that configuration. In comparison to control-flow or instruction-driven execution seen in multicore processors, configuration/data-driven execution can prevent overly serialized PE execution, harness fine-grained parallelism, and facilitate efficient synchronization among PEs. This execution approach also facilitates explicit data communication, reducing the energy overhead associated with data movement. Thus, configuration/data-driven execution stands out as a

pivotal factor contributing to the superior performance and energy efficiency of CGRAs.

Table 2.2: Comparison of different architectures, adapted from [51].

Architecture	Flexibility	Temporal	Spatial	Reconfig. time	Config. driven	Dataflow-driven
CGRA	Domain	✓	✓	ns-µs	✓	✓
FPGA	General	×(*a*)	✓	ms-s	✓	✓
ASIC	Fixed	×(*b*)	✓	×	×(*c*)	✓
Multicore	General	✓	✓	ns	×	×(*d*)

a FPGAs can perform temporal computation, but it is not practical considering the overhead and effectiveness.
b ASICs support hardware resource sharing as temporal computation to some extent.
c ASICs do not support reconfiguration, but there might exist configuration codes.
d Dataflow mechanisms can be supported in software at the task/thread level, e.g., data-triggered multi-threading.

In summary, CGRAs are characterized as specialized hardware that is flexible within a specific domain, enabling computation to occur spatially and temporally, with configuration flow or data flow directing the execution. This comprehensive definition, encompassing and refining previous definitions that have been contentious or one-sided [56, 17], eliminates any ambiguity. As indicated in Table 2.2, these three attributes set CGRAs apart from other computing architectures. While FPGAs share similarities with CGRAs in terms of adaptable spatial computing (reconfigurable computing), FPGAs differ as they are a more detailed, general-purpose flexible architecture and tend to lack support for temporal computation due to their slower reconfiguration process compared to CGRAs (nanoseconds versus milliseconds), making it challenging to pipeline reconfiguration with kernel-level computation. Although certain commercial FPGAs do offer runtime reconfiguration (RTR), the widespread applicability of RTR remains uncertain [57]. Multicore processors resemble CGRAs in structure, featuring multidimensional processing element arrays and message-passing interconnections. Nonetheless, their processing units consist of individual sequential cores driven by control flow or instructions.

Chapter 3

Related Works

In this chapter, we dive into the exploration of various platforms critical to advancing the capabilities of edge computing, focusing particularly on their role in improving real-time data processing in a variety of applications. The main emphasis is on leveraging heterogeneous architectures that combine the computational power of host processors with the specialized efficiency of custom accelerators. This approach aims to address the twin issues of performance and power efficiency that are common in edge-computing devices. This exploration is anchored in the detailed analysis of three distinct but interrelated platforms: the $H\mathcal{E}\mathcal{E}Psilon$ platform, X-HEEP, and OpenEdgeCGRA, each offering unique contributions towards achieving the twin goals of optimized performance and energy efficiency in edge computing environments.

3.1 H$\mathcal{E}\mathcal{E}$Psilon platform

The rapid growth of edge computing is driven by the increasing need for real-time data processing in various applications. Despite this progress, challenges persist due to performance and power efficiency limitations in edge-computing devices. To address these challenges, there has been a rise in heterogeneous architectures that combine host processors with specialized accelerators customized for specific applications. This approach leads to enhanced performance and reduced power consumption. This chapter focuses on describing the heterogeneous environment used, with particular emphasis on the accelerator. The heterogeneous architecture used in this research is known as $H\mathcal{E}\mathcal{E}Psilon$, comprising a RISC-V microcontroller (X-HEEP) and a CGRA architecture named OpenEdgeCGRA. The OpenEdgeCGRA is a low-power, general-purpose, scalable, instruction-based CGRA designed for executing healthcare applications efficiently with minimal area and power requirements. Although not optimized for running intensive kernels like CNNs, this

CGRA was selected for its open-source nature and validation in silicon. Figure 3.1 illustrates the *HEEPsilon* platform, which includes a CPU and memory allowing for real application evaluation.

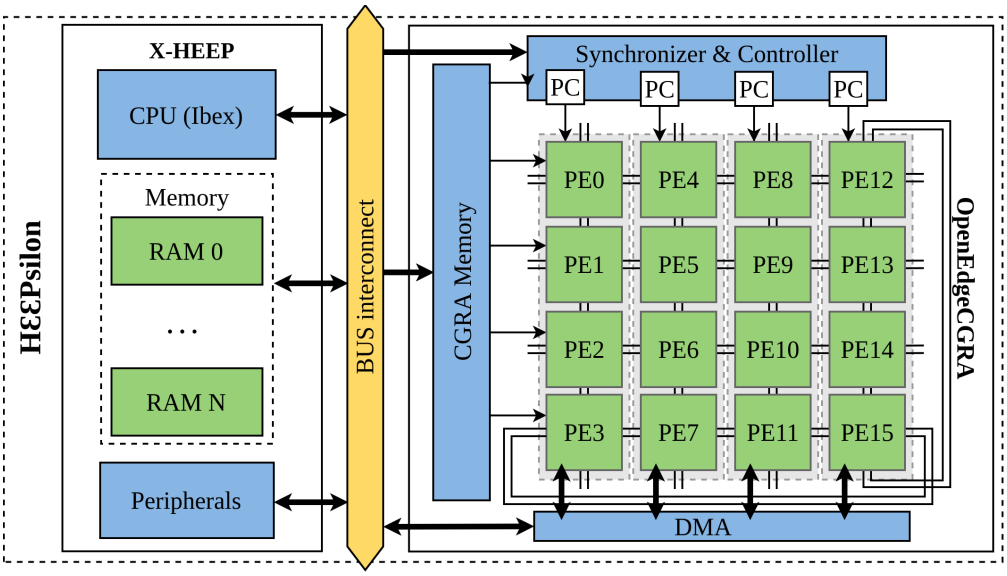


Figure 3.1: Architecture of the **HEEPsilon** platform used as a test bench for this analysis, where the OpenEdgeCGRA is instantiated along with X-HEEP.

3.2 X-HEEP

The eXtendible Heterogeneous Energy-Efficient Platform (X-HEEP) is positioned as an innovative open-source platform specifically designed to natively support the integration of ultra-low-power edge computing accelerators [58]. Through the implementation of power saving strategies, such as clock gating, X-HEEP emphasizes energy efficiency. These strategies are integrated with connected accelerators, e.g., OpenEdgeCGRA, through dedicated power control interfaces.
The platform is distinguished by its architecture, illustrated in Figure 3.2, which includes core components allowing extensive customization to meet specific application requirements. This customization is made possible by the exploration of various core types, bus topologies, and memory addressing modes, along with a granular configuration of memory banks to fit the constraints of integrated accelerators. X-HEEP's cores come from the OpenHW Group's CORE-V family, selected for their maturity and multiple successful implementations in silicon. In addition, the platform incorporates a wide range of IPs from the PULP project for the bus, memory models and debug unit, and peripherals from the OpenTitan project, all chosen for their proven reliability and comprehensive hardware abstraction

functions. X-HEEP further enriches its offerings with custom IPs such as boot ROM, power manager, a fast interrupt controller, and a DMA.

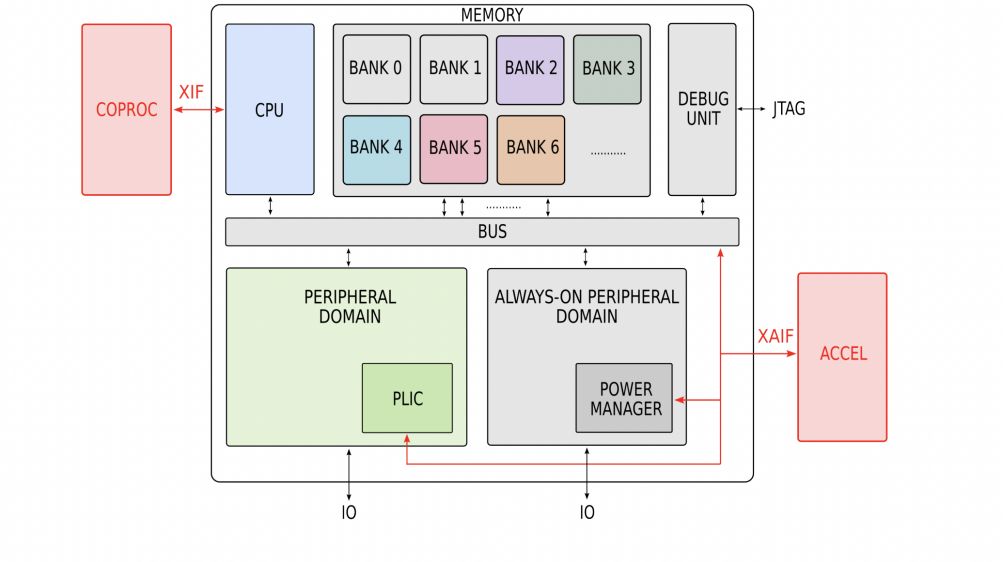


Figure 3.2: X-HEEP architecture, taken from [58].

Turning to X-HEEP's programmable modules, the platform exposes a configurable number of slave and master ports to the external XAIF interface, facilitating the integration of one or more accelerators. This configurability enhances connectivity with state-of-the-art accelerators and agile integration into microcontrollers for real-world applications. Slave ports, using the OBI protocol, provide easy access and configuration for memory-like accelerators, while master ports meet the bandwidth requirements of processor-like accelerators, such as OpenEdgeCGRA, which leverages four 32-bit master ports for read and write operations independent of main memory. An additional peripheral interface connected to X-HEEP's peripheral bus supports custom external peripherals, extended by a FIFO interface to facilitate simple DMA-peripheral connections, enabling efficient data transfers to main memory with DMA support.
For the purposes of this study, focusing on the usage of OpenEdgeCGRA, the selected configuration utilized CV32E20 as the core, employing the default bus configuration, a multi-master architecture (termed N-to-M). Additionally, the study explored two different scenarios of memory organization: initially employing 16 continuous memory banks of 32 kB each, followed by a transition to 16 interleaved memory banks of the same size. The contiguous mode offers limited bandwidth to applications, like CNN mapping, that require multiple masters to access contiguous data stored in memory, but allows for power-gating or setting in retention mode the banks that are not actively used. Conversely, the interleaved mode offers higher

bandwidth to applications that access contiguous data in memory, at the cost of keeping all the banks active all the time. On the contrary, the interleaved mode provides higher bandwidth for such applications, necessitating that all banks remain active. The study delves into latency and power evaluations for the contiguous scenario, with a focus on just the latency analysis for the interleaved mode.

3.3 OpenEdgeCGRA

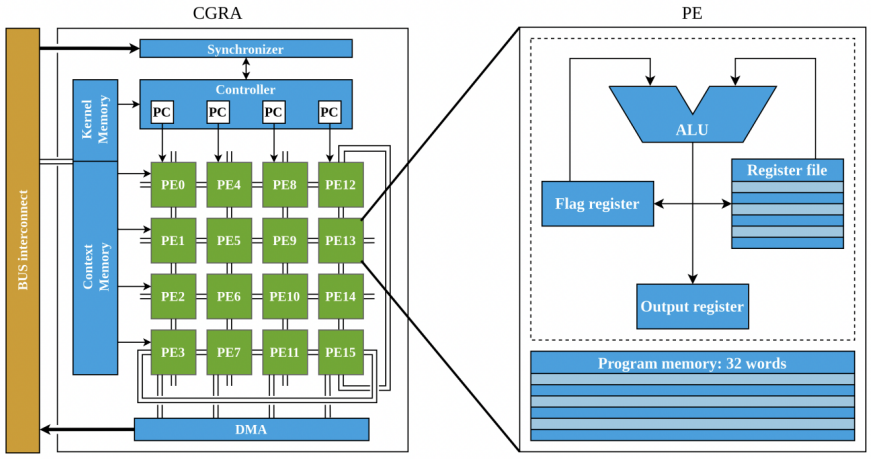


Figure 3.3: Top-level view of the CGRA architecture with a 4×4 PE array. In addition PE-level architectural view. Image taken from [59].

OpenEdgeCGRA is a CGRA design characterized by its open-hardware nature and low power consumption [20]. It can function as a memory-mapped accelerator within a system, as detailed in the preceding section. Complementing this CGRA setup is a C-based firmware library that facilitates the interaction between applications and the accelerator. To test and validate the entire system, simulation tools like Verilator, Questasim have been employed. The size of the reconfigurable array, including the number of cells and their arrangement, is adjustable to support design exploration based on the specific application field.

3.3.1 CGRA Architecture

A block scheme of the CGRA architecture is shown in Figure 3.3. The default setup used consists of a 4×4 grid of identical PEs that are interconnected with their four nearest neighbors in a torus arrangement, facilitating data movement within the reconfigurable mesh. Each PE comprises an ALU, two multiplexed inputs, an output register, a flag register that stores the 1-bit zero and sign flags from the

preceding instruction, a four-element register file, and a private program memory of 32 words. Operation is time-multiplexed, with each PE executing the instruction indicated by the column program counter (PC) at every clock cycle, allowing for the execution of modulo-scheduled loops. Additionally, each PE can utilize its internally stored values or the output from any of its four closest neighbors (top, left, bottom, or right). The proposed accelerator is directly linked to the main memory via four master ports (one for each column), managed by DMA.

Table 3.1: Instruction Set of OpenEdgeCGRA, taken from [20].

Type of operation	**Opcode**
Arithmetic operations	SADD, SSUB, SMUL
Shifts	SLT, SRT, SRA
Bit-wise operations	LAND, LOR, LXOR, LNAND, LNOR, LXNOR
Selects	BSFA, BZFA
Loads and stores	LWD, LWI, SWD, SWI
Conditional and unconditional branches	BEQ, BNE, BLT, BGE, JUMP
No operation	NOP
Finish	EXIT

Furthermore, simultaneous access to the data can occur if the system bus allows for multiple master-slave transactions to take place concurrently (specifically, in this study, an n-to-m configuration bus is utilized, as previously explained). Each column holds the addresses for read and write operations, which can be set up from the CPU. The CGRA instruction set enables the computation of a diverse range of kernels using 32-bit basic arithmetic and logical operations. The Arithmetic Logic Units (ALUs) are capable of executing arithmetic operations such as signed addition, subtraction, and multiplication, arithmetic and logic shifts, and bit-wise operations. The ALUs also support conditional operations based on the zero and sign flags (BXFA and BSFA, respectively), facilitating the implementation of branches through if-conversion. Nevertheless, a Multiply-and-Accumulate (MAC) instruction, which could enhance performance, is not included. Furthermore, both conditional and unconditional jumps are permitted, enabling the implementation of kernels with if statements and for loops. It is important to note that jumps are controlled by the hardware on a column-by-column basis, as multiple kernels can be mapped or executed simultaneously, each potentially requiring one or more columns. Input/output transfers are handled through direct and indirect loads and stores. Direct loads/stores access the memory location specified by the read and write addresses, with automatic incrementation. In contrast, indirect loads/stores encode the address within the instruction itself, which is then stored in a dedicated register.

Table 3.2: PE 32-bit word instruction format, taken from [20].

Field	**muxAsel**	**muxBsel**	**aluOp**	**rfSel**	**rfWe**	**muxFsel**	**imm**
Bits	31:28	27:24	23:18	17:16	15	14:12	11:0

The Instruction Set Architecture (ISA) supported by the CGRA is summarized in Table 3.1, while Table 3.2 illustrates the CGRA instruction word format, displaying the static fields for all instructions. Fields are associated with the multiplexers that govern the operations carried out in the ALU and the sources of operands. This setup eliminates the necessity for a decoder and enables the hardware to execute a single instruction per cycle for basic operations. The time taken for loads and stores in a cycle is dependent on the system bus, while multiplications are completed in three cycles. The CGRA includes a context memory of 2 KiB that contains the instructions defining the kernels that can be dynamically loaded into the PEs based on the application requirements. This memory is linked to the system bus and is programmed by the CPU. A CGRA synchronizer arranges the acceleration requests from the CPU on the available columns of PEs by duplicating and executing the instructions in the PEs, as elaborated in [60]. If a kernel request can be processed in the available columns at runtime, the instructions from the kernel are transferred from the context memory to the private memory of the corresponding PEs, and the execution begins. Otherwise, the synchronizer delays the request until sufficient resources are available. Consequently, kernels are assigned to specific columns within the PEs mesh at runtime rather than during compilation, resulting in a flexible utilization of resources. The synchronizer includes a group of 32 control slave registers to set up and initiate the execution of the kernel. Some of these registers also offer performance counters for measuring the performance and utilization of kernel execution. Lastly, the kernel configuration memory (comprising 15 elements) is employed to outline the kernels stored in the context memory. It specifies, for each kernel, the number of columns needed, the starting position of the instructions in the context memory, and the quantity of instructions that the kernel encodes and must be copied into the private memory of the PEs.

3.3.2 Assembler & Simulator

An assembler is included in the framework to convert descriptions of mapped operations into binary configuration words. This simplifies the programming process for controlling the execution of CGRA, as operations can be articulated in a human-readable format. The framework contains implementations of benchmark kernels, which are described as mapped operations in assembly language. These kernels are categorized into two groups: manually mapped kernels and kernels produced by the SAT-MapIt modulo scheduling compiler [59]. In this study, the

option to manually map kernels was selected, a choice made feasible also by the ability to emulate the CGRA used in HEEPsilon with CGRA instruction precision using the ESL-CGRA simulator. Additionally, as part of the CGRA companion software suite, there are tools available for generating test C code functions, enabling the execution of kernels from an application to validate the CGRA functionality and compare performance against software run-time on the host processor.

Chapter 4

Methods

This work explores different optimization directions to map a convolutional kernel onto the OpenEdgeCGRA. In particular, the implementation paradigm, i.e., the data layout coupled with its access order, is first explored. Then, the computation's parallelization over the CGRA's PEs is investigated. For all experiments, they are always considered convolutions with groups = 1, and a filter of dimension $F_X \times F_Y = 3 \times 3$ [61].

4.1 Convolution Mapping

4.1.1 Direct Convolution

The direct convolution's approach to processing, by avoiding data manipulation and instead fetching data straight from memory, ensures that the raw integrity of the input image is preserved. However, this method incurs significant overhead due to the non-sequential nature of data loading, necessitating complex addressing schemes to access the required data points efficiently. To minimize this overhead, a Channel-Height-Width (CHW) data layout is typically used [62]. Consequently,

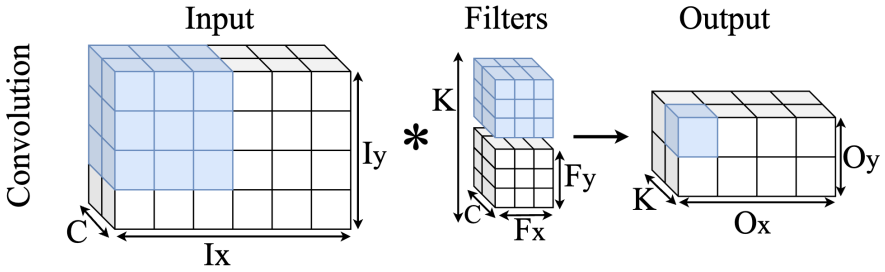


Figure 4.1: Convolution operation for the first output element.

while the simplicity of direct convolution in terms of not altering input data might seem advantageous, it demands a more sophisticated memory management strategy to mitigate the performance costs associated with its inherent data addressing challenges.

4.1.2 Im2col

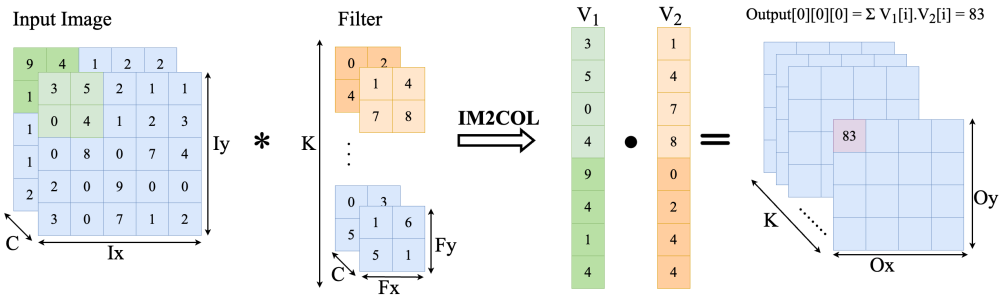


Figure 4.2: Im2col implementation, with $B = 1$, $I_X = I_Y = 5$, $F_X = F_Y = 2$, $C = 2$, $K = n$

On the other hand, the Im2col transformation is the most adopted implementation in CPU and GPU kernel libraries, such as PULP-NN [63], Mxnet [64], or Tensorflow [65]. It transforms multi-channel 2D Convolutions into a vector-matrix product by turning each input activations' patch (originally a 3D tensor) into a 1D vector of dimension input channels (C) $\times$ filter rows (F_X) $\times$ filter columns (F_Y), which is multiplied with the 2D weights matrix, of dimension C $\times$ F_X $\times$ F_Y $\times$ K (output channels); note that this transformation simplifies the memory accesses, which become sequential. The Im2col process for one input window is illustrated in Figure 4.2, showing the creation of two buffers, demonstrating the Im2col characteristic of arranging the selected elements contiguously in memory. As shown in the described figure, this process requires more memory to store the buffer of reordered inputs and additional instructions to create this buffer, which could be non-negligible. In this scenario, with this CGRA used, the Im2col transformation can leverage the loads with automatic index increment. In [62], the authors show that the Height-Width-Channel (HWC) data layout is the most advantageous for the creation of the Im2col reorder buffer. Hence, we select it for our implementation.

4.2 Parallelization axis

The computation in a CNN involves six nested loops (shown in algorithm 1), which can be swapped without changing the result [63].

Algorithm 1 Pseudo-code for convolution operation

```
for k in K (output channel) do
    for c in C (input channel) do
        for o_y in O_Y (output column) do
            for o_x in O_X (output row) do
                for f_y in F_Y (filter column) do
                    for f_x in F_X (filter row) do
                        mac_operation()
                    end for
                end for
            end for
        end for
    end for
end for
```

These loops, shown above, correspond to i) output channels (K), ii) input channels (C), iii) output columns (O_Y), iv) output rows (O_X), v) filter columns (F_Y), and vi) filter rows (F_X). In this study, the parallelization of the C, K, or $F_{x/y}$ loops is explored. There is no parallelization of the $O_{X/Y}$ loops, as it would allow reuse of neither the weights nor the inputs, caused by hardware resource limitations.

4.2.1 Weight Parallelism

This method leverages the parallelization of the filter loops, F_X and F_Y. In this setup, each weight element of a single input and output channel is assigned to a different PE.
For a 3×3 filter, this means that nine weight elements are distributed across nine PEs. Once these weights are retrieved from memory, the system performs multiple MAC operations by updating the inputs for each PE. The partial outputs generated by the PEs then move through the spatial array of the CGRA.
This procedure is illustrated in Figure 4.3. In addition to the nine PEs engaged in computations, the final row (comprising three PEs) is tasked with updating the address to load the new input triplet (3×1), while the other 3×2 inputs can be efficiently reused by shifting them from the first two rows of PEs when computing the next output pixel on the same output image row. During this stage, the last column of PEs aggregates the nine computed partial sums and, if necessary, adds

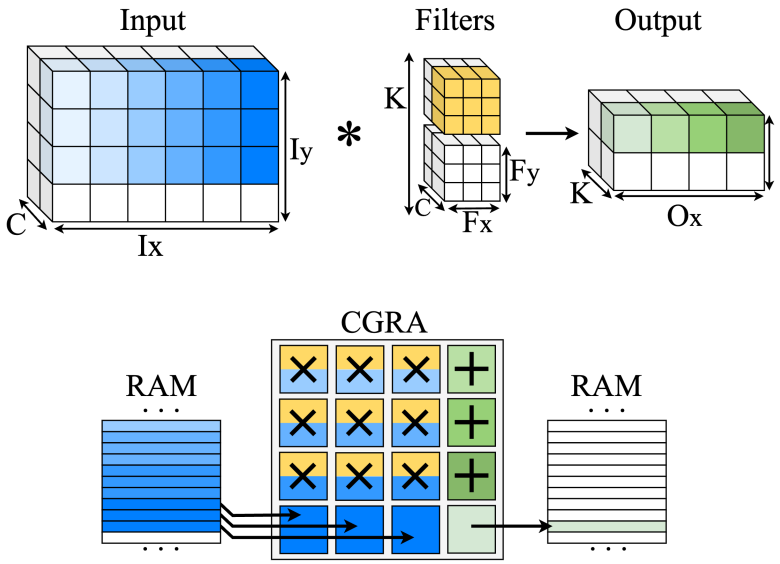


Figure 4.3: (Top) 2D convolution scheme. (Bottom) Direct convolution with weight parallelism. Nine PEs perform dot products. The other PEs load new inputs or sum partial outputs.

them to a previous partial sum when processing input channels $c_i > 0$. The last PE is designated for storing the accumulated partial sum in memory. This cycle is repeated for the entire input spatial position before a new set of weights is loaded to process the next input channel. The outputs are sequentially generated starting from O_X, O_Y, and, finally, K. Importantly, this mapping scheme, which benefits from a CHW input layout, would not benefit from using the Im2col transformation.

In order to conduct a thorough analysis, the following pseudo-code presents the binomial: convolution mapping and OpenEdgeCGRA. This approach involves analyzing one input channel each time the OpenEdgeCGRA function is invoked. Consequently, to complete the convolution process, the CGRA needs to be called $K \times C$ times, as shown in algorithm 2.

Algorithm 2 Call scheme of weight parallelism

```
for k in K (output channel) do
    for c in C (input channel) do
        cgra_call()
    end for
end for
```

4.2.2 Output Channel Parallelism

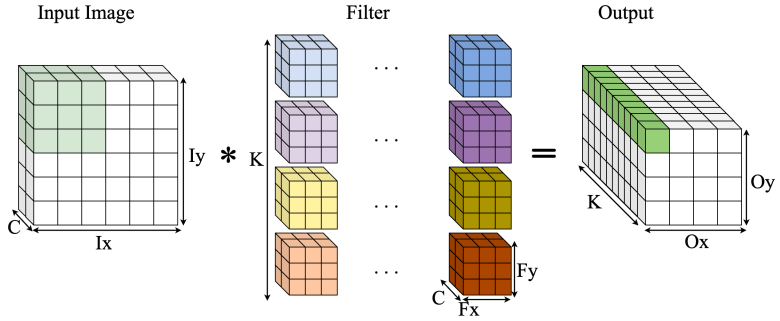
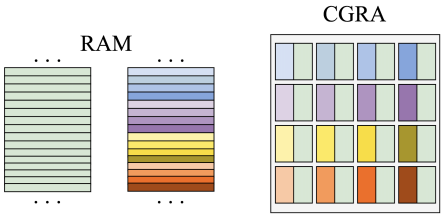
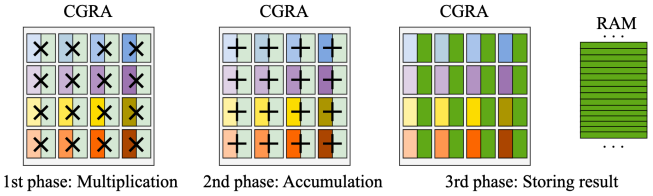


Figure 4.4: Convolution with output channel parallelism. Each K filter multiplies the selected input window, to obtain K different results. In this representation: $B = 1$, $I_X = I_Y = 6$, $F_X = F_Y = 3$, $C = 2$, $K = 16$

This mapping aims to produce results simultaneously for different output channels. This parallelism exploits the output stationary. This latter concept is based on minimize the latency and the energy consumption for reading and writing partial sums by keeping them in the RF of each PE [66]. The combination of both of them, allows to assign a different output channel to each different PE, which stores a different set of weights, and the same input elements are broadcast to all PEs, to produce 16 output channels at the same spatial location in parallel, as shown in Figure 4.4. A detailed illustration of this approach can be found in Figure 4.5, which shows the mapping of input and filter within the OpenEdgeCGRA (Figure 4.5(a), as well as the MAC operation for this specific setup. It is important to note that in this case, there is no opportunity for parallelizing the MAC operation, unlike weight parallelism, where only 9 PEs are dedicated to the MAC operation while the rest handle other tasks. In this scenario, each PEs is fully occupied: in the first case with the multiplication and in the other case with the accumulation. When the last input pixel of the selected window is processed in each PEs, all of them must store the partial sum, saved in their respective register files, as depicted in Figure 4.5. What is not shown in the image is the process related to fetching the new input. In each PEs' RF are contained both the address of last input and weight element fetched. This mapping leverages both direct convolution and the Im2cols technique for implementation. The Im2cols method streamlines the fetching of new input elements, as it reorders the input window such that the displacement for accessing a new input element remains consistent, specifically, an offset of 4. Direct convolution, by contrast, encounters challenges in such scenarios, especially when transitioning to the analysis of the subsequent input row. This necessitates a counter to manage these transitions effectively. Overall, the Im2col implementation reduces the instruction count required to achieve the same



(a) Register File Content



(b) The 3 different phases of the MAC instruction

Figure 4.5: Output channel parallism mapping onto OpenEdgeCGRA

functionality when compared to direct convolution. A preliminary comparison between this approach and weight parallelism suggests that this algorithm may exhibit lower latency performance, primarily due to the third phase where each PEs executes a *store* operation, whereas in weight parallelism, only one PEs is responsible for storing. Furthermore, while weight stationary employs a pipeline scheme (with 9 PEs processing new inputs, 3 accumulating previous results, 3 capturing new inputs, and the final one handling the *store* operation), here there is no opportunity for pipelining in this case since all PEs are fully utilized.

Direct Convolution

The provided pseudo-code, from algorithm 3, illustrates the parallelism mapping of the output, which is achieved through direct convolution. Similar to before, the OpenEdgeCGRA function is invoked within two nested loops. In this scenario, the convolution process is centered on the output channels. This implies that once the chosen input window traverses all the K output channels, a new input window must be selected to produce a distinct output element.

Algorithm 3 Call scheme of output parallelism (direct convolution)

```
for i in Ox (output row) do
    for j in Oy (input column) do
        cgra_call()
    end for
end for
```

Im2col

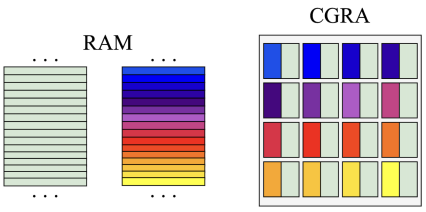
The implementation of Im2col is distinct from the previous one mainly due to the memory allocation for the input window position. This technique organizes the window in a way that its elements are placed contiguously in memory. The code's goal, illustrated in algorithm 4, is to optimize execution time by leveraging parallelism between the Im2col and `cgra_call()` functions. The Im2col function is called before entering the loop to pre-organize the first selected input window into a vector. This prepares the necessary data so that the CGRA can be executed immediately with the already organized input vector in memory. Subsequently, while the CGRA is performing its task, Im2col is invoked again to arrange the next input vector. This overlap of the executions of Im2col and `cgra_call()` aims to reduce downtime and increase overall efficiency.

Algorithm 4 Call scheme of output parallelism (Im2col)

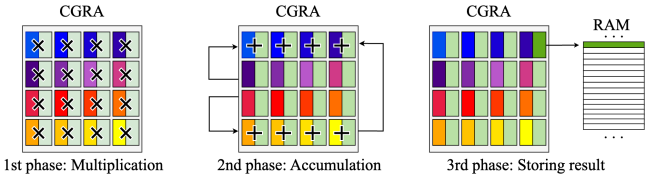
```
im2col()
for i in Ox (output row) do
    for j in Oy (input column) do
        cgra_call()
        im2col()
    end for
end for
```

4.2.3 Input Channel Parallelism

This method involves performing MAC operations relative to various input channels in parallel, as depicted in Figure 4.6. It utilizes the Im2col technique to enable sequential access to the input and filter data. Note that using direct convolution for this parallelism strategy would be suboptimal given that the latency to access the data from each single PE would strongly increase given their storage position. In this mapping strategy, for each iteration of the most external loops (K, O_X, O_Y), every PE handles a distinct set of input channels (C/16 per PE) for the same output channel and spatial position.



(a) Register File Content



(b) The 3 different phases of the MAC instruction

Figure 4.7: Input channel parallism mapping onto OpenEdgeCGRA

This scenario bears many similarities to the previous one, which involved output channel parallelism. Figure 4.7(a) illustrates the register file content for each processing element. The implementation of the MAC operation mirrors that of the output channel implementation. In this instance, each PE is responsible for multiplying the input pixel by the weight of the corresponding input channel, but the accumulation phase differs significantly. Here, to compute a single output, it is necessary to sum the various partial results stored within the register file of each PE. The evaluation of this process is depicted in the second phase of Figure 4.7. Ultimately, all partial sums are aggregated within a single PE, which is then tasked

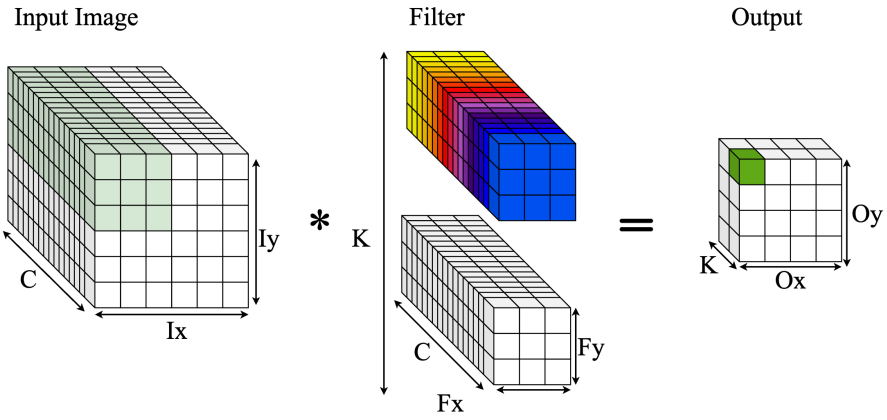


Figure 4.6: Convolution with input channel parallelism. Each PEs is assigned a different input channel C. In this representation: $B = 1$, $I_X = I_Y = 6$, $F_X = F_Y = 3$, $C = 16$, $K = 2$

with storing the computed output. Although there are multiple PEs available to potentially parallelize the MAC operation, the primary challenge lies in the storage phase. Unlike the output channel scenario where 16 different output results are stored, here only one output result is stored. Consequently, to produce 16 distinct elements as output, the OpenEdgeCGRA executing this algorithm is called more times compared to the output channel parallelism approach. An initial comparison among this approach and the two previous ones suggests that this method represents the least efficient algorithm of the three. The algorithm's inefficiency stems from the absence of a stationary method in the input channel parallelism approach. Unlike weight parallelism, which holds 9 distinct filter elements within the RF of each PEs, and output channel parallelism, where every PEs reuses the same partial result for processing new inputs, the input channel parallelism fails to exploit any data reuse, and in addition all 16 PEs are occupied to handle one `store` operation. This makes the least efficient algorithm among the three methods.

Algorithm 5 Call scheme of input parallelism

```
im2col()
for i in K (output channel) do
    for j in Ox (input column) do
        for h in Oy (output column) do
            cgra_call()
            im2col()
        end for
    end for
end for
```

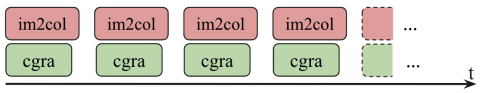


Figure 4.8: Im2col and CGRA function parallelized.

It is initially apparent that this implementation suffers from inefficiencies in both the convolution mapping strategy and the algorithmic execution. The primary source of this inefficiency stems from its operation: as it processes each input channel sequentially to produce a unique output element, it requires switching to a new input window and a new filter. Furthermore, Figure 4.8 uncovers a noteworthy observation: the principal factor affecting latency is the im2col function, as the *CGRA* hardware experiences less strain in comparison to im2col. This technique differs from other convolution methods by reducing one loop in the CGRA invocation process. While typical approaches use two nested loops with CGRA hardware, this specific convolution implementation decreases the number of loops, leading to a partial acceleration of the convolution process and somewhat reduced efficiency. This is because, despite more function calls resulting from the missing nested loop, it does not fully maximize the advantages of stationarity exploited by other methods.

Comparing this viewpoint with the traditional approach helps in understanding the intricacies of this implementation's performance.

4.3 Kernel Operation Mapping

The mapping of a Compute-Intensive Loop CIL such as a convolution onto a CGRA can be divided into three main parts: prologue, kernel and epilogue. The role of the prologue in setting up the initial conditions or configurations tasks, such as loading data into memory, initializing counters, or configuring the CGRA's processing elements. The epilogue phase focuses on the finalization steps, such as writing computed results back to memory. These instructions are executed only once at the beginning and end of the CGRA call respectively, while the kernel is the set of instructions that are executed iteratively, typically accounting for the largest portion of latency and resource utilization. It describes the types of operations it performs (e.g., arithmetic operations, logical operations) and its impact on the overall performance of the CGRA. An analysis of the kernel phase of each mapping strategy defined above is reported in following section, allowing for an initial estimation of their bottlenecks. It is noteworthy that each of the four implementations does not only consist of an inner kernel, but may also include one or more external loops. A summary of the various configurations can be found in Table 4.1.

Table 4.1: Overview of Mapping Strategies detailing the number of instructions (instr.) per stage. Numbers in parentheses indicate the number of cycles each loop executes.

	Conv-WP	**Im2col-IP**	**Im2col-OP**	**Conv-OP**
Prologue	2	2	2	2
External Loop (III)	×	×	×	6 ($\frac{K}{16}$)
External Loop (II)	×	×	×	5 (C)
External Loop (I)	5 ($O_Y * O_X$)[†]	×	6 ($\frac{K}{16}$)	5 (F_Y)
Inner Kernel	4 ($O_X - 1$)[†]	9 ($\frac{F_X \times F_Y \times C}{16}$)[‡]	9 ($F_X \times F_Y \times C$)[‡]	9 (F_X)[‡]
Epilogue	6	6	1	1

[†] See Figure 4.9 for the kernel associated with Weight Parallelism.
[‡] Refer to Figure 4.10 for the generic kernel mapping.

4.3.1 Weight Parallelism Mapping Analysis

The WP mapping is composed of a main internal loop and an external internal loop. The main loop is composed of only 4 instructions that allow the execution of

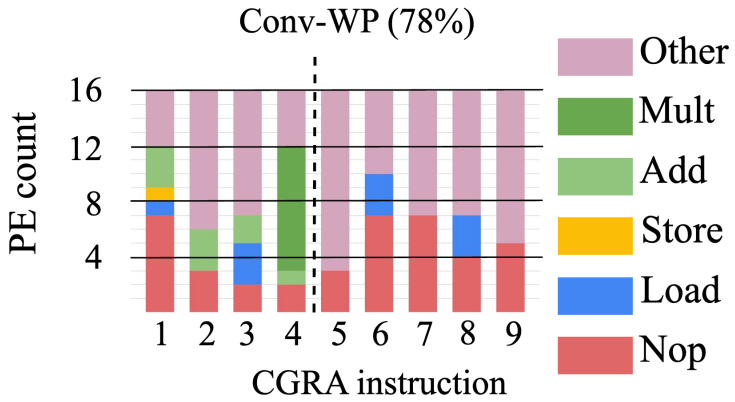


Figure 4.9: Operation distribution for weight parallelism. *Other* includes index updates, branch operations, and index manipulation.

the nine multiplications, the sum reduction, the load of a new input triplet, and the final store. However, as mentioned in section 4.2.1, once a new output row has to be processed, also the other 6 inputs (2×3) should change, necessitating 5 additional instructions (border loop) to load the additional data and update indexes, as shown in Figure 4.9. In this case, the main loop is executed $O_X \times O_Y \times C \times K$ times with an utilization of 78%, while the border one is executed only once per row, i.e., $O_Y \times C \times K$ times. To finalize the analysis, the prologue loads the input and weight memory addresses into the register file of the 9 chosen processing elements for executing the MAC operation. The 6 instructions in the epilogue phase aim to finish the final MAC operation.

4.3.2 Input channel Parallelism Mapping Analysis

Among the mapping strategies evaluated, the Im2col-IP (and weight parallelism methods) stand out due to their minimal instruction footprint. It has a total of 19 instructions, marking the lowest count when compared to the output channel parallelism implementations. It does involve one external loop primarily due to the utilization of the Im2col method, which reduces the number of instructions required to manage border scenarios, such as transitioning to the next output row, or next input channel, or output channel. As before, the prologue is in charge of loading the addresses of the input elements and filters, setting the stage for the subsequent convolution operation. In this case, 16 inputs and weights are loaded (corresponding to 16 input channels). Next, the `mul` and `sum` operations are executed by all PEs. Then, in the last 5 instructions, the input and weight addresses and the iteration counter are updated, followed by the loop's branch instruction. The iteration is repeated as soon as the selected input windows is

analyzed completely, as shown in Figure 4.10. Most PEs execute a `nop` during the last three instructions because only one to two PEs are in charge of updating the iteration counter and branching for the whole CGRA.
The epilogue is in charge of add the parial result stored inside all the PEs. It takes six cycles for adding them and store the result accumulated. This mapping strategy encounters efficiency challenges, primarily due to its low store-to-MAC ratio. Specifically, a single output result requires executing a number of instructions as determined by the following equation:

$$\frac{9 \times F_X \times F_Y \times C}{PE} \tag{4.1}$$

where:

- 9 represents the number of instructions for the inner kernel.
- F_X is the filter width.
- F_Y is the filter height.
- C is the number of input channels.
- PE represents the number of PEs, and is equivalent to 16.

Consequently, to complete the convolution, this algorithm must be invoked $K \times O_X \times O_Y$ times. This frequent invocation, driven by the sparse store operations amidst a high volume of MACs, underscores the strategy's inefficiency in resource utilization and in latency.

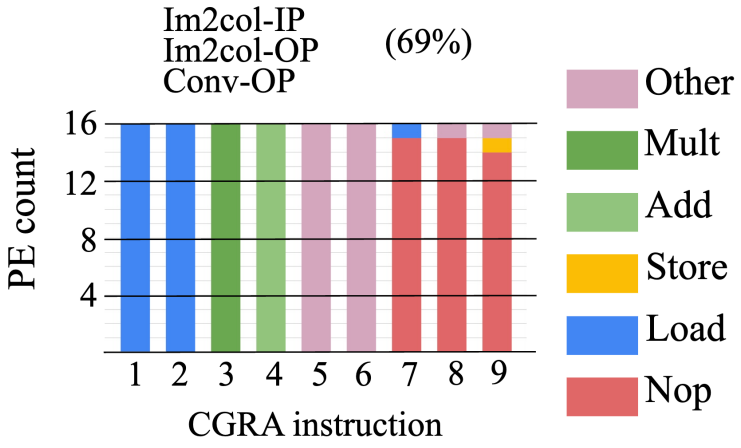
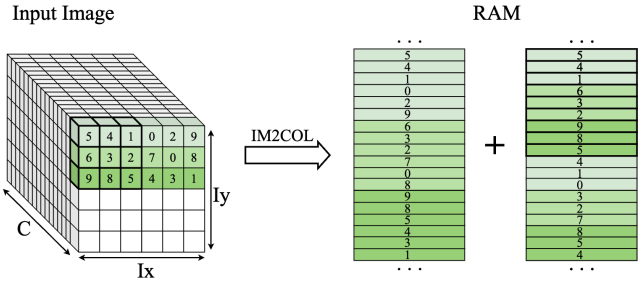


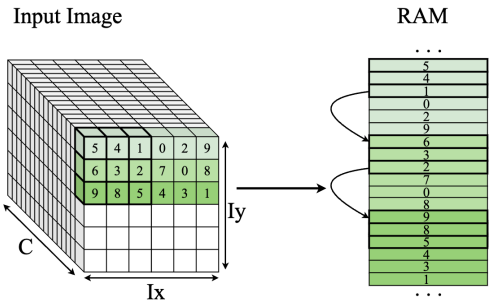
Figure 4.10: Operation distribution for the input and output channel parallelism . *Other* includes index updates, branch operations, and index manipulation.

4.3.3 Output channel Parallelism Mapping Analysis

The kernel of this mapping is exactly the same of the Im2col-IP (both of these techniques use as inner kernel an amount of 9 instructions, resulting with 69% of utilization), but its context is completely different. In addition, also its two implementations (one with the direct convolution, and the other by exploiting Im2col technique) also has a different context. By examining its kernel, the process remains consistent as before: initially, 16 inputs and weights are loaded in the first two steps, corresponding to 16 output channels. Subsequently, all processing elements perform the `mul` and `sum` operations. Following this, the input and weight addresses, along with the iteration counter, are modified in the final 5 steps, culminating in the loop's branch instruction. An in-depth analysis of the Im2col-IP method compared to direct convolution, especially in the context of output channel parallelism, uncovers significant differences. As shown in Table 4.1, the direct convolution method, known as Conv-OP, involves as many as four external loops, in contrast to the Im2col-OP method, which uses only two loops. This distinction is crucial, as illustrated in Figure 4.11(a), highlighting the operational differences between utilizing the Im2col-IP technique and choosing direct convolution. Regarding Im2col-IP, even though there is a larger memory usage because of the input buffer creation, organizing the input window in adjacent



(a) Im2col implementation



(b) Direct convolution implementation

Figure 4.11: Im2col effect on the selected input image

memory segments greatly simplifies the code implementation.
This organization eliminates the necessity for a position-tracking counter, making the entire process more straightforward. The contiguous memory allocation enables a simpler access pattern, thus improving computational efficiency.
On the contrary, as illustrated in Figure 4.11(b), the direct convolution method requires the use of two separate counters: one for input rows and another for input columns. This condition not only adds complexity to the code but also results in an inefficient utilization of instructions. Each step—such as fetching the counter value from memory, reducing it, and updating the value—adds an extra overhead that collectively hampers the computational process. The necessity for two counters to navigate the input data underscores a fundamental inefficiency inherent in the direct convolution approach, contrasting with the enhanced data management provided by the Im2col-IP technique.
In addition, for both implementation, the most external loop contains one more instruction, related to the `store` instruction executed by all PEs. The store-to-load multiplied incremental (MAC) ratio is increased compared to the implementation Im2col-IP. To produce 16 different output results, the calculation requires a specific number of instructions, as determined by the following equation:

$$\frac{9 \times F_X \times F_Y \times C \times K}{PE} \tag{4.2}$$

where:

- 9 represents the total number of instructions required for the inner kernel.
- F_X is the filter width.
- F_Y is the filter height.
- C is the number of input channels.
- K is the number of output channels.
- PE represents the number of Processing Elements, equal to 16.

Consequently, to complete the convolution, this algorithm must be invoked $O_X \times O_Y$ times.

Chapter 5

Experimental Results

5.1 Experimental Setup

In this study, we introduce the methodology for implementing convolution operations on a CGRA embedded within a PYNQ Z2 platform. The deployment process encompasses several stages, beginning with the creation of the FPGA bitstream and progressing to the application of convolution operations. This is followed by generating the instruction list for the CGRA, culminating in the execution of the actual application. Moreover, a detailed exploration of evaluation metrics is presented, with a particular emphasis on assessing latency, energy consumption, memory usage, and MAC (Multiply-Accumulate) operations per cycle.

5.1.1 CGRA preparation and deployment

FPGA Bitstream generation

The process of FPGA bitstream generation for the PYNQ Z2 explicitly involves configuring the FPGA to host an instance of the HEEPsilon: mainly an instance of the X-HEEP microcontroller along with the OpenEdgeCGRA accelerator. The FPGA bitstream generation process is highly customized based on the selection of core types, bus topologies, and memory addressing modes, aligning with the discussion made in section 3.2. This step tailors the FPGA's configuration to optimally support the specific requirements of the convolution application and the CGRA's architecture. Xilinx Vivado is used for this purpose.

Convolution Application Process

The convolution application process is based on the implementation techniques described in the previous chapter. This involves selecting and applying specific

algorithms and methods for performing convolution, which could include selecting between the WP method, instead of output channel parallelism, or input channel parallelism. The choice of technique affects how the convolution is executed within the CGRA, which impacts performance, accuracy, and resource utilization.

CGRA Instruction List Generation

Following the development of the application, an instruction list for the CGRA is generated. This crucial step transforms the convolution application into a dynamic configuration tailored for the CGRA within the FPGA. This process is distinct from generating an FPGA bitstream, which configures the overall FPGA resources in a static manner. Instead, for time-multiplexed or module-scheduled CGRAs, the generated instruction list adapts the CGRA's reconfigurable fabric to efficiently execute the convolution application by reconfiguring the datapath on a cycle-by-cycle basis. The dynamic method described here differs from spatial CGRAs, which set up the CGRA's operation initially and keep that configuration constant during the entire convolution process, similar to how a systolic array operates. Due to the typical understanding of "bitstream" as representing fixed settings in FPGAs, the term "instruction list" is employed in this context to specify a series of changes, each defining the data path setup for a single cycle, instead of a bitwise modification of hardware elements. This distinction is crucial to prevent misunderstandings, as the flexible nature of the instruction list in time-multiplexed CGRAs offers a more dynamic and versatile method for running applications on reconfigurable computing systems.

Running the Convolution Application on the PYNQ Z2

Finally, the convolution application runs on the FPGA. This involves loading the FPGA bitstream to configure the FPGA itself and then deploying the CGRA bitstream to set up the CGRA for the convolution operation. Once configured, the FPGA, using CGRA, runs the convolution application. This step may involve processing data inputs and producing outputs, which are the results of the convolution operations.

5.1.2 Evaluation Metrics

The evaluation of convolution mapping strategies on the described platform is detailed through specific metrics to provide insights into both utilization and efficiency. This includes comparisons involving the CPU, specifically an Ibex processor, alongside the CGRA-based computations to ensure a comprehensive evaluation of performance and efficiency.

Latency: Defined as the duration necessary to complete a full convolution process, which encompasses both the creation of the *im2col* (if applicable) and the execution of the kernel. For CPU-based computations, labeled as "CPU" in the comparative graphs, an Ibex processor was utilized. The Ibex processor, designed for efficiency and compact implementation, is a 32-bit RISC-V core. Convolution computations on the Ibex processor were optimized using the `-Ofast` compiler attribute, aiming to enhance execution speed by allowing compiler optimizations that may not strictly adhere to language standards but maximize performance. This consideration is crucial for a fair comparison, as optimizations can significantly impact measured latency and energy consumption. The initial loading time for instructions before the commencement of the first iteration is disregarded in latency measurements. An attempt to replace the Ibex core with a RI5CY core was explored to investigate potential performance improvements. However, this substitution was not successful for implementation on the FPGA platform due to the significantly larger footprint of the RI5CY core. This increased footprint resulted in an inability to generate a functional bitstream for the FPGA.
Energy: The power consumption of an integrated system, including CGRA, CPU, and memory subsystems, is considered. For the CPU scenario, energy consumption is specifically attributed to the Ibex processor under the `-Ofast` optimization setting, as described before. The comparative analysis includes this dimension to provide a balanced view of how different computational strategies — CGRA-based versus CPU-based — fare in terms of energy efficiency. The power for the CGRA was determined from post-synthesis simulations using a TSMC 65 nm technology process, indicating the CGRA's requirement of approximately 0.4 mm^2 area.
Memory Usage: The memory footprint of each strategy is evaluated based on the required storage for input and output samples, along with the weight filters. This metric is essential for understanding scalability, particularly when adjusting hyper-parameters to observe latency and memory usage trends. Memory usage is measured by calculating the 32-kilobyte memory banks instantiated.
MAC/Cycle: To benchmark the execution speed against other state-of-the-art implementations, the performance metric of `MAC` operations per clock cycle ($^{MAC}/_{cycle}$) is utilized. This is determined by evaluating the overall MAC operation involved in the convolution process and then dividing it by the number of cycles required. The total number of MAC operation is calculated using the following formula:

$$\#\ \text{MAC} = F_X \times F_Y \times C \times O_X \times O_Y \times K \tag{5.1}$$

where:

- F_X and F_Y represent the dimensions of the filter kernel in the X and Y directions, respectively.

- C stands for the number of channels in the input feature map.
- O_X and O_Y are the dimensions of the output feature map in the X and Y directions, respectively.
- K denotes the number of output channels applied during the convolution.

5.2 Latency and Energy analysis

We run a baseline convolution with $C = K = O_X = O_Y = 16$, and a 3×3 filter. For each mapping method, we measure execution latency and energy consumption of the three main blocks involved: CGRA, CPU, and memory. The results, when compared against a CPU-only implementation in Figure 5.1, showcase the energy and latency improvements achieved by the different approaches, with the WP approach highlighting significant efficiencies. It reaches energy and latency improvements of 3.4× and 9.9×, respectively, at an average power of 2.5 mW, the highest among the CGRA-approaches. The detailed outcomes of all the measurements are systematically presented in Table 5.1 for latency and Table 5.2 for energy consumption.

Latency Analysis: Observations from Table 5.1 indicate that all implementations achieve execution latencies in the order of 10 ms. This uniformity suggests a consistent efficiency across various strategies in handling convolution computations. Particularly noteworthy is the minimal impact of CPU involvement in the output channel parallelism, where the CPU's contribution to the total execution time is marginal. The slight CPU latency observed in the Conv-OP approach (0.102 ms) is attributed to the management of output data reordering, necessitated by insufficient register space for output address storage. This scenario underscores a

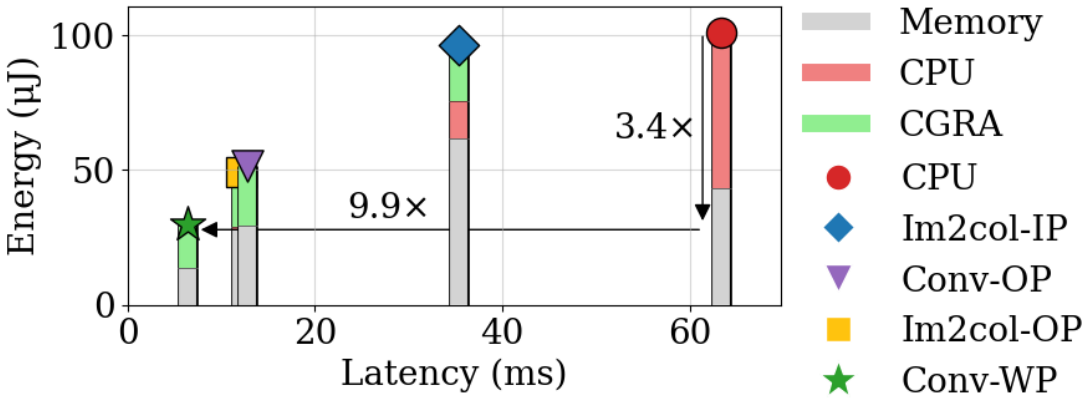


Figure 5.1: Energy vs. Latency comparison.

critical design constraint that required CPU intervention for accurately positioning the output data. On the other hand, the CPU's contribution in the Im2col-OP approach is linked to the execution of Im2col. The Im2col-IP method, as shown in Table 5.1, exhibits a higher latency predominantly due to the frequent computations of Im2col. The method's design, which necessitates the computation of Im2col for each output position and its repetition for every output channel, significantly inflates the computational overhead, thereby affecting the latency.

Energy Consumption Analysis: Reviewing the data in Table 5.2 shows that the CGRA's energy consumption remains consistent across methods, around 20 µJ. The most significant distinction among the strategies becomes apparent in the comparison of memory energy consumption. Specifically, WP method not only demonstrates the most efficient use of CGRA in terms of energy but also stands out for its memory access efficiency, resulting in the lowest overall energy consumption. This outcome highlights optimized data reuse management, significantly limiting memory accesses and, consequently, reducing the dynamic energy consumption of the memory subsystem. Notably, among the four methods analyzed, WP is the only one where memory energy consumption is lower than that of the CGRA. In contrast, in other methods, the greater energy consumption is associated with memory rather than the CGRA, emphasizing the importance of efficient memory management in improving overall energy efficiency.

Version	CPU (ms)	CGRA (ms)	Total (ms)
CPU	63.0	0.0	63.4
IM2Col-IP	20.0	15.0	35.4
Conv-OP	0.102	12.0	12.5
IM2Col-OP	0.95	11.0	12.4
WS	0.0	6.0	6.4

Table 5.1: Latency comparison in milliseconds (ms)

Version	Memory (µJ)	CPU (µJ)	CGRA (µJ)	Total (µJ)
CPU	43	57	0	101
IM2Col-IP	62	14	21	96
Conv-OP	29	0.093	22	52
IM2Col-OP	28	0.871	21	49
WS	14	0	16	30

Table 5.2: Energy consumption comparison in microjoules (µJ)

5.3 Ablation Study: Exploration of layer scalability on CGRA

We evaluate the performance deviation from the *baseline* case explored in the previous section by swiping the layer hyperparameters. We vary O_X and O_Y in $[16, 64]$, C and K in $[16, 144]$, increasing by 1 the dimension of each parameter until 32, and then in steps of 16 given the similar scalability. We limit our search to the maximum memory available in the system (512 kiB from $H\mathcal{E}\mathcal{E}Psilon$'s RAM banks). The results, illustrated in Figure 5.2, show that WP has the greatest robustness to hyperparameter changes, with increasing layer dimensions always leading to improved performance. WP remains the best approach for any hyperparameter combination, reaching up to $0.665^{MAC}/_{cycle}$ with C =16, K =16, and $O_X = O_Y =$ 64. It is noteworthy that increasing O_X and O_Y translates into an improvement in performance for the WP case thanks to two different contributions: first, the larger the input size, the higher the reuse of the loaded weights; second, a larger feature map reduces the occurrence of row changes while swiping the input activations, thus, the associated overhead of border loop.
On the other hand, all the other approaches see a drop in performance every time their parallelization dimensions are not a multiple of the number of PEs (i.e., 16), reaching their lowest performance ($\sim 0.1^{MAC}/_{cycle}$) when the parallelization dimension is equal to 17 due to the strong imbalance in the workload distribution. In this case, the Im2col-OP results the least robust with a performance reduction of 3.62× when compared to its best case.

5.4 Ablation Study: Memory interleaved impact

Memory interleaving is a strategy employed to enhance memory access speed by segmenting the memory into multiple modules and accessing them concurrently.

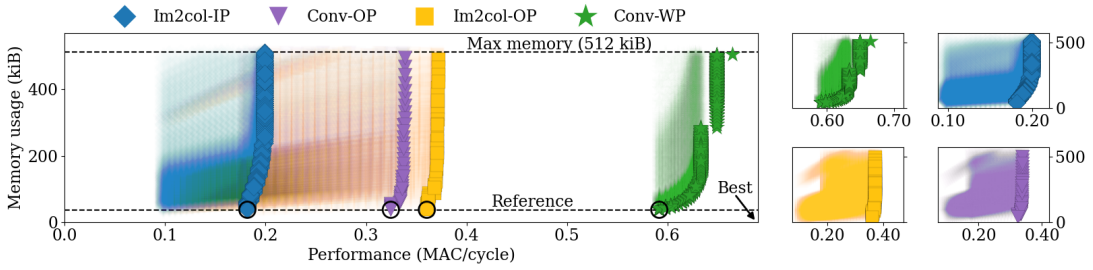


Figure 5.2: Impact on memory and performance of different hyperparameters. Pareto-optimal results are highlighted with a greater color intensity.

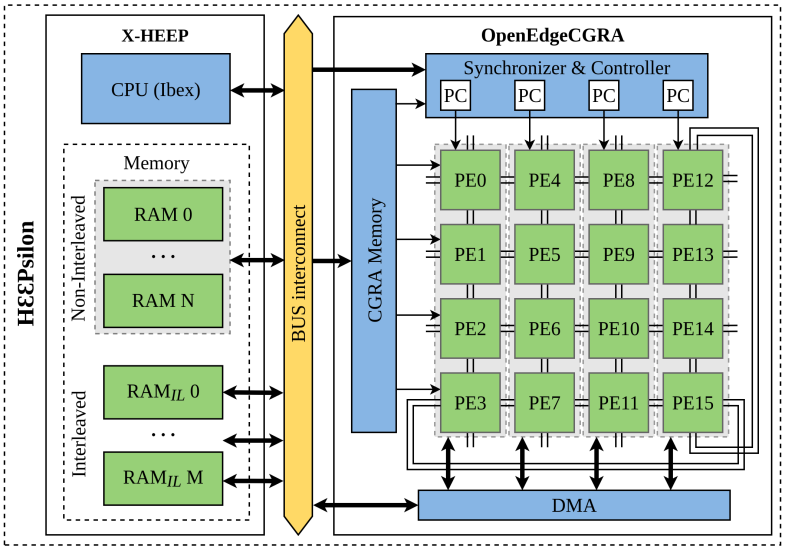


Figure 5.3: *HEEPsilon* with memory interleaved banks.

The concept involves distributing memory addresses among various memory banks or modules in a non-sequential fashion. This configuration enables a processor to access distinct memory sections simultaneously, thereby decreasing the latency for memory access and enhancing the overall system performance.
On the other hand, continuous memory allocation refers to the storage of data in a consecutive manner within a single memory module or bank. The main distinction lies in the way data is retrieved: memory interleaving allows for simultaneous access to data by distributing it across several modules, improving data transfer rate and decreasing response time. In contrast, continuous memory access follows a linear order, which may result in increased response time and diminished performance.
All the analysis conducted for this section is based on the following configuration: C=K=O_X=O_Y=16. The general configuration of this new environment is depicted in Figure 5.3.
As previously mentioned, this memory arrangement enhances performance by reducing latency in nearly all implementations. In Figure 5.4, it is evident that only the Im2col-IP technique does not exhibit any latency improvements. As explained in section 4.2.3, this technique has a unique setup where the primary factor affecting latency is the `im2col` function rather than the `cgra_call` function. Consequently, even if the latter factor diminishes thanks to the interleaved memory, the `im2col` function remains the predominant contributor to latency, as depicted in Figure 5.5.
The initial reference point in Figure 5.4 is a baseline configuration, which quantifies the latency incurred by the X-HEEP's processor (RISC-V) during the execution of a convolution operation. This baseline serves as a standard comparison for

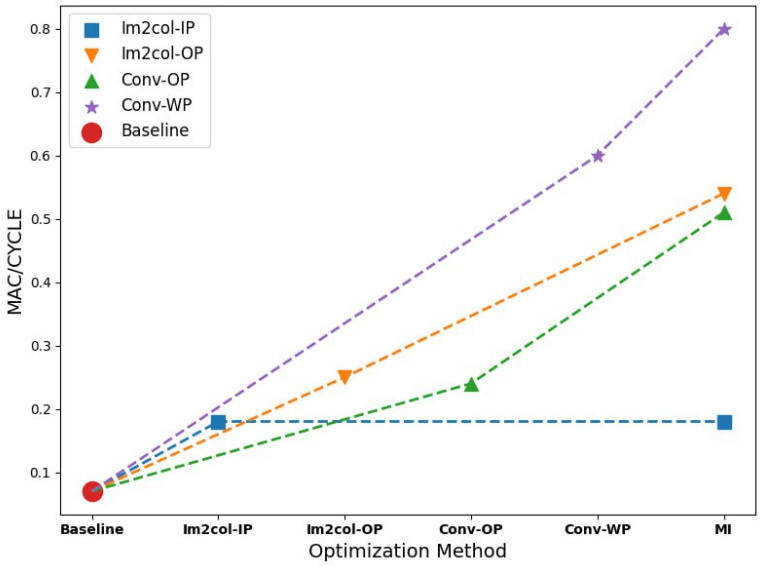


Figure 5.4: Impact of memory interleaved on the different implementations

evaluating the performance of different optimization methods applied to convolution operations. To leverage the interleaved memory configuration and the output channel parallelism, a unique weight storage approach was adopted, diverging from the conventional CHW (Channel, Height, Width) and HWC (Height, Width, Channel) data layouts. Instead, the weights were arranged in a specialized order that prioritizes the output channels. This custom storage format sequentially saves one element from each filter before moving to the next, effectively cycling through the filters across all output channels first. By doing this, the memory storage pattern aligns with the operational demands of the output channels, ensuring that each output channel's filter element is stored consecutively and so belonging to a different interleaved memory bank, thereby optimizing the memory access patterns for the convolution operation. This configuration significantly improves latency performance in terms of output channel parallelism. Specifically, the performance of Conv-OP doubles, increasing from 0.24 to $0.51^{MAC}/_{cycle}$. Similarly, the performance of Im2col-OP also nearly doubles, improving from 0.25 to $0.53^{MAC}/_{cycle}$.

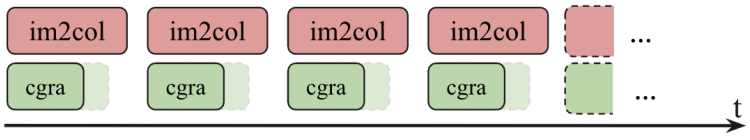


Figure 5.5: Im2col-IP with interleaved memory

Additionally, WP sees a 33% increase in its efficiency metrics. Furthermore, the WP, already highly optimized in its codebase, reaches its peak in performance. Leveraging memory interleaving offers a pathway to elevate this already efficient performance further, pushing it from a 0.60 to a $0.80^{MAC}/_{cycle}$, reaching up to 12.5× in terms of latency compared to a CPU implementation.

Chapter 6

Conclusions

In the current state of the art, Convolutional Neural Networks (CNNs) are extensively utilized in a variety of modern applications, making them a focal point for researchers. It is worth noting that, apart from offering dedicated hardware, enhancing the software design for convolutional operations can also significantly enhance both accuracy and energy efficiency. These software enhancements encompass a range of techniques including algorithmic advancements, parallelization methods, and hardware-software co-design. On the other side, there is a plethora of specialized hardware options available such as Field-Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs), and Coarse-Grained Reconfigurable Architectures (CGRAs). Research on CGRAs and their applications had been relatively quiet until recent years, but there have been numerous studies focusing on leveraging CGRA architectures to boost the performance of CNNs.

The primary focus of this thesis is to tackle the issue of mapping convolutions on CGRA. It involves exploring different cutting-edge computational and memory management approaches to identify the most effective mapping method that strikes a balance between performance and resource limitations.

During the thesis research, it becomes apparent how challenging it is to explore the vast convolution implementation space. This complexity arises from the numerous available methods for implementing convolution. It is crucial to consider the hardware resources of the analyzed CGRA while exploring these implementation strategies.

One crucial finding from this study is the significance of attempting to parallelize various hyperparameters across the available number of processing elements within the CGRA. It is essential to make use of this insight while also beginning to consider the stationarity of the different element such as weight element, input element and partial sum.

Throughout the thesis, we carried out an extensive examination of the outcomes,

focusing on the influence of various convolution implementations such as weight parallelism, output channel parallelism, and input channel parallelism. Furthermore, we observed the effects of transitioning from contiguous memory to an interleaved memory structure. An additional significant aspect that came to light was the investigation into the effects of altering various hyperparameters.

Future work

The research conducted in this thesis has demonstrated the significance of optimization in mapping CNNs onto a CGRA. There are various potential future research directions that can further expand on this thesis. This work has laid a solid foundation by demonstrating various optimization strategies and their impacts on performance. However, despite these advancements, the CGRA's general-purpose design, which includes capabilities for conditional jump instructions, still lacks specialized instructions like MAC, which are critical for such applications. This reveals a clear path for future research to incorporate such specialized instructions to optimize the hardware further for deep learning tasks.

Moreover, the current challenges associated with the CGRA's handling of blocking loads have been identified as a significant bottleneck. The blocking `LW` instruction, with its 2-cycle duration and resulting throughput of 1/2, imposes a latency that cannot be overlooked. This is in contrast to, i.e. a processor with a 4-stage pipeline where concurrent load requests can lead to more efficient utilization and reduced overall latency. The constraint of the CGRA, which can be likened to a system with just two stages, leads to a lack of capacity to deliver high throughput for instructions with longer latency, highlighting another area that could be enhanced.

Furthermore, the thesis suggests that by incorporating an additional output register within the Processing Elements (PEs), a method known as spatial dimension parallelism could be implemented. This technique would alleviate the need for each PE to store weight elements, as they could instead utilize weights from the output registers of neighboring PEs. The current setup, where the output register is 'locked' due to the presence of weights, could be enhanced with an extra output register, thus enabling a more advanced parallelism scheme.

These insights indicate several avenues for future work. Enhancing OpenEdgeCGRA with MAC instructions, introducing pipeline mechanisms to improve instruction throughput, and expanding the capabilities of PEs through additional registers are all prospective developments that could substantially increase its computational efficiency.

In conclusion, this thesis has not only addressed the complex challenge of mapping convolutions onto CGRAs, but has also made a significant contribution to the field

with the identification of optimized methods for computation and memory management. The methods introduced, such as optimizing hyperparameters in parallel and switching to interleaved memory layouts, represent a substantial advance in the pursuit of more effective CGRA architectures. A thorough examination was conducted to illustrate how various convolution techniques and adjustments to hyperparameters can impact outcomes, delineating potential avenues for enhancing performance and developing hardware specifically tailored for deep learning applications.

Chapter 7

Appendix

7.1 Assembly code

7.1.1 Weight Parallelism

This is the assembly code for the weight parallelism implementation, the best solution founded for both latency and energy evaluation.

```
0,,,
LWD R0,LWD R0,LWD R0,NOP
LWD R0,LWD R0,LWD R0,NOP
NOP,NOP,NOP,NOP
LWD R0,LWD R0,LWD R0,NOP
1,,,
LWD R1,LWD R1,LWD R1,NOP
LWD R1,LWD R1,LWD R1,NOP
LWD R1,LWD R1,LWD R1,NOP
"SADD ROUT, R0, ZERO","SADD ROUT, R0, ZERO","SADD ROUT, R0, ZERO",LWD R1
2,,,
"LWI R3, R0","LWI R3, R0","LWI R3, R0",NOP
"LWI R3, R0","LWI R3, R0","LWI R3, R0","SADD R0, ${output_row * output_col}, ZERO"
"LWI R3, RCB","LWI R3, RCB","LWI R3, RCB","SADD R0, ${output_row - 1}, ZERO"
"SADD R1, RCL, 0",NOP,NOP,NOP
3,,,
"SMUL R2, R3, R1","SMUL R2, R3, R1","SMUL R2, R3, R1",NOP
"SMUL R2, R3, R1","SMUL R2, R3, R1","SMUL R2, R3, R1","SSUB R0, R0, 1"
"SMUL R2, R3, R1","SMUL R2, R3, R1","SMUL R2, R3, R1",NOP
NOP,"SADD ROUT, RCL, ZERO","SADD R1, ZERO, 0",NOP
4,,,
NOP,NOP,"SADD R2, R2, RCL","SADD R2, RCR, ZERO"
NOP,NOP,"SADD R2, R2, RCL","SADD R2, RCR, ZERO"
NOP,NOP,"SADD R2, R2, RCL","SADD R2, RCR, ZERO"
"LWI ROUT, R1",NOP,"BEQ R1, 0, 6",NOP
5,,,
NOP,NOP,"SADD R2, R2, RCL","SADD R2, RCR, ZERO"
NOP,NOP,"SADD R2, R2, RCL","SADD R2, RCR, ZERO"
NOP,NOP,"SADD R2, R2, RCL","SADD R2, RCR, ZERO"
"LWI ROUT, R1",NOP,"SWI RCR, RCL","BEQ RCT, 0, 9"
```

```
6,,,
NOP,NOP,NOP,"SADD R2, R2, RCL"
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD R2, R2, RCL"
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD R2, R2, RCL"
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4","SADD R2, RCR, ZERO"
7,,,
"SADD R3, RCB, ZERO","SADD R3, RCB, ZERO","SADD R3, RCB, ZERO","SADD R2, R2, RCB"
"SADD R3, RCB, ZERO","SADD R3, RCB, ZERO","SADD R3, RCB, ZERO","SSUB R0, R0, 1"
"LWI R3, RCB","LWI R3, RCB","LWI R3, RCB","SSUB R0, R0, 1"
"SADD R1, R1, 4",NOP,NOP,"SADD R2, R2, RCT"
8,,,
"SMUL R2, R3, R1","SMUL R2, R3, R1","SMUL R2, R3, R1","BEQ RCB, 0, 14"
"SMUL R2, R3, R1","SMUL R2, R3, R1","SMUL R2, R3, R1","BNE R0, 0, 5"
"SMUL R2, R3, R1","SMUL R2, R3, R1","SMUL R2, R3, R1","SADD ROUT, R0, ZERO"
NOP,"SSUB ROUT, RCL, 4",NOP,"SADD R2, R2, RCB"
9,,,
NOP,NOP,NOP,"SADD R2, R2, RCL"
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD R2, R2, RCL"
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD R2, R2, RCL"
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4","SADD R2, RCR, ZERO"
10,,,
"SADD R3, RCB, ZERO","SADD R3, RCB, ZERO","SADD R3, RCB, ZERO",NOP
"SADD R3, RCB, ZERO","SADD R3, RCB, ZERO","SADD R3, RCB, ZERO",NOP
"LWI R3, RCB","LWI R3, RCB","LWI R3, RCB",NOP
NOP,NOP,NOP,NOP
11,,,
NOP,NOP,NOP,NOP
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO",NOP
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO",NOP
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4",NOP
12,,,
"SADD R3, RCB, ZERO","SADD R3, RCB, ZERO","SADD R3, RCB, ZERO",NOP
"SADD R3, RCB, ZERO","SADD R3, RCB, ZERO","SADD R3, RCB, ZERO",NOP
"LWI R3, RCB","LWI R3, RCB","LWI R3, RCB","SADD R0, ${output_row}, ZERO"
NOP,NOP,"SADD ROUT, R3, ZERO",NOP
13,,,
NOP,NOP,NOP,NOP
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO",NOP
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R2, ZERO"
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4","BEQ RCL, 0, 7"
14,,,
NOP,NOP,"SADD R2, R2, RCL","SADD R2, RCR, ZERO"
NOP,NOP,"SADD R2, R2, RCL","SADD R2, RCR, ZERO"
NOP,NOP,"SADD R2, R2, RCL","SADD R2, RCR, ZERO"
NOP,NOP,"SWI RCR, RCL",NOP
15,,,
NOP,NOP,NOP,"SADD R2, R2, RCL"
NOP,NOP,NOP,"SADD R2, R2, RCL"
NOP,NOP,NOP,"SADD R2, R2, RCL"
NOP,"SADD R1, ROUT, 4",NOP,NOP
16,,,
NOP,NOP,NOP,"SADD R2, R2, RCB"
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,"SADD R2, ZERO, RCT"

```

```
17,,,
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
"LWI R3, RCR",NOP,NOP,"SADD R2, R2, RCB"
18,,,
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
"SADD ROUT, RCL, R3",NOP,NOP,NOP
19,,,
EXIT,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,"SWI RCL, R1",NOP,NOP
```

This code is tailored to the specific number of input dimensions. Such code offers greater flexibility compared to others, as it relies on only one hyperparameter, the input dimension. This is another significant factor contributing to its robustness compared to other implementations.

7.1.2 Output channel Parallelism

IM2COL

```
0,,,
LWD R0,LWD R0,LWD R0,LWD R0
LWD R0,LWD R0,LWD R0,LWD R0
LWD R0,LWD R0,LWD R0,LWD R0
LWD R0,LWD R0,LWD R0,LWD R0
1,,,
LWD R2,LWD R2,LWD R2,LWD R2
LWD R2,LWD R2,LWD R2,LWD R2
LWD R2,LWD R2,LWD R2,LWD R2
LWD R2,LWD R2,LWD R2,LWD R2
2,,,
NOP,NOP,"SADD ROUT, ${output_channel//16}, ZERO",NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
3,,,
NOP,NOP,"SWI ROUT, 8",NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
4,,,
NOP,NOP,"SADD ROUT, ${9 * input_channel}, ZERO",NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
5,,,
NOP,NOP,"SWI ROUT, 4",NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
```

```
6,,,
"LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
"LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
"LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
"LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
7,,,
"LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2"
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
8,,,
NOP,NOP,NOP,NOP
"SADD ROUT, RCT, ZERO","SADD ROUT, RCT, ZERO","SADD ROUT, RCT, ZERO","SADD ROUT,
    RCT, ZERO"
NOP,NOP,NOP,NOP
"SADD ROUT, RCB, ZERO","SADD ROUT, RCB, ZERO","SADD ROUT, RCB, ZERO","SADD ROUT,
    RCB, ZERO"
9,,,
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
"SADD ROUT, RCB, ZERO","SADD ROUT, RCB, ZERO","SADD ROUT, RCB, ZERO","SADD ROUT,
    RCB, ZERO"
NOP,NOP,NOP,NOP
10,,,
"SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT
    , R1"
"SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT
    , R1"
"SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT
    , R1"
"SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT
    , R1"
11,,,
"SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
"SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
"SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
"SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
12,,,
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4"
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4"
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4"
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4"
13,,,
"SADD R2, R2, 4","SADD R2, R2, 4","SADD R2, R2, 4","SADD R2, R2, 4"
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
14,,,
"LWI ROUT, 4",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
15,,,
"SSUB ROUT, ROUT, 1",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
16,,,
"BNE ROUT, ZERO, 6",NOP,NOP,NOP
"SWI RCT, 4",NOP,NOP,NOP
NOP,NOP,NOP,NOP
```

```
NOP,NOP,NOP,NOP
17,,,
SWD R3,SWD R3,SWD R3,SWD R3
SWD R3,SWD R3,SWD R3,SWD R3
SWD R3,SWD R3,SWD R3,SWD R3
SWD R3,SWD R3,SWD R3,SWD R3
18,,,
NOP,NOP,NOP,NOP
"SADD ROUT, R2, ZERO","SADD ROUT, R2, ZERO","SADD ROUT, R2, ZERO","SADD ROUT, R2, ZERO"
"LWI ROUT, 8",NOP,NOP,NOP
NOP,NOP,NOP,NOP
19,,,
"SADD R2, RCB, ZERO","SADD R2, RCB, ZERO","SADD R2, RCB, ZERO","SADD R2, RCB, ZERO"
NOP,NOP,NOP,NOP
"SSUB ROUT, ROUT, 1",NOP,NOP,NOP
NOP,NOP,NOP,NOP
20,,,
NOP,NOP,NOP,NOP
"BNE RCB, ZERO, 4",NOP,NOP,NOP
"SWI ROUT, 8",NOP,NOP,NOP
NOP,NOP,NOP,NOP
21,,,
EXIT,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
```

Direct Convolution

```
0,,,
LWD R0,LWD R0,LWD R0,LWD R0
LWD R0,LWD R0,LWD R0,LWD R0
LWD R0,LWD R0,LWD R0,LWD R0
LWD R0,LWD R0,LWD R0,LWD R0
1,,,
LWD R2,LWD R2,LWD R2,LWD R2
LWD R2,LWD R2,LWD R2,LWD R2
LWD R2,LWD R2,LWD R2,LWD R2
LWD R2,LWD R2,LWD R2,LWD R2
2,,,
NOP,NOP,"SADD ROUT, ${output_channel//16}, ZERO",NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
3,,,
NOP,NOP,"SWI ROUT, 16",NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
4,,,
NOP,NOP,"SADD ROUT, ${input_channel}, ZERO",NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
5,,,
```

```
27 NOP,NOP,"SWI ROUT, 12",NOP
28 NOP,NOP,NOP,NOP
29 NOP,NOP,NOP,NOP
30 NOP,NOP,NOP,NOP
31 6,,,
32 NOP,"SADD ROUT, 3, ZERO",NOP,NOP
33 NOP,NOP,NOP,NOP
34 NOP,NOP,NOP,NOP
35 NOP,NOP,NOP,NOP
36 7,,,
37 NOP,"SWI ROUT, 8",NOP,NOP
38 NOP,NOP,NOP,NOP
39 NOP,NOP,NOP,NOP
40 NOP,NOP,NOP,NOP
41 8,,,
42 "SADD ROUT, 3, ZERO",NOP,NOP,NOP
43 NOP,NOP,NOP,NOP
44 NOP,NOP,NOP,NOP
45 NOP,NOP,NOP,NOP
46 9,,,
47 "SWI ROUT, 4",NOP,NOP,NOP
48 NOP,NOP,NOP,NOP
49 NOP,NOP,NOP,NOP
50 NOP,NOP,NOP,NOP
51 10,,,
52 "LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
53 "LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
54 "LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
55 "LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
56 11,,,
57 "LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2"
58 NOP,NOP,NOP,NOP
59 NOP,NOP,NOP,NOP
60 NOP,NOP,NOP,NOP
61 12,,,
62 NOP,NOP,NOP,NOP
63 "SADD ROUT, RCT, ZERO","SADD ROUT, RCT, ZERO","SADD ROUT, RCT, ZERO","SADD ROUT,
       RCT, ZERO"
64 NOP,NOP,NOP,NOP
65 "SADD ROUT, RCB, ZERO","SADD ROUT, RCB, ZERO","SADD ROUT, RCB, ZERO","SADD ROUT,
       RCB, ZERO"
66 13,,,
67 NOP,NOP,NOP,NOP
68 NOP,NOP,NOP,NOP
69 "SADD ROUT, RCB, ZERO","SADD ROUT, RCB, ZERO","SADD ROUT, RCB, ZERO","SADD ROUT,
       RCB, ZERO"
70 NOP,NOP,NOP,NOP
71 14,,,
72 "SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT
       , R1"
73 "SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT
       , R1"
74 "SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT
       , R1"
75 "SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT
       , R1"
76 15,,,
77 "SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
78 "SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
79 "SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
80 "SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
```

```
16,,,
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4"
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4"
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4"
"SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4","SADD R0, R0, 4"
17,,,
"SADD R2, R2, 4","SADD R2, R2, 4","SADD R2, R2, 4","SADD R2, R2, 4"
"LWI ROUT, 4",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
18,,,
NOP,NOP,NOP,NOP
"SSUB ROUT, ROUT, 1",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
19,,,
NOP,NOP,NOP,NOP
"BNE ROUT, ZERO, 10",NOP,NOP,NOP
"SWI RCT, 4",NOP,NOP,NOP
NOP,NOP,NOP,NOP
20,,,
"SADD R2, R2, ${first_cycle}","SADD R2, R2, ${first_cycle}","SADD R2, R2, ${
    first_cycle}","SADD R2, R2, ${first_cycle}"
"LWI ROUT, 8",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
21,,,
NOP,NOP,NOP,NOP
"SSUB ROUT, ROUT, 1",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
22,,,
NOP,NOP,NOP,NOP
"BNE ROUT, ZERO, 8",NOP,NOP,NOP
"SWI RCT, 8",NOP,NOP,NOP
NOP,NOP,NOP,NOP
23,,,
"SADD R2, R2, ${second_cycle}","SADD R2, R2, ${second_cycle}","SADD R2, R2, ${
    second_cycle}","SADD R2, R2, ${second_cycle}"
"LWI ROUT, 12",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
24,,,
NOP,NOP,NOP,NOP
"SSUB ROUT, ROUT, 1",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
25,,,
NOP,NOP,NOP,NOP
"BNE ROUT, ZERO, 6",NOP,NOP,NOP
"SWI RCT, 12",NOP,NOP,NOP
NOP,NOP,NOP,NOP
26,,,
SWD R3,SWD R3,SWD R3,SWD R3
SWD R3,SWD R3,SWD R3,SWD R3
SWD R3,SWD R3,SWD R3,SWD R3
SWD R3,SWD R3,SWD R3,SWD R3
27,,,
NOP,NOP,NOP,NOP
"SADD ROUT, R2, ZERO","SADD ROUT, R2, ZERO","SADD ROUT, R2, ZERO","SADD ROUT, R2,
    ZERO"
```

```
"LWI ROUT, 16",NOP,NOP,NOP
NOP,NOP,NOP,NOP
28,,,
"SADD R2, RCB, ZERO","SADD R2, RCB, ZERO","SADD R2, RCB, ZERO","SADD R2, RCB, ZERO
    "
NOP,NOP,NOP,NOP
"SSUB ROUT, ROUT, 1",NOP,NOP,NOP
NOP,NOP,NOP,NOP
29,,,
NOP,NOP,NOP,NOP
"BNE RCB, ZERO, 4",NOP,NOP,NOP
"SWI ROUT, 16",NOP,NOP,NOP
NOP,NOP,NOP,NOP
30,,,
EXIT,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
```

The assembly code presented reveals a performance bottleneck stemming from its dependence on two key hyperparameters: the output and input channels. While the effect of the output channel is reduced by distributing the workload across 16 Processing Elements (PEs), as evidenced by dividing the output channel by 16, the input channel does not experience the same level of parallelization. Consequently, each PE is burdened with an unequal workload, with a heavier emphasis on the input channel. This disparity implies that although parallelization helps alleviate the impact of the output channel on performance, the input channel remains a crucial factor that could potentially lead to suboptimal performance, particularly in terms of latency.

7.1.3 Input channel Parallelism

```
0,,,
LWD R0,LWD R0,LWD R0,LWD R0
LWD R0,LWD R0,LWD R0,LWD R0
LWD R0,LWD R0,LWD R0,LWD R0
LWD R0,LWD R0,LWD R0,LWD R0
1,,,
LWD R2,LWD R2,LWD R2,LWD R2
LWD R2,LWD R2,LWD R2,LWD R2
LWD R2,LWD R2,LWD R2,LWD R2
LWD R2,LWD R2,LWD R2,LWD R2
2,,,
"SADD ROUT, ZERO, ${9 * input_channel//16}",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
3,,,
"SWI ROUT, 4",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
```

```
4,,,
"LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
"LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
"LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
"LWI R1, R0","LWI R1, R0","LWI R1, R0","LWI R1, R0"
5,,,
"LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2"
"LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2"
"LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2"
"LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2","LWI ROUT, R2"
6,,,
"SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1"
"SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1"
"SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1"
"SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1","SMUL ROUT, ROUT, R1"
7,,,
"SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
"SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
"SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
"SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3","SADD R3, ROUT, R3"
8,,,
"SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}"
"SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}"
"SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}"
"SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}","SADD R0, R0, ${input_channel * 4}"
9,,,
"SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}"
"SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}"
"SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}"
"SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}","SADD R2, R2, ${input_channel * 4}"
10,,,
"LWI ROUT, 4",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
11,,,
"SSUB ROUT, ROUT, 1",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
12,,,
"BNE ROUT, ZERO, 4",NOP,NOP,NOP
"SWI RCT, 4",NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
13,,,
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO"
```

```
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO"
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO"
"SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO","SADD ROUT, R3, ZERO"
14,,,
"SADD ROUT, ROUT, RCB","SADD ROUT, ROUT, RCB","SADD ROUT, ROUT, RCB","SADD ROUT, ROUT, RCB"
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
"SADD ROUT, ROUT, RCT","SADD ROUT, ROUT, RCT","SADD ROUT, ROUT, RCT","SADD ROUT, ROUT, RCT"
15,,,
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
"SADD ROUT, ROUT, RCB","SADD ROUT, ROUT, RCB","SADD ROUT, ROUT, RCB","SADD ROUT, ROUT, RCB"
16,,,
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
"SADD ROUT, ROUT, RCR",NOP,NOP,"SADD ROUT, ROUT, RCL"
17,,,
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,"SADD ROUT, ROUT, RCR"
18,,,
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,SWD ROUT
19,,,
EXIT,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
NOP,NOP,NOP,NOP
```

Upon initial inspection, this code appears to be based on the same logic as described in Section 7.1.1, as it also involves only one hyper-parameter dependency, which is the input channel. However, in this research, this particular implementation performs the poorest in terms of both latency and energy consumption. The underlying cause for this outcome lies in its operation: while there is indeed only one dependency, the number of `store` operations is limited to just one throughout the entire execution.

Bibliograpy

[1] Yann LeCun, Y. Bengio, and Geoffrey Hinton. «Deep Learning». In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539 (cit. on p. 1).

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386 (cit. on p. 1).

[3] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. «DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving». In: Dec. 2015, pp. 2722–2730. DOI: 10.1109/ICCV.2015.312 (cit. on p. 1).

[4] Panakanti Shashanka and Tatireddy Subba Reddy. «Dermatologist-Level Classification of Skin Cancer Using Cascaded Ensembling of Convolutional Neural Network». In: *2023 International Conference on Research Methodologies in Knowledge Management, Artificial Intelligence and Telecommunication Engineering (RMKMATE)*. 2023, pp. 1–5. DOI: 10.1109/RMKMATE59243.2023.10368872 (cit. on p. 1).

[5] Xiang Fu. «GomokuPro: An Implementation of Enhanced Machine Learning Algorithm Utilizing Convolutional Neural Network in Gomoku Strategy and Predictions Model». In: *2022 7th International Conference on Intelligent Computing and Signal Processing (ICSP)*. 2022, pp. 1671–1677. DOI: 10.1109/ICSP54964.2022.9778476 (cit. on p. 1).

[6] Kyoung-Su Oh and Keechul Jung. «GPU implementation of neural networks». In: *Pattern Recognition* 37.6 (2004), pp. 1311–1314. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2004.01.013. URL: https://www.sciencedirect.com/science/article/pii/S0031320304000524 (cit. on p. 1).

[7] Chiao-Yu Liang, Yang-Rwei Chang, Po-Hsiang Yang, and Horng-Yuan Shih. «A High Efficiency Hardware Accelerator for Convolution Neural Network». In: *2023 9th International Conference on Applied System Innovation (ICASI)*. 2023, pp. 157–159. DOI: 10.1109/ICASI57738.2023.10179516 (cit. on p. 1).

[8] *Fpga based hardware accelerator for convolution neural network*. 2022, pp. 260–264. DOI: 10.1109/ICMACC54824.2022.10093648 (cit. on p. 1).

[9] Ke Cheng, Jiaxuan Fu, Yulong Shen, Haichang Gao, Ning Xi, Zhiwei Zhang, and Xinghui Zhu. «Manto: A Practical and Secure Inference Service of Convolutional Neural Networks for IoT». In: *IEEE Internet of Things Journal* 10.16 (2023), pp. 14856–14872. DOI: 10.1109/JIOT.2023.3251982 (cit. on p. 1).

[10] Xiaoyang Wang, Zhe Zhou, Zhihang Yuan, Jingchen Zhu, Yulong Cao, Yao Zhang, Kangrui Sun, and Guangyu Sun. «FD-CNN: A Frequency-Domain FPGA Acceleration Scheme for CNN-Based Image-Processing Applications». In: 22.6 (Nov. 2023). ISSN: 1539-9087. DOI: 10.1145/3559105. URL: https://doi.org/10.1145/3559105 (cit. on p. 1).

[11] Yixing Li et al. «A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks». In: *ACM JETC* 14 (2017), pp. 1–16. URL: https://api.semanticscholar.org/CorpusID:37965395 (cit. on p. 1).

[12] Francesco Conti and Luca Benini. «A Ultra-Low-Energy Convolution Engine for Fast Brain-Inspired Vision in Multicore Clusters». In: vol. 2015. Mar. 2015. DOI: 10.7873/DATE.2015.0404 (cit. on p. 1).

[13] Ali A. D. Farahani, Hakem Beitollahi, and Mahmood Fathi. «A Dynamic General Accelerator for Integer and Fixed-Point Processing». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.12 (2020), pp. 2509–2517. DOI: 10.1109/TVLSI.2020.3023106 (cit. on pp. 1, 2, 11).

[14] Yanan Lu, Leibo Liu, Jianfeng Zhu, Shouyi Yin, and Shaojun Wei. «Architecture, challenges and applications of dynamic reconfigurable computing». In: *Journal of Semiconductors* 41.2 (Feb. 2020), p. 021401. DOI: 10.1088/1674-4926/41/2/021401. URL: https://dx.doi.org/10.1088/1674-4926/41/2/021401 (cit. on pp. 1, 2).

[15] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. «A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications». In: *ACM Computing Surveys* 52 (Oct. 2019), pp. 1–39. DOI: 10.1145/3357375 (cit. on p. 2).

[16] Li Zhou, Jianfeng Zhang, and Hengzhu Liu. «Dual-Issue CGRA for DAG Acceleration: 4th International Conference of Pioneering Computer Scientists, Engineers and Educators, ICPCSEE 2018, Zhengzhou, China, September 21-23, 2018, Proceedings, Part I». In: Jan. 2018, pp. 505–511. ISBN: 978-981-13-2202-0. DOI: 10.1007/978-981-13-2203-7_40 (cit. on p. 2).

[17] Mark Wijtvliet, Luc Waeijen, and Henk Corporaal. «Coarse grained reconfigurable architectures in the past 25 years: Overview and classification». In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)* (2016), pp. 235–244. URL: https://api.semanticscholar.org/CorpusID:18520899 (cit. on pp. 2, 15).

[18] Jungi Lee and Jongeun Lee. «Specializing CGRAs for Light-Weight Convolutional Neural Networks». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.10 (2022), pp. 3387–3399. DOI: 10.1109/TCAD.2021.3123178 (cit. on p. 2).

[19] Masakazu Tanomoto, Shinya Takamaeda-Yamazaki, Jun Yao, and Yasuhiko Nakashima. «A CGRA-Based Approach for Accelerating Convolutional Neural Networks». In: *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. 2015, pp. 73–80. DOI: 10.1109/MCSoC.2015.41 (cit. on p. 2).

[20] Rubén Rodríguez Álvarez, Benoıt Denkinger, Juan Sapriza, José Miranda Calero, Giovanni Ansaloni, and David Atienza Alonso. «An Open-Hardware Coarse-Grained Reconfigurable Array for Edge Computing». In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. CF '23. Bologna, Italy: Association for Computing Machinery, 2023, pp. 391–392. ISBN: 9798400701405. DOI: 10.1145/3587135.3591437. URL: https://doi.org/10.1145/3587135.3591437 (cit. on pp. 2, 19–21).

[21] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. Aug. 13, 2017. arXiv: 1703.09039[cs]. URL: http://arxiv.org/abs/1703.09039 (visited on 08/22/2023) (cit. on p. 4).

[22] Keiron O'Shea and Ryan Nash. «An Introduction to Convolutional Neural Networks». In: *CoRR* abs/1511.08458 (2015). arXiv: 1511.08458. URL: http://arxiv.org/abs/1511.08458 (cit. on p. 5).

[23] Ahnaf Farhan, Olga Kosheleva, and Vladik Kreinovich. «Why Max and Average Poolings are Optimal in Convolutional Neural Networks». In: 2018. URL: https://api.semanticscholar.org/CorpusID:53056693 (cit. on p. 5).

[24] Mobeen Ahmad, Jooyeon Joe, and Dongil Han. «CortexNet: Convolutional Neural Network with Visual Cortex in human brain». In: *2018 IEEE International Conference on Consumer Electronics - Asia (ICCE-Asia)*. 2018, pp. 206–212. DOI: 10.1109/ICCE-ASIA.2018.8552151 (cit. on p. 5).

[25] Quan Deng, Lei Jiang, Youtao Zhang, Minxuan Zhang, and Jun Yang. «DrAcc: a DRAM based Accelerator for Accurate CNN Inference». In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)* (2018), pp. 1–6. URL: https://api.semanticscholar.org/CorpusID:49303184 (cit. on p. 11).

[26] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. «PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory». In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA '16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 27–39. ISBN: 9781467389471. DOI: 10.1109/ISCA.2016.13. URL: https://doi.org/10.1109/ISCA.2016.13 (cit. on p. 11).

[27] Yuhao Zhang, Zhiping Jia, Hongchao Du, Runzhen Xue, Zhaoyan Shen, and Zili Shao. «A Practical Highly Paralleled ReRAM-Based DNN Accelerator by Reusing Weight Pattern Repetitions». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.4 (2022), pp. 922–935. DOI: 10.1109/TCAD.2021.3071116 (cit. on p. 11).

[28] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. «Neurocube: a programmable digital neuromorphic architecture with high-density 3D memory». In: *SIGARCH Comput. Archit. News* 44.3 (June 2016), pp. 380–392. ISSN: 0163-5964. DOI: 10.1145/3007787.3001178. URL: https://doi.org/10.1145/3007787.3001178 (cit. on p. 11).

[29] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. «Origami: A Convolutional Network Accelerator». In: *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*. GLSVLSI '15. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2015, pp. 199–204. ISBN: 9781450334747. DOI: 10.1145/2742060.2743766. URL: https://doi.org/10.1145/2742060.2743766 (cit. on p. 11).

[30] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. «Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks». In: Feb. 2016. DOI: 10.1109/ISSCC.2016.7418007 (cit. on p. 11).

[31] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. «ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars». In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 14–26. DOI: 10.1109/ISCA.2016.12 (cit. on p. 11).

[32] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. «YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights». In: (June 2016) (cit. on p. 11).

[33] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. «Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks». In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 161–170. ISBN: 9781450333153. DOI: 10.1145/2684746.2689060. URL: https://doi.org/10.1145/2684746.2689060 (cit. on p. 12).

[34] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. «A programmable parallel accelerator for learning and classification». In: *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. Vienna, Austria: Association for Computing Machinery, 2010, pp. 273–284. ISBN: 9781450301787. DOI: 10.1145/1854273.1854309. URL: https://doi.org/10.1145/1854273.1854309 (cit. on p. 12).

[35] Clement Farabet, Cyril Poulet, Jefferson Han, and Yann Lecun. «CNP: An FPGA-based processor for Convolutional Networks». In: Aug. 2009. DOI: 10.1109/FPL.2009.5272559 (cit. on p. 12).

[36] Gerald Estrin. «Organization of computer systems: the fixed plus variable structure computer». In: *IRE-AIEE-ACM '60 (Western)*. 1960. URL: https://api.semanticscholar.org/CorpusID:16384320 (cit. on p. 12).

[37] Reiner Hartenstein, Alexander Hirschbiel, Michael Riedmuller, Karin Schmidt, and Michael Weber. «A Novel Asic Design Approach Based On A New Machine Paradigm». In: *Solid-State Circuits, IEEE Journal of* 26 (Aug. 1991), pp. 975–989. DOI: 10.1109/4.92017 (cit. on p. 12).

[38] Sabih H. Gerez, Sonia M. Heemstra de Groot, Erwin R. Bonsma, and Marc J. M. Heijligers. «Overlapped Scheduling Techniques for High-Level Synthesis and Multiprocessor Realizations of DSP Algorithms». In: *Advanced Techniques for Embedded Systems Design and Test*. Ed. by Juan Carlos López, Román Hermida, and Walter Geisselhardt. Boston, MA: Springer US, 1998, pp. 125–150. ISBN: 978-1-4757-4419-4. DOI: 10.1007/978-1-4757-4419-4_6. URL: https://doi.org/10.1007/978-1-4757-4419-4_6 (cit. on p. 12).

[39] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. «ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix». In: *International Conference on Field-Programmable Logic and Applications*. 2003. URL: https://api.semanticscholar.org/CorpusID:39182312 (cit. on p. 12).

[40] H. Singh, Ming-Hau Lee, Guangming Lu, F.J. Kurdahi, N. Bagherzadeh, and E.M. Chaves Filho. «MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications». In: *IEEE Transactions on Computers* 49.5 (2000), pp. 465–481. DOI: 10.1109/12.859540 (cit. on p. 12).

[41] Mark Horowitz. «1.1 Computing's energy problem (and what we can do about it)». In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323 (cit. on p. 12).

[42] G. Theodoridis, D. Soudris, and S. Vassiliadis. «A Survey of Coarse-Grain Reconfigurable Architectures and Cad Tools». In: *Fine- and Coarse-Grain Reconfigurable Computing*. Ed. by Stamatis Vassiliadis and Dimitrios Soudris. Dordrecht: Springer Netherlands, 2007, pp. 89–149. ISBN: 978-1-4020-6505-7. DOI: 10.1007/978-1-4020-6505-7_2. URL: https://doi.org/10.1007/978-1-4020-6505-7_2 (cit. on p. 12).

[43] L. Liu, Z. Li, Y. Chen, C. Deng, S. Yin, and S. Wei. «HReA: An energy-efficient embedded dynamically reconfigurable fabric for 13-dwarfs processing». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.3 (2017), pp. 381–385. DOI: 10.1109/TCSII.2017.2728814 (cit. on pp. 12, 13).

[44] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matthew Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. «Plasticine: A reconfigurable architecture for parallel patterns». In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (2017), pp. 389–402. URL: https://api.semanticscholar.org/CorpusID:10243767 (cit. on p. 12).

[45] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. «Stream-Dataflow Acceleration». In: *ACM SIGARCH Computer Architecture News* 45 (June 2017), pp. 416–429. DOI: 10.1145/3140659.3080255 (cit. on p. 12).

[46] Defense Advanced Research Projects Agency (DARPA). *DARPA*. 2017. URL: https://www.darpa.mil/ (cit. on p. 12).

[47] Sukjin Kim, Young-Hwan Park, Jaehyun Kim, Minsoo Kim, Wonchang Lee, and Shihwa Lee. «Flexible video processing platform for 8K UHD TV». In: Aug. 2015, pp. 1–1. DOI: 10.1109/HOTCHIPS.2015.7477475 (cit. on p. 12).

[48] Samsung. *Exynos 5 Octa 5430*. 2014. URL: http://www.samsung.com/semiconductor/minisite/exynos/products/mobile%20processor/exynos-5-octa-5430/ (cit. on p. 12).

[49] PACT XPP TECHNOLOGIES. *PACT*. URL: http://www.pactxpp.com/ (cit. on p. 12).

[50] INTEL. *INTEL*. 2016. URL: https://newsroom.intel.com/news-releases/intel-tsinghua-university-and-montage-tech (cit. on p. 13).

[51] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. «A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications». In: *ACM Comput. Surv.* 52.6 (Oct. 2019). ISSN: 0360-0300. DOI: 10.1145/3357375. URL: https://doi.org/10.1145/3357375 (cit. on pp. 13–15).

[52] Bingfeng Mei, Bjorn De Sutter, Tom Vander Aa, Maryse Wouters, Andreas Kanstein, and S. Dupont. «Implementation of a Coarse-Grained Reconfigurable Media Processor for AVC Decoder». In: *Journal of Signal Processing Systems* 51 (June 2008), pp. 225–243. DOI: 10.1007/s11265-007-0152-8 (cit. on p. 13).

[53] Mahendra Kumar Angamuthu Ganesan, Sundeep Singh, Frank May, and Jürgen Becker. «H. 264 Decoder at HD Resolution on a Coarse Grain Dynamically Reconfigurable Architecture». In: *2007 International Conference on Field Programmable Logic and Applications* (2007), pp. 467–471. URL: https://api.semanticscholar.org/CorpusID:17979368 (cit. on p. 13).

[54] Leibo Liu, Chenchen Deng, Dong Wang, Min Zhu, Shouyi Yin, Peng Cao, and Shaojun Wei. «An energy-efficient coarse-grained dynamically reconfigurable fabric for multiple-standard video decoding applications». In: *Proceedings of the IEEE 2013 Custom Integrated Circuits Conference, CICC 2013, San Jose, CA, USA, September 22-25, 2013*. IEEE, 2013, pp. 1–4. DOI: 10.1109/CICC.2013.6658434. URL: https://doi.org/10.1109/CICC.2013.6658434 (cit. on p. 13).

[55] Arthur H. Veen. «Dataflow machine architecture». In: *ACM Comput. Surv.* 18.4 (Dec. 1986), pp. 365–396. ISSN: 0360-0300. DOI: 10.1145/27633.28055. URL: https://doi.org/10.1145/27633.28055 (cit. on p. 14).

[56] R. Hartenstein. «A decade of reconfigurable computing: a visionary retrospective». In: *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*. 2001, pp. 642–649. DOI: 10.1109/DATE.2001.915091 (cit. on p. 15).

[57] Russell Tessier, Kenneth Pocek, and Andre Dehon. «Reconfigurable Computing Architectures». In: *Proceedings of the IEEE* 103 (Mar. 2015), pp. 332–354. DOI: 10.1109/JPROC.2014.2386883 (cit. on p. 15).

[58] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Miguel Peón-Quirós, and David Atienza. *X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators*. 2024. arXiv: 2401.05548 [cs.AR] (cit. on pp. 17, 18).

[59] Cristian Tirelli et al. «SAT-based Exact Modulo Scheduling Mapping for Resource-Constrained CGRAs». In: *ACM Journal on Emerging Technologies in Computing Systems* (2024). arXiv: 2402.12834 (cit. on pp. 19, 21).

[60] Loris Duch, Soumya Basu, Rubén Braojos, David Atienza, Giovanni Ansaloni, and Laura Pozzi. «A multi-core reconfigurable architecture for ultra-low power bio-signal analysis». In: *2016 IEEE Biomedical Circuits and Systems Conference (BioCAS)*. 2016, pp. 416–419. DOI: 10.1109/BioCAS.2016.7833820 (cit. on p. 21).

[61] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*. Vol. 1. MIT Press, 2017 (cit. on p. 23).

[62] Liangzhen Lai et al. «CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs». In: *CoRR* abs/1801.06601 (2018). arXiv: 1801.06601. URL: http://arxiv.org/abs/1801.06601 (cit. on pp. 23, 24).

[63] Angelo Garofalo et al. «PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors». In: *Phil. Trans. of the Royal Society A* 378.2164 (2020), p. 20190155 (cit. on pp. 24, 25).

[64] Tianqi Chen et al. «MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems». In: *CoRR* abs/1512.01274 (2015). arXiv: 1512.01274. URL: http://arxiv.org/abs/1512.01274 (cit. on p. 24).

[65] Sanjay Krishnan et al. «Artificial Intelligence in Resource-Constrained and Shared Environments». In: *SIGOPS Oper. Syst. Rev.* 53.1 (July 2019), pp. 1–6. ISSN: 0163-5980. DOI: 10.1145/3352020.3352022. URL: https://doi.org/10.1145/3352020.3352022 (cit. on p. 24).

[66] Vivienne Sze et al. «Designing DNN Accelerators». In: *Efficient Processing of Deep Neural Networks*. Springer Cham, 2022. Chap. 5, pp. 73–118 (cit. on p. 27).