



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

qGamma: An Exploration Framework for the Mapping of Mixed-Precision Quantized DNN Models on Hardware Accelerators

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA IN-
FORMATICA

Author: **Luigi Altamura**

Student ID: 978261

Advisor: Prof. Cristina Silvano

Co-advisor: Dr. Leandro Fiorin

Academic Year: 2023-24

Abstract

Nowadays, there is a huge interest in custom spatial accelerators for on-the-edge Artificial Intelligence applications. In particular, on-the-edge Deep Neural Network accelerators are typically based on spatial architectures composed of multiple processing elements interacting with the memory hierarchy through a network-on-chip. The energy-performance efficiency of these accelerators is given by an optimized mapping of the dataflow to the hardware resources and strategies to optimize data movement and reuse. In addition, mixed-precision quantization models can help reduce latency, energy, and memory consumption. The goal of this thesis is to propose an exploration framework for mapping Deep Learning (DL) models to a mixed-precision quantized target architecture as a DNN accelerator.

To achieve this main goal, we developed qGamma, a flexible framework enabling the exploration and optimal mapping of mixed-precision DNNs on general on-the-edge accelerators supporting multiple compute fixed- and floating- point precisions.

Using a domain-specific genetic algorithm-based method and qMaestro, an analytical performance and energy model based on hardware synthesis results and CACTI-D, qGamma explores the immense design space to find the mapping minimizing latency, total energy, or energy-delay product. We evaluated the exploration results on various DNNs inference workloads showing the impact of using mixed-precision models when compared to fixed-precision implementations.

Keywords: spatial accelerators, DNN accelerators, DNN mapping, DNN mixed-precision quantization

Abstract in lingua italiana

Al giorno d'oggi, c'è un grande interesse riguardo gli acceleratori per applicazioni di Intelligenza Artificiale nell'ambito dell'on-the-edge computing. In particolare, gli acceleratori per Deep Neural Network (DNN) sono tipicamente basati su architetture spaziali composte da più elementi di elaborazione che interagiscono con la gerarchia di memoria attraverso una rete su chip. L'efficienza energetica e le performance di questi acceleratori è data da una mappatura ottimizzata del flusso di dati basata sulle risorse hardware e da strategie per ottimizzare il movimento e il riutilizzo dei dati stessi. Inoltre, i modelli di quantizzazione a precisione mista possono contribuire a ridurre la latenza, l'energia e il consumo di memoria. L'obiettivo di questa tesi è proporre un framework di esplorazione per la mappatura dei modelli di Deep Learning (DL) su un'architettura quantizzata a precisione mista come può essere un acceleratore per DNN.

Per raggiungere questo obiettivo, abbiamo sviluppato qGamma, un framework flessibile che consente l'esplorazione e la mappatura ottimale di DNN a precisione mista su acceleratori on-the-edge generici che supportano precisioni multiple di calcolo in virgola mobile e fissa.

Utilizzando un metodo basato su un algoritmo genetico specifico per il dominio e qMaestro, un modello analitico di prestazioni ed energia basato sui risultati della sintesi hardware e su CACTI-D, qGamma esplora l'immenso spazio di progettazione per trovare la mappatura che minimizza la latenza, l'energia totale o il prodotto energia-latenza. Abbiamo valutato i risultati dell'esplorazione su varie DNN, mostrando l'impatto dell'uso di modelli a precisione mista rispetto alle implementazioni a precisione fissa.

Parole chiave: acceleratori, acceleratori per DNN, architetture spaziali, precisione mista quantizzata in DNN, mappature per DNN

Contents

Abstract	i
Abstract in lingua italiana	iii
Contents	v
1 Introduction	1
1.1 Motivation	1
1.2 Thesis Objective	2
1.3 Thesis Organization	2
2 Background	5
2.1 Overview of Machine Learning and Neural Network Techniques	5
2.1.1 Machine Learning	5
2.1.2 Artificial Neural Networks	7
2.1.3 Deep Learning	9
2.1.4 Convolutional Neural Networks	10
2.2 DNN Accelerators	15
2.2.1 Accelerator structure	15
2.3 Mixed-Precision Quantization in Deep Learning	16
2.4 Data Reuse in DNN Accelerators	19
2.4.1 Dataflow	20
2.4.2 Weight Stationary	22
2.4.3 Input Stationary	24
2.4.4 Output Stationary	24
2.4.5 No Local Reuse	25
2.4.6 Row Stationary	25
3 Related Work	27

3.1	GAMMA	27
3.1.1	GAMMA algorithm	28
3.1.2	Flow for Automated Mapping Search	30
3.2	MAESTRO	31
3.2.1	Analytical Cost Model	32
3.3	DiGamma	33
3.3.1	Technical Approach Overview	33
3.4	Other Tools	35
3.4.1	TimeLoop	35
4	Target Architecture	37
4.1	Target Architecture Overview	37
4.1.1	Hardware model of a PE	38
5	Proposed Methodology	41
5.1	High-level Overview	41
5.2	qGamma	42
5.2.1	Genome Extension	43
5.2.2	Dataflow Description	44
5.2.3	GA Heuristics	44
5.3	qMaestro	46
5.3.1	Configurable Accelerator Model	47
5.3.2	Activity-Based Energy Model	48
6	Experimental Results	49
6.1	Experiment Setup	49
6.1.1	Accelerator configuration and Setup	49
6.1.2	Benchmarks	50
6.1.3	Evaluation Metrics	52
6.2	Results	52
6.2.1	Results on ResNetV1 optimized for EDP	52
6.2.2	Results on MobileNetV1 optimized for energy	55
6.2.3	Results on ResNet18 optimized by latency	57
7	Conclusions and future developments	59
	Bibliography	61

List of Figures

67

List of Tables

69

1 | Introduction

1.1. Motivation

Artificial Intelligence (AI), especially Deep Learning, has made remarkable progress in various fields, including computer vision, prediction, classification, and pattern recognition.

Deep Neural Network (DNN) and Convolutional Neural Network (CNN) play an important role in AI applications used in business and everyday life. Instead of traditional programming, where explicit instructions are given to perform tasks, these techniques generate their result without explicit programming, but by learning from data. These networks are structured with multiple layers that perform feature extraction and classification, taking inspiration from the functioning human brain.

DL inference requires a huge computational effort, so the accelerator running the target application needs to allocate a lot of hardware resources to produce results in a reasonable time.

In order to achieve execution efficiency, the research is moving towards better DNN models that reduce the power consumption, but it is also moving towards specialized accelerators. There are different types of DNN accelerators based on different technologies, ranging from small/medium FPGAs and microcontrollers to large-scale GPUs such as the NVIDIA H100 Tensor Core. In general, these accelerators have limited resources, such as memory capacity and computational power and must be used to achieve the application's goals.

Many optimizations can be done to reduce the execution time of DNNs, especially CNNs, to optimize accelerator resources. Each CNN layer computes millions of operations, mainly Multiply and Accumulate (MAC), which can be described as multidimensional loops that can be ordered, tiled, and scheduled in myriad ways across the space and time of the DNN accelerators. Each of these options is called a mapping. Choosing the right mapping increases data reuse and optimizes memory usage.

Another optimization is the use of a mixed-precision quantization network. Mixed-

precision quantization allows activations and filters to be stored and processed in different quantizations based on layer characteristics. This allows for shortening the inference time with approximately the same accuracy, keeping sensitive layers in higher precision, and applying low precision quantization only to insensitive layers [8].

The main objective of this thesis is to extend an existing framework that automates the hardware mapping of the DNN model on accelerators, named GAMMA [15], to allow the exploration of mixed-precision quantization mapping and evaluate the performance of accelerators and DNNs under different mixed-precision scenarios. We named the new version of the tool qGamma (quantized GAMMA).

GAMMA relies on MAESTRO [20]. MAESTRO is a cost analysis framework that GAMMA uses to evaluate mappings on the accelerator. To ensure the proper functioning of qGamma, MAESTRO has also been modified to its quantized version. We have named this new version qMAESTRO (quantized MAESTRO).

1.2. Thesis Objective

Based on the existing framework GAMMA, we developed an automatic hardware mapper for mixed-precision DNN models on accelerators. To do so, we defined a target architecture and we modified MAESTRO to support qGamma. The main objectives are:

- Define the target spatial architecture to be used as a model in our framework. Also define the memory hierarchy, processing element architecture, and energy model.
- Modify the open-source SotA framework GAMMA to enable mixed precision analysis by updating the input DNN model and adapting the genetic algorithm for evaluation.
- Modify the SotA open-source MAESTRO framework to include the model of the target architecture and to evaluate mixed-precision mappings and model the target architecture for cost analysis.
- Define and execute a set of experiments to evaluate the efficiency of the proposed tools on various benchmarks and case studies in terms of energy and delay metrics.

1.3. Thesis Organization

The thesis is organized into seven chapters, described as follows:

- **Chapter 1 - Introduction:** This chapter introduces the motivations and objec-

tives of this thesis.

- **Chapter 2 - Background:** This chapter provides an overview on the prior knowledge needed to better understand the problem. The chapter starts with an overview of machine learning and AI and goes deeper into the explanation of DNNs and CNNs. The chapter also presents an explanation of accelerators and different strategies to optimize the flow of data and the mixed-precision technique to have better performance while maintaining accuracy.
- **Chapter 3 - Related Works:** This chapter describes all work related to this thesis. Some of them were used as a starting point for this work.
- **Chapter 4 - Target Architecture:** This chapter provides all the information about the selected target architecture. The chapter starts with a general description of the accelerator and then goes into the composition of a processing element.
- **Chapter 5 - Proposed Methodology:** This chapter describes the design choices and modifications to GAMMA and MAESTRO to enable mixed precision quantization mapping. The chapter includes both the theoretical decision and some implementation explanations.
- **Chapter 6 - Experimental Results:** This chapter presents the experimental setup, the tools and accelerator configuration used, and the results of the experiments.
- **Chapter 7 - Conclusions and future developments:** This chapter discusses the conclusions of the thesis and the possible improvements and future work.

2 | Background

This chapter covers several aspects. Section 2.1 begins with an overview of machine learning, continues with artificial neural networks, and concludes with a focus on convolutional neural networks (CNNs), exploring their architecture and applications. Section 2.2 discusses accelerators tailored for DNNs, detailing their role and importance in improving computational performance for convolutional tasks. In Section 2.3, the focus shifts to mixed-precision quantization, and finally, Section 2.4 discusses data reuse in accelerators.

2.1. Overview of Machine Learning and Neural Network Techniques

2.1.1. Machine Learning

Machine Learning (ML) is a subfield of Artificial Intelligence (AI) that focuses on the development of algorithms and models that enable computers to learn from and make predictions or decisions based on data. Unlike traditional rule-based programming where explicit instructions are provided to perform tasks, in machine learning, the emphasis is on allowing the system to learn from data patterns and improve its performance over time without being explicitly programmed as shown in Fig. 2.1. This ability to learn and adapt autonomously makes machine learning algorithms powerful tools for solving a wide range of complex problems across various domains.

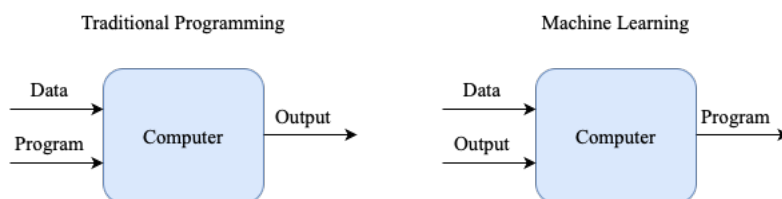


Figure 2.1: Traditional Programming vs. Machine Learning.

A formal definition is given by Tom M. Mitchell in his book in 1997: *"A computer program*

is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E ."[24].

Machine learning approaches can be categorized into three main types: **supervised learning**, **unsupervised learning**, and **reinforcement learning**.

Supervised learning uses labeled input-output pairs to predict outcomes for unknown data. In this framework, a dataset is divided into training and test sets. Given the training dataset with desired outputs $D = \{\langle x, t \rangle\}$ of an unknown function f , the goal is to find a good approximation of f that generalizes well to the test data. The input variables x are called features and the output variables t are usually called labels or targets.

If the target variables t are discrete, the task is called **classification**. Here, the algorithm aims to categorize input data points into predefined classes or categories. For example, in binary classification, the algorithm might determine whether an email is "spam" or "not spam" based on features like keywords and sender information. In multi-class classification, the algorithm might classify images of animals into categories like "cat," "dog," or "bird".

On the other hand, when the target variables t are continuous, the task is known as **regression**. In regression, the algorithm attempts to predict numerical values or outcomes based on input features. For example, when predicting house prices, the algorithm can use features such as square footage, number of bedrooms, and location to estimate the price of a property as a continuous numerical value. Other examples of regression tasks include estimating the sales revenue of a product or predicting the temperature for the next day.

Unsupervised learning involves algorithms that autonomously discover patterns and structures in unlabeled data. These algorithms operate without explicit guidance, recognizing relationships and groupings based on similarities, differences, and patterns within the data itself. This approach is particularly useful for organizing large data sets into meaningful clusters, uncovering previously unrecognized patterns, and identifying features critical to data categorization. For example, when analyzing weather data, an unsupervised learning algorithm can separate data points based on temperature or weather patterns, helping to identify seasonal changes or specific weather phenomena. Unsupervised learning involves three main tasks: **clustering**, **association**, and **dimensionality reduction**.

Clustering techniques group data points into clusters based on similarities or differences for applications such as customer segmentation and image analysis.

Association rule mining uncovers relationships between data points in large datasets, revealing correlations and co-occurrences within data, with applications in retail analytics and clinical diagnosis.

Dimensionality reduction techniques extract essential features from data sets, reducing the number of irrelevant or redundant dimensions while preserving data integrity. Techniques such as **Principal Component Analysis (PCA)** and **Singular Value Decomposition (SVD)** facilitate the analysis of high-dimensional data, improving understanding and insight extraction.

Reinforcement learning aims to achieve specific goals by maximizing the cumulative rewards over multiple actions. For example, in a game context, this could mean maximizing the points accumulated over successive moves. The term "*reinforcement*" comes from the nature of the system, which penalizes incorrect decisions while rewarding correct ones. Unlike supervised learning, reinforcement learning doesn't require labeled input/output pairs or immediate correction of suboptimal actions. Instead, it often operates on a delayed-return principle, where the consequences of decisions are realized only after several steps, allowing the learning of long-term strategies. This method relies on a **trial-and-error** approach where the agent autonomously discovers rewarding actions through iterative trials, striking a balance between exploiting learned knowledge and exploring new actions to improve performance. In addition, reinforcement learning emphasizes the importance of **delayed rewards**, recognizing that actions can affect not only current but also future outcomes. Currently, reinforcement learning is being applied in diverse domains such as board games, healthcare and self-driving car facilitating autonomous decision making and adaptive behavior.

2.1.2. Artificial Neural Networks

The history of Artificial Neural Networks (ANNs) dates back to the pioneering work of McCulloch and Pitts in 1943 [23], who proposed a computational model of a simplified neuron. This laid the foundation for the development of ANNs, which simulate the behavior of interconnected neurons in the biological brain.

These networks consist of interconnected nodes, or neurons, organized into layers: an input layer, one or more hidden layers, and an output layer. As explained in the book *Artificial Intelligence A Modern Approach* [28], each neuron receives input signals from predecessor nodes, performs a weighted sum of these inputs, and applies a nonlinear function to produce an output. Let a_j denote the output of unit j , and let $w_{i,j}$ be the weight

attached to the connection from neuron i to neuron j ; then we have

$$a_j = g_j \left(\sum_i w_{i,j} a_i \right)$$

where g_j is a nonlinear **activation function** associated with unit j and $\sum_i w_{i,j} a_i$ is the weighted sum of the inputs to neuron j (Fig. 2.2).

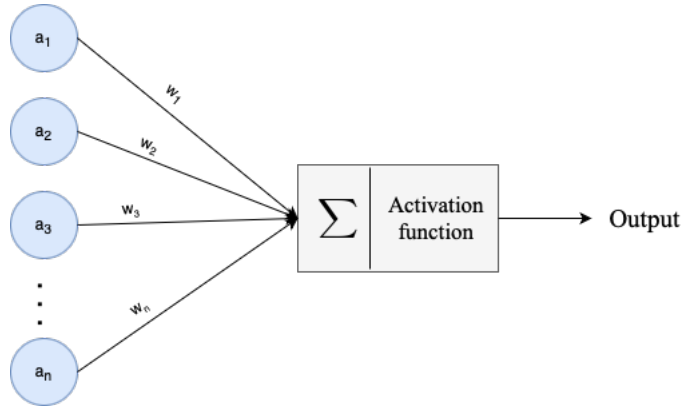


Figure 2.2: Schema of an Artificial Neuron.

A variety of different activation functions are used. The most common (Fig. 2.3) are the following:

- The logistic or **sigmoid** function:

$$\sigma(x) = 1 / (1 + e^{-x})$$

- The **ReLU** function, whose name is an abbreviation for **rectified linear unit**:

$$ReLU(x) = \max(0, x)$$

- The **softplus** function, a smooth version of the ReLU function:

$$softplus(x) = \log(1 + e^x)$$

The derivative of the softplus function is the sigmoid function.

- The **tanh** function:

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

The range of of tanh is $(-1, +1)$. Tanh is a scaled and shifted version of the sigmoid, as $\tanh(x) = 2\sigma(2x) - 1$.

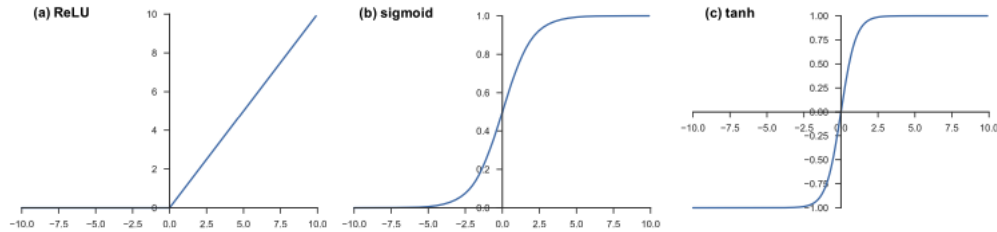


Figure 2.3: Activation functions usually applied to neural networks: (a) ReLU, and (b) sigmoid, (c) tanh [37].

The weights assigned to the connections between neurons are adjusted during training, typically using optimization algorithms such as gradient descent, to minimize a defined loss function.

Over the years, ANNs have evolved into diverse architectures, including feedforward, recurrent, convolutional, and generative adversarial networks, each tailored to address specific tasks ranging from classification and regression to image recognition and natural language processing.

2.1.3. Deep Learning

Deep learning, synonymous with **deep neural networks**, represents a significant advancement over traditional artificial neural networks. While both use interconnected nodes or neurons arranged in layers, deep learning differs by incorporating multiple hidden layers, an example of which is shown in Fig. 2.4.

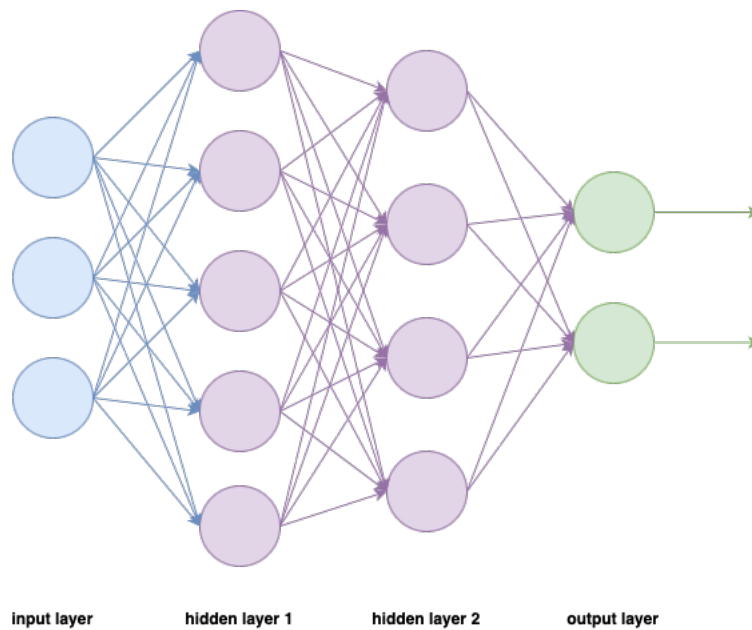


Figure 2.4: An example of Deep Neural Network.

These layers allow for a more complex and hierarchical representation of data, enabling deep neural networks to extract complex features and patterns with unprecedented depth and accuracy. Unlike neural networks, which struggle to capture complex relationships within data, deep neural networks excel at modeling high-dimensional and abstract data such as images, text, and audio. In addition, the depth of the network, characterized by the number of hidden layers, is a defining feature of deep learning architectures. With the advent of deep learning, significant advancements have been made in areas such as computer vision, natural language processing, and autonomous systems, driving innovation and revolutionizing multiple industries.

2.1.4. Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a deep learning architecture designed to learn directly from data. Particularly suited for pattern recognition tasks, CNNs excel at detecting and identifying objects, classes, and categories within images. In addition, their effectiveness extends to other data modalities such as audio, time series, and signal data, making them versatile tools for classification tasks in various domains.

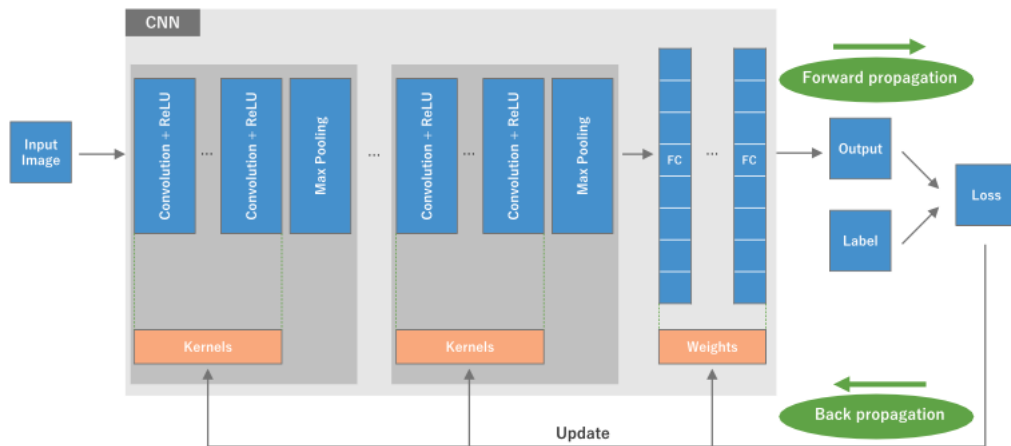


Figure 2.5: An overview of a CNN [37].

The CNN structure [37] includes several components, including **convolutional layers**, **pooling layers**, and **fully connected layers**. A standard configuration, shown in Fig. 2.5, often involves iterations of a series consisting of multiple convolutional layers and a pooling layer, followed by one or more fully connected layers. The process by which input data is transformed into output through these layers is called forward propagation.

A **convolutional layer** is a fundamental element within CNN architectures, facilitating feature extraction through a combination of linear and nonlinear operations, namely the convolution operation and the activation function.

Convolution is a specialized form of linear operation used for feature extraction. It involves applying a small numerical array, called kernel or filter, to the input, called feature map (fmap), which is represented as an array of numbers or a tensor. At each location of the tensor, an element-wise product between each filter element and the corresponding input fmap (ifmap) is computed and summed to yield the output value at that location of the output tensor, called output feature map (ofmap). This process is iterated by applying multiple kernels to generate numerous feature maps, each representing different characteristics of the input tensors. Thus, different kernels can be thought of as different feature extractors. The two primary hyperparameters that control the convolution operation are the size and the number of filters. The former typically ranges from 3x3 to 7x7, while the latter is arbitrary and determines the depth of the ofmap.

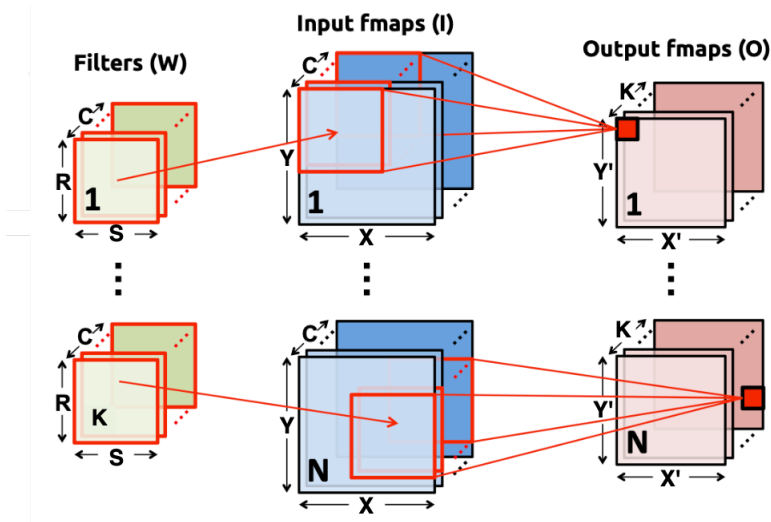


Figure 2.6: An example of convolution operation with a kernel size of 3x3, no padding and stride 1 [5].

Given the shape parameters in Table 2.1, the computation of convolutional layer is defined as

$$\mathbf{O}[z][u][x][y] = \mathbf{B}[u] + \sum_{k=0}^{C-1} \sum_{i=0}^{S-1} \sum_{j=0}^{R-1} \mathbf{I}[z][k][Ux+i]x\mathbf{W}[u][k][i][j],$$

$$0 \leq z < N, 0 \leq u < K, 0 \leq x < X', 0 \leq y < Y', X' = (X - R + U)/U, Y' = (Y - S + U)/U.$$

O, I, W and B are the matrices of the ofmaps, ifmaps, filters and biases, respectively. U is a given stride size. Fig. 2.6 shows a graphical representation of this calculation.

Shape Parameter	Description
N	batch size of 3D fmaps
K	# of 3D filters / # of ofmap channels
C	# of ifmap/filter channels
X,Y	ifmap plane width/height
S,R	filter plane width/height
X',Y'	ofmap plane width/height

Table 2.1: Shape parameters of a convolutional layer [5].

Conventional convolution operations do not allow the central part of each kernel to overlap with the outermost element of the input tensor, resulting in a reduced height and width of the output feature map relative to the input tensor. Padding, typically in the form of zero padding, overcomes this limitation by adding rows and columns of zeros to each

side of the input tensor, ensuring that the center of a kernel aligns with the outermost element and maintains the same in-plane dimension throughout the convolution operation. Modern CNN architectures often use zero padding to maintain in-plane dimensions for the application of multiple layers. Without zero padding, each successive feature map would become smaller after the convolution operation.

A key aspect of convolution operations is weight sharing, where kernels are shared across all image locations. This feature provides several benefits: preserving local feature patterns across image locations, facilitating the learning of spatial hierarchies of feature patterns through downsampling and pooling operations, and improving model efficiency by reducing the number of parameters compared to fully connected neural networks.

During CNN model training, the convolution layer training process revolves around identifying the most effective kernels for a given task based on the provided training data set. Kernels are the only parameters automatically learned during training, while kernel size, number of kernels, padding, and stride are predetermined hyperparameters set before the training process begins. The table 2.2 contains all parameters and hyperparameters in all layers of a CNN.

	Parameters	Hyperparameters
Convolution layer	Kernels	Kernel size, number of kernels, stride, padding, activation function
Pooling layer	None	Pooling method, filter size, stride, padding
Fully connected layer	Weights	Number of weights, activation function
Others		Model architecture, optimizer, learning rate, loss function, mini-batch size, epochs, regularization, weight initialization, dataset splitting

Table 2.2: A list of parameters and hyperparameters in a CNN [37].

A **pooling layer** performs a common downsampling process to reduce the in-plane dimensionality of the feature maps. This reduction helps to introduce translation invariance for small displacements and distortions, while reducing the number of subsequent learnable parameters. In particular, pooling layers do not contain learnable parameters, but hyperparameters such as filter size, stride, and padding, similar to convolution operations.

Of the various pooling operations, **max pooling** is the most widely used. It involves extracting patches from the ifmaps, identifying the maximum value within each sample, and discarding the remaining values. A common practice is to use max pooling with a

2x2 filter size and a stride of 2, which effectively downsamples the in-plane dimension of the feature maps by a factor of 2, as shown in Fig. 2.7. While the height and width dimensions are reduced, the depth dimension remains unchanged.

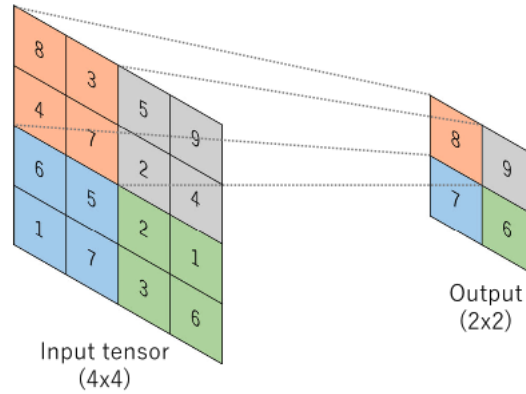


Figure 2.7: An example of max pooling operation with a filter of size 2x2 no padding, and stride of 2 [37].

Another pooling operation is **global average pooling**, which performs extensive downsampling. Here, a feature map with height x width dimensions is downsampled to a 1x1 array by calculating the average of all elements within each feature map, while preserving the depth dimension. This operation is typically applied before fully connected layers. The advantage of global average pooling is that it reduces the number of parameters to learn and allows the CNN to handle inputs of different sizes without requiring resizing.

The ofmaps from the final convolution or pooling layer are typically flattened, transforming them into a one-dimensional array or vector of numbers. These flattened features are then connected to one or more fully connected layers, also called **dense layers**, where each input is connected to each output via learnable weights. After extracting and downsampling features through convolution and pooling layers, respectively, a subset of fully connected layers maps these features to the final outputs of the network, such as class probabilities in classification tasks. Typically, the final fully connected layer contains the same number of output nodes as there are classes in the task. Each fully connected layer is followed by a nonlinear activation function, typically ReLU.

The choice of activation function for the last **fully connected layer** is usually different from the others and should be tailored to the task. For multiclass classification tasks, a **softmax** function is often applied to the last fully connected layer. This function normalizes the output real values to the target class probabilities, ensuring that each

value lies between 0 and 1, with the sum of all values equal to 1. Different types of tasks may require different activation functions for the final layer.

2.2. DNN Accelerators

To achieve optimal performance, it is critical to optimize not only the DNNs but also the accelerators. Accelerators are based on a variety of technologies, ranging from small/medium FPGAs and microcontrollers to large-scale GPUs such as the NVIDIA H100 Tensor Core. Accelerators excel in parallel processing, making them suitable for handling the massive matrix operations inherent in convolution operations within DNNs. The high throughput greatly accelerates the computation of millions of multiply-accumulate (MAC) operations per second, a critical metric for Deep Learning performance. This capability makes them essential for both training and inference tasks in DNNs, enabling rapid experimentation and deployment of complex models. In addition, accelerators offer significant advantages in energy efficiency and power consumption, enabling efficient processing of Deep Learning algorithms while minimizing energy costs and environmental impact. To further improve efficiency, techniques such as mixed-precision quantization and data reuse and mapping strategies have emerged. These techniques are discussed in sections 2.4 and 2.3, respectively.

2.2.1. Accelerator structure

As shown in Figure 2.8, most DNN accelerators use numerous processing elements (**PE**s) to exploit the parallelism of DNN applications. These PEs typically consist of scratchpad memories (**L1**) and arithmetic logic units (**ALUs**) responsible for performing MACs operations. To reduce the energy and time required to access dynamic random access memory (**DRAM**), many DNN accelerators include a scratchpad buffer (**L2**) with sufficient capacity to stage data to feed all PEs. The shared L2 buffer and the PEs are interconnected via a network-on-chip (**NoC**) [19].

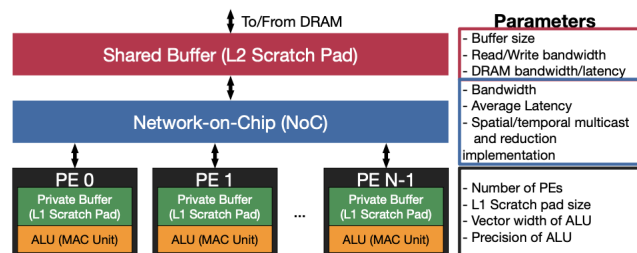


Figure 2.8: Abstract DNN accelerator architecture model [19].

In this thesis, spatial architectures (SAs) [5] are considered. SAs are a class of accelerators that exploit direct communication between PEs to achieve high computational parallelism. Instead of SIMD/SIMT, this architecture is particularly suitable for producer-consumer relationships or for exploring efficient data sharing between regions of PEs. There are two types of SAs: coarse-grained SAs, which consist of tiled arrays of ALU-style PEs interconnected by an on-chip network, and fine-grained SAs, which are usually FPGAs. Fig. 2.9 shows a CNN block diagram accelerator with an off-chip DRAM representing a coarse-grained SA proposed in Eyeriss [5].

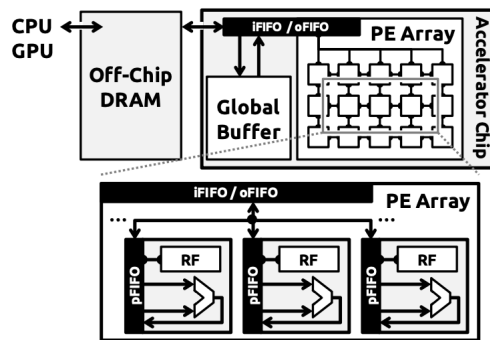


Figure 2.9: High level block diagram spatial architecture for CNN with an off-chip DRAM [5].

2.3. Mixed-Precision Quantization in Deep Learning

Mixed-precision quantization refers to the practice of using different numerical precision formats during the execution of neural networks to achieve a balance between computational efficiency and accuracy. In the context of Deep Learning, this technique allows the use of lower precision floating- and fixed-point arithmetic in certain parts of the neural network where high precision is not critical to maintaining inference accuracy [8]. By strategically using mixed precision, computational efficiency and power savings can be realized, as lower-precision formats require fewer computational resources and consume less power compared to higher-precision formats such as 32-bit floating point (FP32) or 64-bit floating point (FP64). Despite the reduced precision, careful optimization and algorithmic advancements ensure that the accuracy of the neural network is preserved in critical operations, enabling efficient training and inference without compromising overall performance. This approach enables the use of larger models, handling of larger datasets, and acceleration of deep learning tasks while minimizing the use of computational resources and energy consumption.

An example of an accelerator that supports mixed-precision computations is the **7nm four-core mixed-precision AI chip** proposed by IBM [21], which is designed to support a range of computational accuracies to meet diverse application requirements for both training and inference tasks. This innovative chip architecture, shown in Fig 2.10, incorporates cutting-edge algorithmic techniques to introduce a new Hybrid FP8 (**HFP8**) format alongside existing FP16 support, enabling efficient training and inference without compromising model accuracy. The chip also integrates INT4 and INT2 formats for highly scalable inference capabilities. In addition, its design incorporates a high-bandwidth ring interconnect for efficient multi-core scalability, facilitating seamless data communication between cores. To optimize power consumption, it employs workload-aware power control mechanisms to ensure maximum performance within specified power constraints. This innovative work demonstrates the potential of mixed-precision computing to achieve leading-edge efficiency for low-precision training and inference tasks in deep learning applications.

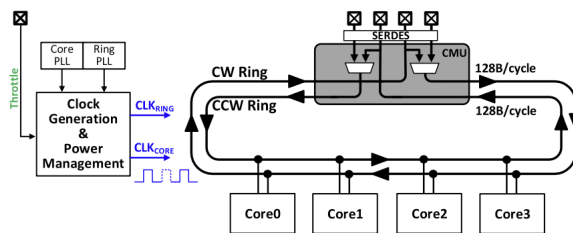


Figure 2.10: Four-core mixed-precision AI chip architecture [21]

In industry, numerous examples illustrate the practical implementation of mixed-precision computing to improve the efficiency and performance of deep learning tasks. For example, companies such as NVIDIA have introduced GPU architectures such as Ampere and Turing, which include dedicated tensor cores [22] capable of performing mixed-precision computations. Enabling mixed-precision involves two key steps: adapting the model to use half-precision data where appropriate, and using loss scaling to preserve small gradient values during training (Fig. 2.11).

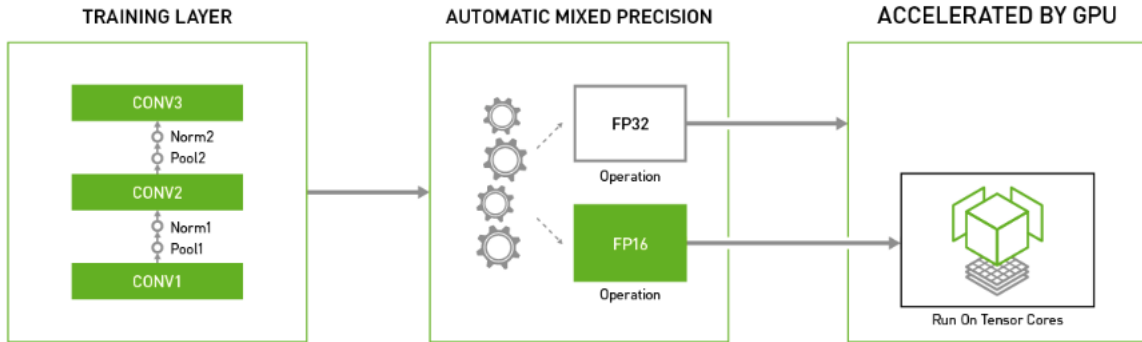


Figure 2.11: Mixed precision example in deep learning [25].

Early implementations of mixed-precision computing in GPUs played a critical role, with architectures such as NVIDIA’s Pascal and Tesla setting the stage for the development of more advanced multi-precision GPUs in subsequent generations. The NVIDIA Pascal architecture [9] not only supports half-precision FP16 computation, but also introduces additional mixed-precision capabilities to improve performance in various applications. GPUs such as the Tesla P100, powered by the GP100 GPU, excel at FP16 arithmetic, delivering twice the throughput of FP32. In addition, the GP102, GP104, and GP106 GPUs include instructions for performing integer dot products on 8-bit and 16-bit vectors, providing significant efficiency gains for deep learning inference tasks and other applications such as radio astronomy.

These advancements in mixed-precision computing have profound implications for various computational tasks. For example, the introduction of 8-bit integer 4-element vector dot product (DP4A) and 16-bit 2-element vector dot product (DP2A) instructions, shown in Fig. 2.12, in the Pascal GPUs enables highly efficient linear algebraic computations, including matrix multiplications and convolutions. This is particularly valuable for implementing 8-bit integer convolutions in deep learning inference, a common requirement in image classification and object detection tasks [9].

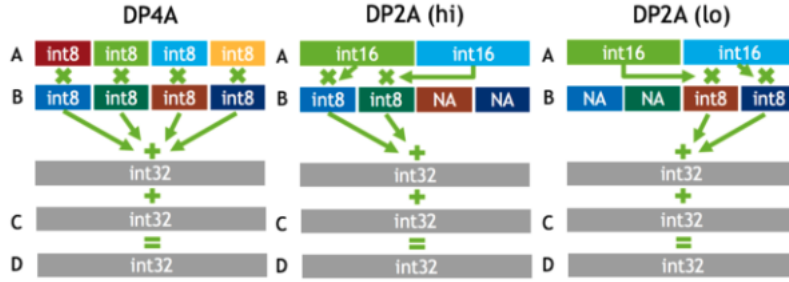


Figure 2.12: New DP4A and DP2A instructions in Tesla P4 and P40 GPUs provide fast 2- and 4-way 8-bit/16-bit integer vector dot products with 32-bit integer accumulation [9].

One notable application of these new instructions is in the cross-correlation algorithm used in radio telescope data processing pipelines. Radio astronomers often work with low-precision data captured by telescope elements, making floating-point computation unnecessary for cross-correlation. By leveraging DP4A instructions, GPUs can significantly improve the power efficiency of this computation, as demonstrated by the Tesla P40 GPU. Modifications to cross-correlation codes to utilize DP4A result in substantial efficiency improvements, making the process nearly 12 times more efficient compared to FP32 computation on previous-generation GPUs [9].

2.4. Data Reuse in DNN Accelerators

Data reuse plays an important role in enhancing both latency and energy efficiency within DNN accelerators by minimizing the number of remote buffer accesses. Organizing data usage provides opportunities for data reuse, either across successive time instances on the same PE, called **temporal reuse**, or across multiple PEs but not within successive time instances, called **spatial reuse**.

It is possible to categorize data reuse in DNN accelerators into four different types [20]. Each type requires appropriate hardware support to effectively exploit the identified data reuse opportunities. These types can be grouped under communication types:

- **Spatial/Temporal Multicast.** This category refers to spatial or temporal reuse capabilities within input tensors, such as filters and input activations. Data can be multicast to multiple PEs simultaneously (spatial reuse) or across different time instances (temporal reuse). Fig. 2.13 illustrates examples of such patterns, using a fanout NoC for spatial multicast and buffering for temporal multicast. For example, spatial multicast delivers tiles 1 and 2 to PE1 and PE2 simultaneously, taking

advantage of hardware capabilities such as fanout. Alternatively, temporal multicast involves storing data for future reuse, which requires buffering mechanisms.

- **Spatial/Temporal Reduction.** In this category, spatial reuse opportunities manifest within the output activation tensor, involving the accumulation of partial outputs across multiple PEs. Fig.2.13 illustrates this with examples of spatial reduction, facilitated by mechanisms such as store-and-forward hardware. Temporal reuse, on the other hand, involves computing partial sums over time and accumulating them in the same location, requiring buffering mechanisms to store intermediate results.

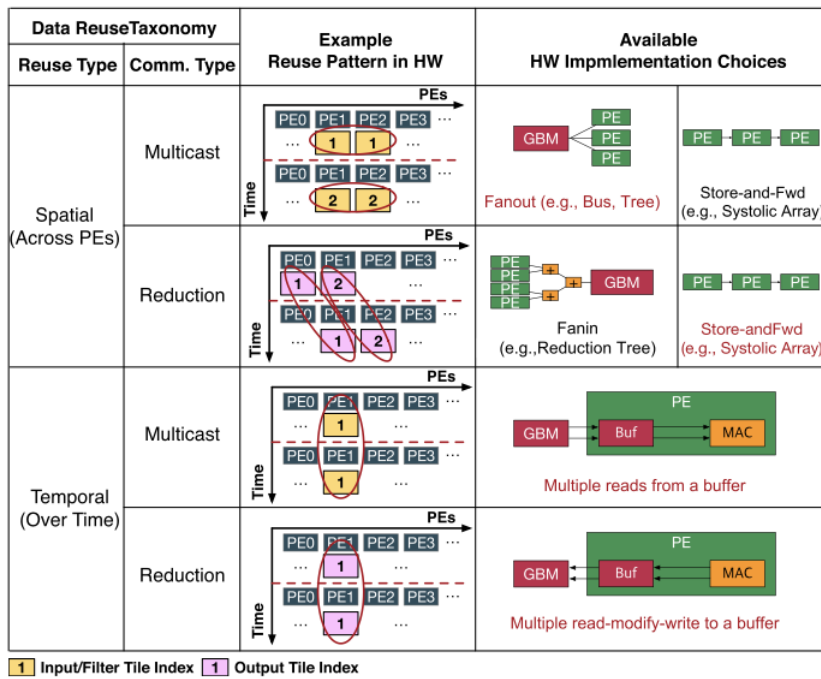


Figure 2.13: The taxonomy of data reuse types in DNN accelerators and the corresponding implementation options for each type [20].

2.4.1. Dataflow

In DNNs, data partitioning and scheduling strategies to exploit reuse and perform staging, known as dataflow [19], play a critical role in optimizing computational efficiency and accelerating model training and inference. Efficient dataflow management is essential to minimize memory accesses, reduce processing time, and maximize hardware utilization.

The dataflow of an accelerator consists of two aspects: the scheduling of DNN computations over time, leveraging a wide spectrum of reuse, and the mapping of the DNN computation across PEs for parallelism.

This representation is based on four key components [19] as follows:

- **Spatial Map(size, offset)** α defines how the dimension α of a data structure is distributed across PEs. The *size* parameter specifies the number of indices in the dimension α that are assigned to each PE, while the *offset* parameter specifies the shift of the starting indices of α across successive PEs.
- **Temporal Map(size, offset)** α specifies how the dimension α is distributed across time steps within a PE. The *size* parameter denotes the number of indices in dimension α allocated to each PE, and the *offset* parameter describes the shift of the starting indices of α across successive time steps within a PE. The chunk of dimension indices mapped remains consistent across PEs within a given time step.
- **Data Movement order** determines the order in which spatial and temporal maps are arranged in the dataflow specification. This sequence dictates how data mappings to PEs change over time.
- **Cluster(size)** enables exploration of the spatial distribution of more than one data dimension. This directive groups PEs of the *size* parameter or creates nested sub-clusters if there is more than one cluster directive.

An example of a dataflow is shown in Fig. 2.14. This figure shows the MAESTRO-like notation of these constructs. MAESTRO is a cost analysis framework discussed in section 3.2. In this example there is a convolutional layer with stride two in both dimensions X and Y and in the dimension section we have the description of the dimensions of filters and ifpam already presented in table 2.1. The dataflow section contains the directives for allocating data between PEs. The part before the cluster directive shows how data is distributed between clusters and the part shows the dataflow between PEs on the same cluster. The K dimension is distributed across clusters and the C dimension is distributed across PEs of the same cluster. The other dimensions are distributed across time steps between clusters and PEs.

There are many types of dataflows. In the next Subsections, the most common ones are presented following the Eyeriss taxonomy [5].

```

Layer Conv2d-1 {
Type: CONV
Stride { X: 2, Y: 2 }
Dimensions { K: 32, C: 3, R: 3, S: 3, Y: 224, X: 224 }
Dataflow {
    SpatialMap(1,1) K;
    TemporalMap(64,64) C;
    TemporalMap(3,3) R;
    TemporalMap(3,3) S;
    TemporalMap(3,1) Y;
    TemporalMap(3,1) X;
    Cluster(64, P);
    SpatialMap(1,1) C;
    TemporalMap(3,1) Y;
    TemporalMap(3,1) X;
    TemporalMap(3,3) R;
    TemporalMap(3,3) S;
}
}

```

Figure 2.14: Example of dataflow in MAESTRO notation.

2.4.2. Weight Stationary

In case of weight stationary dataflows, the weight of a filter remains fixed in the L1 memory of a PE to optimize filter reuse. Once a weight is retrieved from DRAM and placed in the L1 memory of a PE, that PE performs all operations associated with that particular filter weight. Specifically, a block of weights from the same filter and channel is assigned to a set of PEs and remains static. At the same time, each pixel within an ifmap belonging to the same channel is sequentially distributed to the corresponding PEs. Fig. 2.16 shows two computation steps of a weight stationary dataflow on a spatial architecture, where a 3x3 dimensional filter weight window remains static and a portion of ifmap is distributed across PEs at each time step. Consequently, the partial sums (psums) computed by each PE are spatially aggregated across these PEs. Multiple layers of weights from different filters and/or channels can be distributed over either multiple PE arrays or the same PEs. Fig. 2.15 shows the first two steps of the weight stationary dataflow. From a hardware perspective, the L1 is used to store these fixed filter weights. Since operational scheduling aims to maximize the reuse of these fixed weights, psums may not always be immediately reducible, leading to their temporary storage in a global buffer. If the buffer capacity is insufficient, the number of generated psums that can be stored simultaneously must be limited. Consequently, this constraint limits the number of filters that can be loaded onto the chip at any given time.

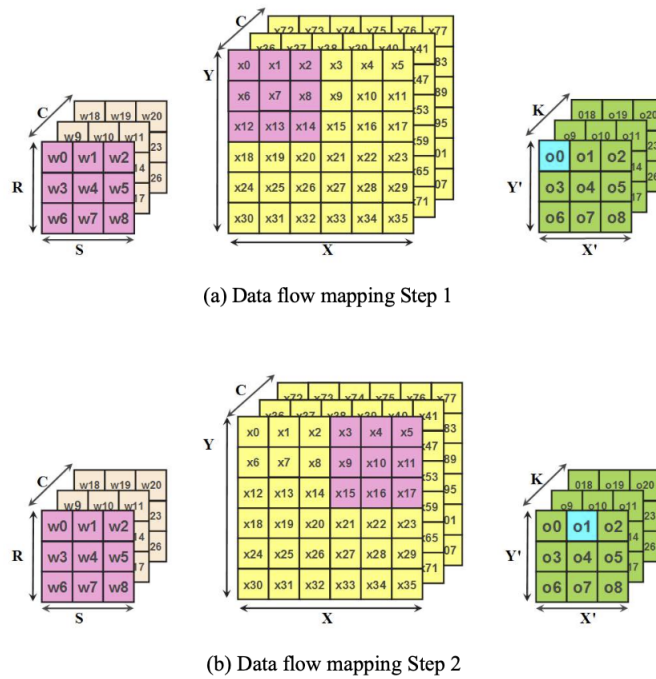


Figure 2.15: Weight Stationary dataflow steps [27].

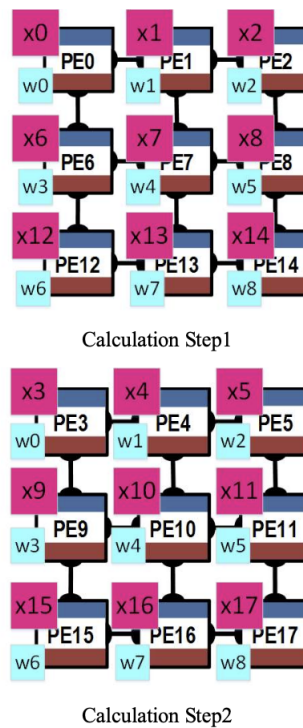


Figure 2.16: Weight Stationary computation steps [27].

2.4.3. Input Stationary

In the Input Stationary (IS) dataflow approach, each PE maintains the input activation throughout the computation. While the input activation remains constant, the weights and partial sums are dynamic. In the IS dataflow method, filter weights and accumulate psums are propagated spatially through the PE array.

2.4.4. Output Stationary

In the output stationary dataflows, the accumulation of each ofmap pixel remains fixed within a PE. The psums resulting from these accumulations are stored within the same L1 memory to minimize the cost of accumulation.

In terms of processing, this dataflow approach uses the PEs array space to simultaneously process a region of the 4D ofmap. To determine which region to process, two decisions must be made: (1) whether to process multiple ofmap channels (MOC) or a single ofmap channel (SOC), and (2) whether to process multiple ofmap plane pixels (MOP) or a single ofmap plane pixel (SOP). This results in three practical subcategories of OS (operation scheduling) data streams: SOC-MOP, MOC-MOP, and MOC-SOP.

- **SOC-MOP** primarily serves convolutional layers, focusing on processing one level of the map at a time to maximize convolutional reuse and psum accumulation.
- **MOC-MOP** deals with multiple ofmap planes and multiple pixels within the same plane simultaneously, aiming to further exploit both convolutional and ifmap reuse.
- **MOC-SOP** is tailored for fully connected layers, handling multiple ofmap channels, but only one pixel per channel at a time to emphasize ifmap reuse.

The differences between these OS data flows are shown in Fig. 2.17. All additional input data reuse occurs at the array level, i.e., inter-PE communication, while the L1 level exclusively manages psum accumulation.

In terms of hardware usage, all OS dataflows use the L1 for psum storage to achieve steady-state accumulation. In addition, SOC-MOP and MOC-MOP require additional L1 storage for ifmap buffering to fully exploit convolutional reuse within the PE array.

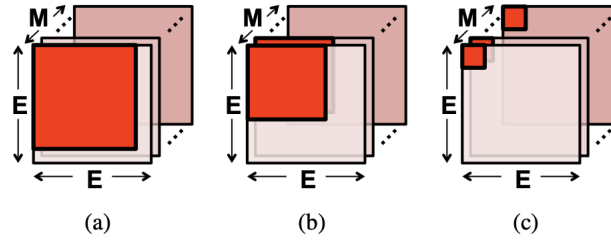


Figure 2.17: Different OS datflow variant: (a) SOC-MOP, (b) MOC-MOP, and (c) MOC-SOP [5].

2.4.5. No Local Reuse

In this context, the NLR (No Local Reuse) dataflow includes two primary characteristics:

- It does not capitalize on data reuse at the L1 memory level.
- It relies on inter-PE communication for both ifmap reuse and psum accumulation.

Regarding processing, the NLR approach partitions the PEs array into groups of PEs. PEs within the same group handle the same ifmap pixel but with varying filter weights from the identical input channel. Conversely, different PEs groups handle ifmap pixels and filter weights from distinct input channels. The resulting psums are aggregated across PEs groups within the entire array.

In terms of hardware utilization, the NLR dataflow does not require L1 memory. Because the PE array consists primarily of arithmetic logic unit (ALU) data paths, it allocates a significant amount of space for the L2 global buffer. This buffer is used to store both psums and input data for later reuse.

2.4.6. Row Stationary

The Row-Stationary (RS) dataflow [5] is a design inspired by strip-mining techniques commonly used in spatial architectures [33]. This approach ingeniously decomposes high-dimensional convolution into 1D convolution primitives that can operate simultaneously. Each primitive processes a single row of filter weights along with a corresponding row of ifmap pixels, producing a row of psums as output. These psums, which come from different primitives, are then combined to produce the final ofmap pixels.

For 1D convolution, the inputs come from the memory hierarchy, such as the global buffer or DRAM, adding an element of versatility to the process. Each primitive is carefully assigned to a single PE, ensuring that the computation of each row pair remains fixed

within the PE. This static allocation promotes convolutional reuse of both filter weights and ifmap pixels at the L1 memory level, optimizing computational efficiency.

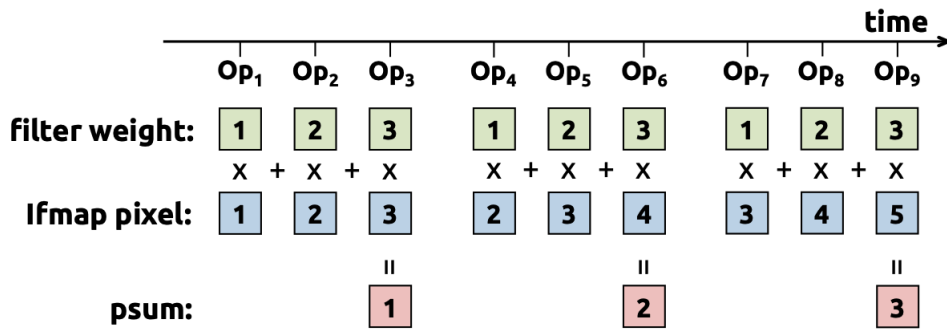


Figure 2.18: Processing of an 1D convolution primitive in the PE. In this example, $R = 3$ and $S = 5$ [5].

An illustrative example of this sliding window processing is shown in Fig. 2.18, demonstrating the synergy between computation and data flow. It's important to note, however, that orchestrating the mapping of all primitives to the PE array is not a trivial task. This mapping complexity has a significant impact on the energy efficiency of the entire system, underscoring the importance of careful planning and optimization in the RS dataflow implementation.

3 | Related Work

This Chapter describes the work related to this thesis. Section 3.1 describes GAMMA, which is a framework that optimizes mappings for DNN models on accelerators. Section 3.2 provides an overview of MAESTRO, a cost analysis framework for evaluating mappings on hardware accelerators. Section 3.4 focuses on DiGamma, which is a co-optimization framework that jointly optimizes mapping for DNNs and accelerator design, and TimeLoop, which is a framework that provides performance and energy projection on hardware architecture designs and an optimal mapping searcher.

3.1. GAMMA

GAMMA [15] (Genetic Algorithm-based Mapped for ML Accelerators) addresses the challenge of optimizing hardware mapping for DNN models on accelerators. This optimization is essential because DNN layers consist of multidimensional loops that can be ordered, tiled, and scheduled in a variety of ways that directly affect performance and efficiency.

The efficient execution of DNNs strongly depends on the hardware mapping strategy, which determines the amount of data reuse and the overall performance of the accelerator. Given fixed hardware resources but configurable mappings at compile time, the framework focuses on finding the optimal per-layer mappings for each layer of DNN models. Previous research [19, 26, 35, 39], show that there are no mapping strategies that are suitable for every layer.

To take advantage of different mappings, they consider accelerators where the number of PEs and buffer sizes are fixed at design time, but the mapping can be configured at compile time for each layer.

GAMMA uses a domain-specific genetic algorithm designed to efficiently explore the immense space of possible mappings. Unlike previous approaches that either target rigid accelerators with limited map spaces or restrict the set of mappings, GAMMA constructs a flexible and comprehensive map space. This space includes computation order, tile sizes, and parallelization strategies, allowing it to target a wide variety of fixed and flexible

single and multi-accelerator systems.

3.1.1. GAMMA algorithm

As mentioned above, GAMMA uses a genetic algorithm (GA). So many of the terms of this type of algorithm like *gene*, *genome*, *elite*, *population* are used and summarized in the table 3.1. GAs is one of the most popular algorithms for the scheduling problem due to its lightness and simplicity [6, 12, 29, 30, 36].

Term	Description
Gene	The encoded value of one of the dimensions of a design point.
Genome (Individual)	A complete series of genes representing a design point.
Elite	A set of genomes that has the highest evaluated fitness.
Population (Generation)	An entire set of genomes forms a population (one generation). The populations evolves with time by mutation/crossover and selection of the well-performing genomes to the next generation.
Crossover	Blend two parents' genes to reproduce children genomes.
Mutation	Randomly perturb a parent's genes to reproduce children genomes.

Table 3.1: Terminology in Genetic Algorithm [15].

To understand how the GA works, it is first necessary to explore the concepts of one-level and two-level mappers and how they are translated into a format suitable for the cost model, MAESTRO discussed in section 3.2. A mapper level is translated in MAESTRO as a cluster that fully describes the tiling strategy, computation order, and parallelism dimension. Thus, a one-level mapper explores only one level of parallelism, while a multi-level mapper can explore multiple levels of parallelism by concatenating clusters. It is possible to choose the parallelism based on the dimensions on filter, ifmap, or ofmap shown in the table 2.1, and the tile size of each dimension can range from 1 to the size of the dimension itself. In the algorithm, the genome is composed of seven pairs of genes (7,2) and represents a one-level mapper. Each pair contains a DNN layer dimension notation and its corresponding tile size, where the order of the pairs indicates the computational

order. The first pair of genes specifies the parallelization dimension. The parallelized dimension is associated to SpatialMap directive, and other dimensions are associated to TemporalMap directive. For a two-level mapper, the encoding is similar, with the tile size of the parallelized dimension of the L1 mapper (P_{L1}) limited by the number of available PEs. The L1 mapper represents the inner loop, while the L2 mapper represents the outer loop, containing multiple instances of the L1 mapper based on the size of the parallelized dimension and thus the number of PEs contained in a cluster. An example of decoding a 2-level genome is shown in Fig. 3.1.

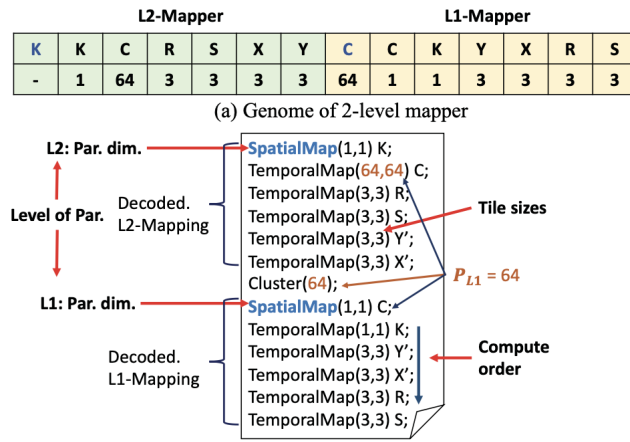


Figure 3.1: Decoded 2-level genome in MAESTRO’s description [15].

The GA works as follows:

- **Inizialization.** It involves randomly creating a population of one-level mappers, ensuring that each tile size is smaller than the corresponding layer dimension.
- **Evolution: Crossover.** It is used to combine genes from well-performing genomes by swapping the tile size values of two selected parent genomes.
- **Evolution: Mutation.** It performs two tasks:
 - changing the parallelism dimension by randomly selecting one of the six tensor dimensions
 - changing the tile sizes by assigning new random values, penalizing those that do not fit within the buffer of the processing element.
- **Evolution: Reorder.** It is a form of mutation where two paired genes swap positions, changing the mapping order.
- **Evolution: Growth.** It involves expanding the genome by adding a new one-level

genome, promoting the existing L1 mapper to an L2 mapper.

- **Evolution: Aging.** Simulates the natural shortening of DNA by trimming the tail of the genome, returning the genome to a shorter form.
- **Evaluate and Selection.** It involves evaluating the fitness of the population in an evaluation environment and selecting the fittest individuals to proceed to the next generation.

3.1.2. Flow for Automated Mapping Search

The goal of GAMMA is to find the optimal hardware mapping of a DNN layer with limited resources, especially PEs and buffers. This is a complex task because different mappings require different amounts of hardware resources, especially buffer sizes, which are considered fixed during algorithm iterations.

The objective is to minimize a chosen hardware performance metric, such as latency, power, energy, area, or energy-delay product. Minimizing this objective is complex due to interdependent factors such as tensor shape, available PEs, buffer sizes, and tile sizes.

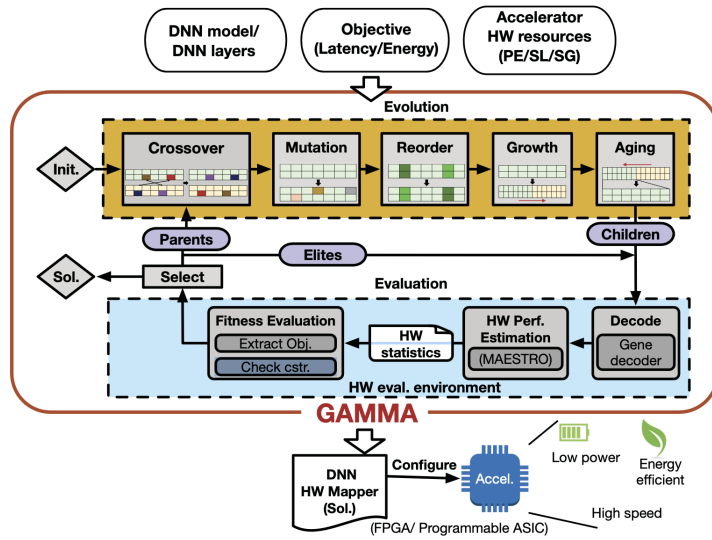


Figure 3.2: Overview of GAMMA [15].

The interactive environment (Env) is set up with the target DNN layer, hardware constraints, and optimization goals. Env uses MAESTRO to model and evaluate different mappings, collect performance statistics, and calculate fitness scores. Mappings that exceed the hardware constraints receive a large penalty, ensuring that only those mappings

that respect the constraints are considered for the target accelerators. The fitness function can be summarized as follows:

$$Fitness = \begin{cases} reward, & \text{if constraint met} \\ -Infinite, & \text{otherwise} \end{cases} \quad (3.1)$$

A complete overview of GAMMA is shown in Fig. 3.2.

3.2. MAESTRO

Accelerating DNN inference requires exploiting parallelism across many PEs for high performance and optimizing energy efficiency through effective data reuse mechanisms within PEs and on-chip scratchpads. The efficiency of a DNN accelerator depends on three key factors depicted in Figure 3.3: the workload (DNN layers), the available hardware resources, and the mapping strategy of the DNN layers to the target hardware (mapping).

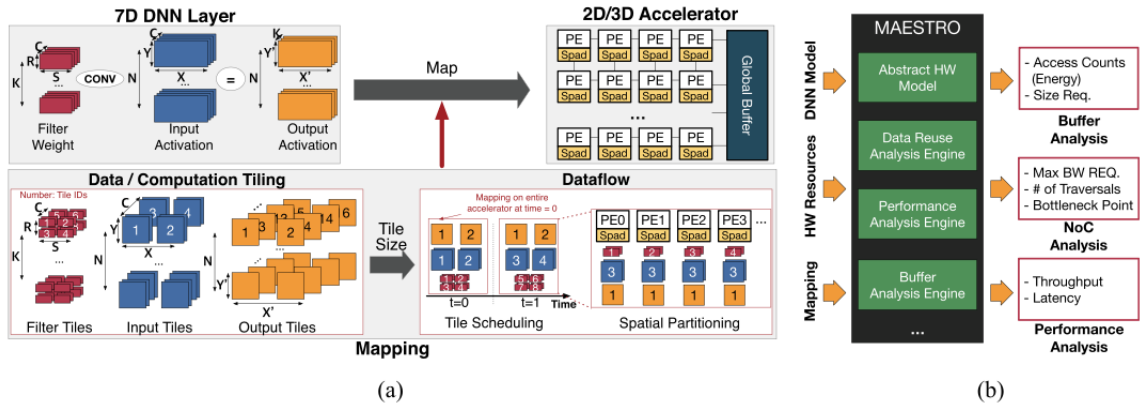


Figure 3.3: High-level overview of mapping a high-dimensional DNN layer (CONV2D in this figure) to an accelerator with 2-D PE array. (a) An Overview of Mapping CONV2D to an Accelerator. (b) High-level Tool flow of MAESTRO [20].

Efficiently predicting the performance and energy efficiency of an accelerator requires a comprehensive view of these factors. A critical aspect is the rapid estimation of efficiency due to the large design space, where a large number of candidate designs must be evaluated.

To address this, an analytical cost model has been developed as an alternative to cycle-accurate simulators. However, modeling the intricate high-dimensional design space that includes DNN layers, hardware configurations, and mapping strategies is inherently chal-

lenging. In particular, understanding the complex interaction between hardware components, mappings, and DNN layers is critical. Specifically, optimizing data reuse within the scratchpad memory hierarchy is a primary focus for improving energy efficiency in DNN accelerators. Achieving this optimization requires accurate modeling and analysis of data reuse patterns imposed by dataflow, including data/computation tile scheduling and spatial partitioning strategies as described in Fig. 3.3a. For instance, in the illustrated dataflow scenario the consistency of output tiles across time instances indicates temporal reuse, while the mapping of input tile 3 onto all PEs within a time instance exemplifies spatial reuse.

As a result, a data-centric representation of mapping is proposed to facilitate concise and compiler-friendly descriptions of potential mappings. Leveraging this representation, MAESTRO [20] is introduced, a comprehensive framework for cost-benefit analysis that is based on a systematic analysis of data reuse. As shown in Fig 3.3b, MAESTRO takes into account DNN layer characteristics, hardware specifications, and mapping strategies as input, generating over 20 estimated statistics for efficiency evaluation. Of outmost significance, MAESTRO offers rapid cost-benefit estimation, and accommodates various layer sizes and operations of state-of-the-art DNN models, ensuring versatility and applicability across different scenarios.

3.2.1. Analytical Cost Model

Utilizing the data-centric guidelines, MAESTRO was designed with all three factors - DNN layers, hardware specifications, and mapping strategies - in mind, while carefully modeling data reuse as discussed earlier. MAESTRO consists of five primary engines: Tensor, Cluster, Reuse, Performance Analysis, and Cost Analysis. Fig.3.4 shows an overview of the engine’s framework.

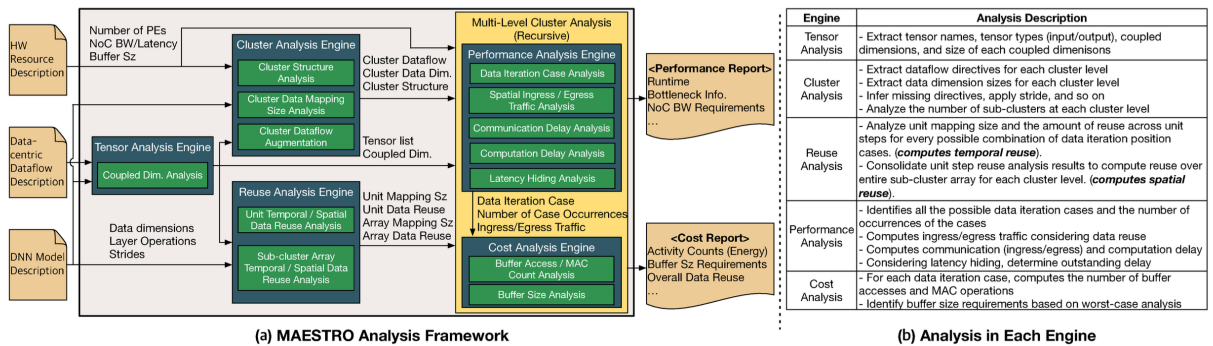


Figure 3.4: An overview of MAESTRO’s analysis framework [19].

Validation of MAESTRO’s performance model involved comparing its estimates with RTL simulation results. Specifically, the processing delay of two accelerators—MAERI and Eyeriss—when executing VGG16 and AlexNet models, respectively, was assessed. MAESTRO’s latency predictions exhibited a maximum absolute error of 3.9% compared to cycle-accurate RTL simulation and reported processing delay averages.

3.3. DiGamma

DiGamma [16] is a GA part of a HW-mapping co-optimization framework, and an efficient encoding of the design space constructed by HW and mapping as shown in Fig. 3.5. DiGamma was born as an extension of GAMMA, discussed in section 3.1. As an extension, it finds the best mapping for each layer and the best accelerator configuration for the DNN.

Instead of considering a naive optimization-based HW-mapping co-optimizer consisting of two search loops, which can lead to a design space as large as $O(10^{36})$ [16], they propose a unified co-optimization process. Note that two main factors decide the effectiveness of an optimization framework: the efficiency of the design point encoding and the efficiency of the optimization/search algorithm. This work proposes an efficient encoding method for HW configuration and mapping and a sample-efficient optimization algorithm (DiGamma).

3.3.1. Technical Approach Overview

The Co-opt Framework optimizes accelerator designs by taking inputs such as the target model, the optimization objective, the design budget, the optimization algorithm, and optionally a design constraint. Design constraints must support two additional use cases: optimizing only the mapping for a fixed hardware design, and determining optimal hardware configurations for a fixed mapping. The framework includes an optimization block, as shown in Fig. 3.6a, that abstracts the details of handling different DNN models, budgets, and encoding/decoding of hardware and mappings and provides a generic interface for different optimization algorithms. The optimization block allows the user to set a sampling budget to control the number of optimization loops.

The evaluation block consists of a decoding module and a fitness evaluation module. The fitness evaluation includes a hardware performance evaluator and a constraint checker. The hardware performance evaluator uses MAESTRO [20] to generate detailed performance reports based on the proposed design points. The Constraint Checker ensures that design items meet resource constraints and invalidates those that exceed the specified

budget by assigning them a negative fitness value.

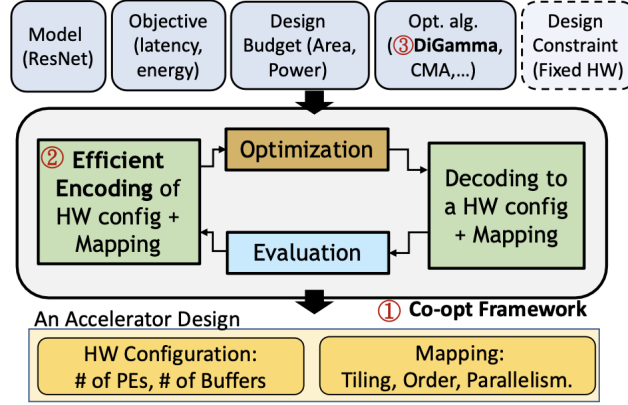


Figure 3.5: HW/Mapping Co-optimization Framework with the three technical contributions highlighted [16].

The Co-opt framework includes a custom encoding, as shown in Fig. 3.6b-c with notation shown in Fig. 3.6g, to describe accelerator design points, capture compute resources, mapping, and memory hierarchy. The notation uses key-value pairs to represent genes, where L1-config represents configurations for 1-D and L2-config shows the HW and mapping across multiple 1-D PE arrays, effectively describing a 2-D PE array. π_{L1} and π_{L2} are hardware parameters that define the size and number of PE arrays, while other genes describe mapping parameters such as tiling, order, and parallelism.

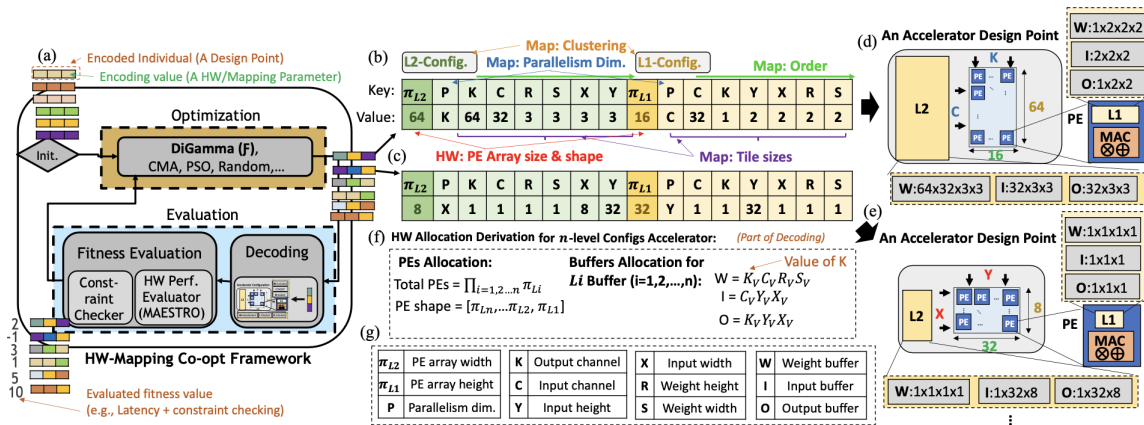


Figure 3.6: (a) Co-opt Framework, (b-c) the HW-Mapping encoding representation, and (d-e) the corresponding decoded accelerator configuration. (f) The formula for calculating minimum on-chip buffer requirement. (g) The definition of notations. [16].

For decoding, the framework translates these genes back into specific accelerator designs

to evaluate their performance. The PEs array sizes and aspect ratios are determined by the π_{L1} and π_{L2} genes, and the P values indicate how the computations are parallelized. The order of the genes determines the computation order for each tile in the PE arrays, while the tile sizes and hierarchy levels determine the buffer requirements at the L1 and L2 levels.

3.4. Other Tools

3.4.1. TimeLoop

To address the challenges of evaluating and exploring the design space of DNN accelerators, TimeLoop [26] is introduced as a comprehensive infrastructure. There are two main parts in TimeLoop: a power, area, and energy projection model, and a mapper that constructs and explores the map space based on a workload given on the target architecture.

For analysis, users describe the organization of an architecture using a highly configurable template that includes abstractions for compute units, storage in a multi-tiered hierarchy, and communication links. Additional architectural constraints such as dataflow and hardware attributes such as utilization and bandwidth can also be specified to define the map space. Given a workload and architecture specification, TimeLoop’s mapper systematically constructs the mapspace, evaluates the quality of each mapping using its cost model, and iteratively searches for an optimal mapping. The efficiency of this search is due to the speed and accuracy of the model, which has been validated against existing designs.

TimeLoop’s key contributions are as follows:

- Provides a unified and concise method for describing the key attributes of different DNN architectures, facilitating the exploration of a wide range of designs.
- Combines the exploration of a large design space with a mapper that optimally maps each workload to the target architecture, enabling fair comparisons and systematic design of DNN accelerators.
- Enable studies that provide insights into current DNN accelerator architectures, highlight their strengths and weaknesses, and inspire future research.

4 | Target Architecture

This Chapter describes the target architecture considered during the thesis. Section 4.1 presents an overview of the chosen architecture.

4.1. Target Architecture Overview

We chose as our target architecture a spatial architecture consisting of an array of PEs, each equipped with an L1 scratchpad memory and a shared L2 scratchpad global buffer. The architecture considers the possibility of having different levels of PEs clusterization. An example of this architecture is shown in Fig.4.1.

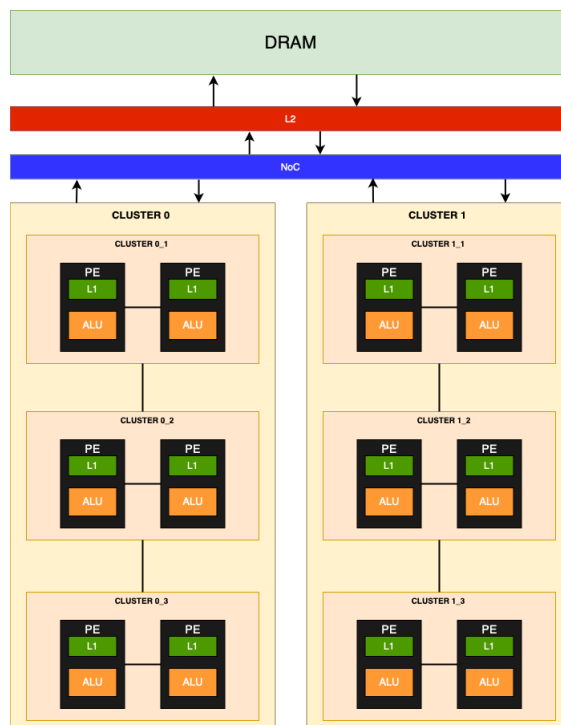


Figure 4.1: Target architecture of our research.

4.1.1. Hardware model of a PE

Fig. 4.2 shows a simplified block diagram of the PE considered in this work. The high-level model of the architectural template is derived from the description of state-of-the-art implementations [1, 9, 21]. As in [21], the model considers two decoupled pipelines for the floating-point and fixed-point compute engines. When both are available, this configuration allows optimizing each pipeline differently, increasing the system’s energy efficiency at the cost of some area overhead.

Our experiments consider the PE’s input and output ports of 32 bits. For the floating-point pipeline, we model the possibility of implementing instructions using FP32, FP16, FP8 [3, 18], and FP4 [31]. For the fixed-point pipeline, we provide models for the INT32, INT16, INT8, INT4, INT2 data types. The precision taken into account is the one used for the calculation of both the filter and the activations, therefore zero padding is used for values with a smaller number of bits than the standard configuration. In the FP32 (INT32) mode, the engine performs a fused multiply-accumulate instruction (MAC). When using a smaller quantization Q , the 32-bit lane is partitioned in $P = 32/Q$ operands, and the engine performs a $\sum_P A_i \cdot B_i + C$ instruction.

We consider the multipliers of the floating and fixed point engines to be implemented with different numbers of pipeline stages [21]. As the targeted algorithms are significantly parallel, we can schedule threads to minimize the influence of data dependency and achieve, ideally, a throughput of one instruction per cycle [7].

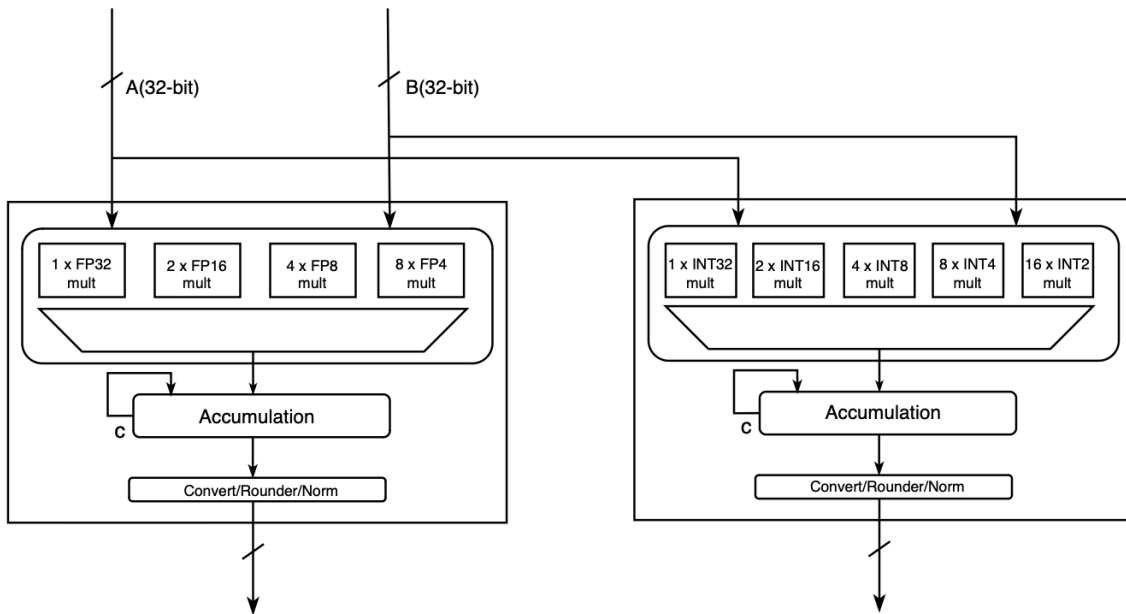


Figure 4.2: Simplified block diagram of the configurable PE.

The template components are automatically instantiated depending on the precision requirements of the use-cases targeted during the design-space exploration, together with the proper accumulator data width and output results packing. We have extracted and derived energy and performance values for the elements of the functional unit from the implementation results published in [3, 14, 17, 21, 31], scaled to 22nm technology and shown in Table 4.1. The on-chip interconnection was modeled on implementation results extracted from [4]. Energy and area values for the several memory elements were generated with CACTI-D [32], scaling the obtained results from a 32-nm to a 22-nm technology.

MAC (pJ/op)					
Tech(nm)	INT32	INT16	INT8	INT4	INT2
22	2.221	1.299	0.826	0.608	0.424

MAC (pJ/op)				
Tech(nm)	FP32	FP16	FP8	FP4
22	10.752	5.024	2.433	1.397

Table 4.1: Values of MAC energy based on different quantization in 22nm architecture.

5 | Proposed Methodology

In this Chapter, we discuss how we redesigned the open-source frameworks GAMMA and MAESTRO to allow the optimized execution of mixed-precision quantization neural networks. The section 5.1 gives an overview of the work done on the frameworks. Sections 5.2 and 5.3 discuss the developed methodology and the re-design performed on GAMMA and MAESTRO, respectively. We have named the new versions of the tools qGamma (quantized GAMMA) and qMaestro (quantized MAESTRO).

5.1. High-level Overview

Fig. 5.1 shows the new design of qGamma and qMaestro. The boxes in light red are the ones that were modified during the implementation. The changes made to obtain qGamma are as follows:

- Increase the genome size to permit the support of the reconfigurable PE based on the quantization.
- Create a new dataflow description. We made some modifications in the dataflow representation, moving from output-centric dataflow to input-centric dataflow and allowing different strides to be supported.
- Add heuristics to the GA to reduce the search space. Increasing the genome size significantly increases the search space, and introducing heuristics limits the search space by excluding inefficient dataflows.

The changes made to obtain qMaestro are as follows:

- Integrate the configurable accelerator model into the framework, providing a configurable PE based on the precision described in the target architecture. As part of the integration of the new accelerator model, there was an expansion of the dataflows accepted by the model.
- Propose a new activity-based energy model with memory energy access values calculated at runtime based on the sizes found by the framework. The correct MAC

energy value is selected based on the quantization and the NoC is computed considering the quantization of the activations and filters.

A more detailed explanation can be found in Section 5.2 for qGamma and in Section 5.3 for qMaestro.

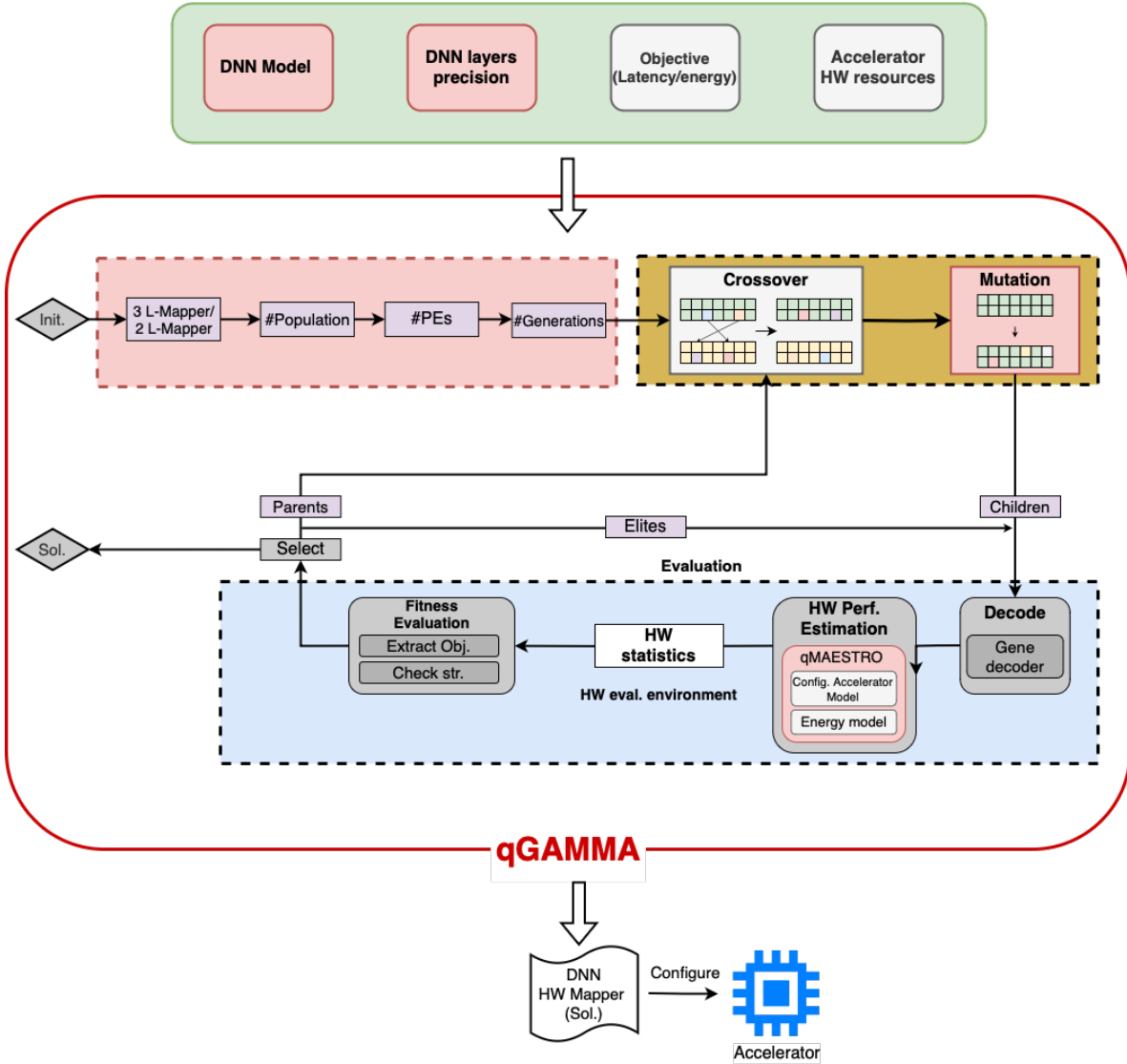


Figure 5.1: qGamma and qMaestro overview.

5.2. qGamma

This Section describes the methodology developed and the modifications made to GAMMA. Subsection 5.2.1 describes the work done on the genome to extend it to the new architecture model. Subsection 5.2.2 describes the new input model in qGamma and Subsection

5.2.3 provides a description of the heuristics used to reduce the search space of the exploration.

5.2.1. Genome Extension

To enable mixed-precision quantization, we model a configurable PE with different FUs based on the precision of the filters and activations, as described in Subsection 4.1.1. To enable dataflows suitable for our target architecture, we add another level of parallelism, passing from a 2-level mapper to a 3-level (L3) mapper. In the L3 mapper, the genome is composed of 3 blocks of (7,2) genes, where the first block represents the dataflow for clusters, the second block represents the dataflow between PEs within clusters, and the third block represents the dataflow between FUs. Fig. 5.2 shows an example of a 3-dimensional mapped genome. This new genome must satisfy the rule

$$P_{L1} * P_{L2} \leq \#PEs$$

so that the multiplication between the first level of clustering and the second level must be less than the number of PEs, this to avoid using more PEs than the one given in the accelerator specification.

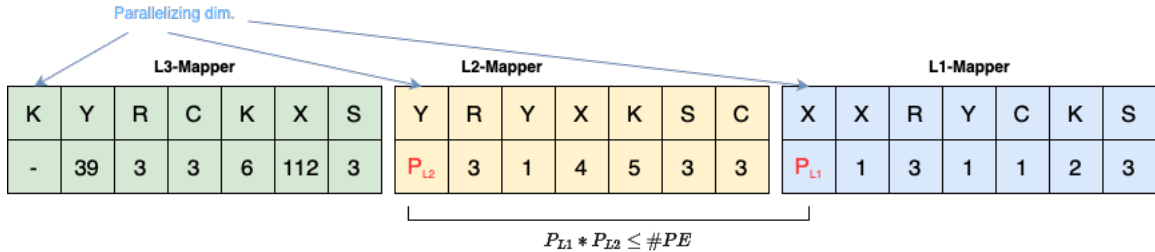


Figure 5.2: Example of 3 level-mapper genome.

The cluster dimension of the inner level is fixed based on the quantization and corresponds to the number of FUs in the PE. Instead, the outermost cluster can be explorable if needed. It is even possible to use qGamma in an architecture where the clustering is already fixed and the number of elements in the clusters is not explorable.

The L3 mapper is needed for any quantization other than the standard one, which covers the FP32 and INT32 cases. For these cases, the algorithm instantiates an L2 mapper because the target architecture is the one already proposed in MAESTRO. Increasing the dimension of the genome means even increasing the search space for exploration. For this reason, when the L3 mapper is instantiated, the population is doubled concerning the L2

mapper. Increasing the population means increasing the execution time of the algorithm, so we implement an early stop. If the best solution found remains exactly the same for a certain number of generations, the GA stops its execution before the end.

5.2.2. Dataflow Description

Extending the genome and implementing an L3 mapper is not enough to represent the target architecture. The innermost level of the genome represents the dataflow between FUs. For this level, we exclude parallelization in K because it means merging partial sums of different output feature maps according to our hardware PE model described in Subsection 4.1.1. This behavior produces incorrect results and implies that K is not considered for parallelization at the inner level. Considering the dataflow, we even changed the representation from output-centric to input-centric to better represent the new model architecture, and we extended the support of different strides in the input model file.

5.2.3. GA Heuristics

Increasing the genome size significantly increases the search space of exploration. Therefore, we decided to add some heuristics to limit the search space by excluding inefficient dataflows. Considering that an accelerator typically has a power of two PEs, we have limited the number of PEs within a cluster to a power of two. This allows clusters to include all available PEs. When the number of PEs in a cluster is fixed, exploration is not performed by the algorithm.

An L3 mapper parallelizes over three dimensions. Dimensions with more elements are more parallelizable because they can be better distributed among PEs. Therefore, for the first level, the three dimensions with the highest number of elements are considered for parallelization; for the second level, the two highest dimensions are considered and selected based on the dataflow of the previous level. For the third level, no heuristic is considered because the space is already reduced without considering the K dimension to respect the architectural model.

Another heuristic added during genome generation is to map the K and C dimensions to a power of two. K and C are typically a power of two for most layers, and dividing these dimensions into blocks of power two allows the dimensions to be evenly distributed between clusters and between PEs.

Figure 5.3 shows the heuristics used to create two different genomes, one with explorable clusters and one with fixed clusters.

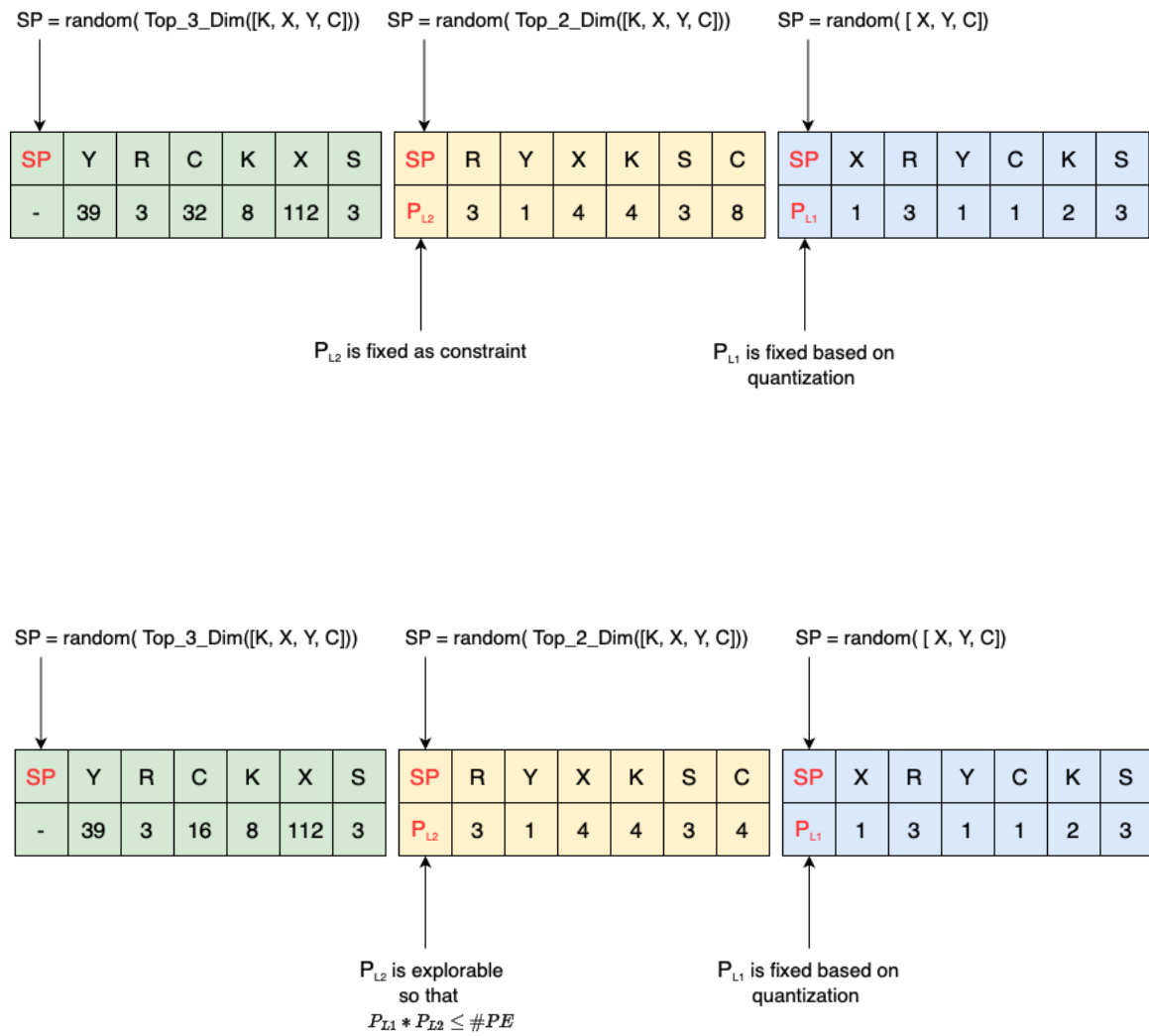


Figure 5.3: Example of genomes with and without explorable clusterization.

When the algorithm starts, the population is generated randomly, respecting the constraint and heuristic just provided, and the candidates are evaluated. Then the population is subjected to crossover and mutation. For the crossover, no modification is needed because the genes are selected from the parents and they are modified based on the parallelization level without changing the dimension parallelized. As shown in Fig. 5.4, the crossover is done in blocks of seven chunks, avoiding the first element of the block maintaining also the same clustering.

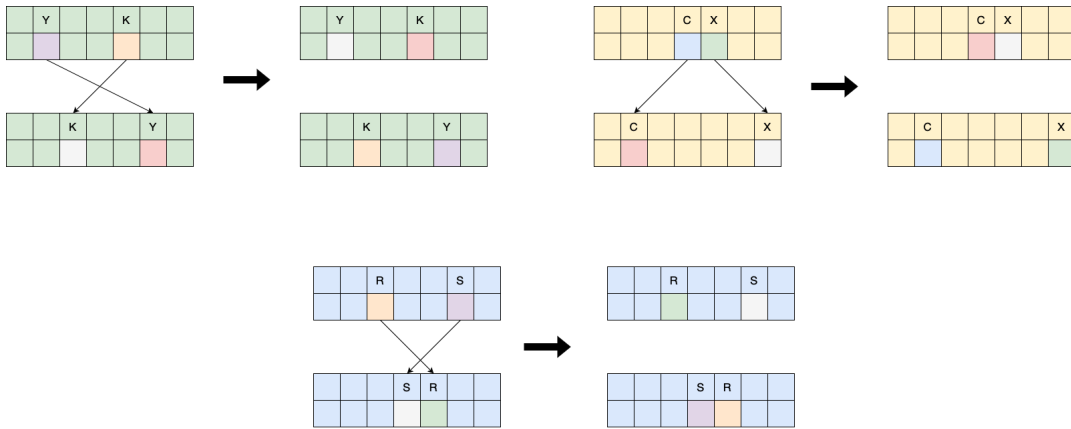


Figure 5.4: Example of the cross tile algorithm with a cross over on two dimensions in every mapper.

Instead, we adapt the mutation function to follow the clustering heuristic and avoid parallelizing to K at the last level of the mapper. The innermost cluster remains fixed even during mutation based on quantization according to the target architecture model.

To evaluate the effectiveness of the proposed heuristics, we explored the dataflows in the case of ResNetV1, detailed in Subsection 6.1.2, by comparing the exploration time of the baseline without heuristics with our implementation with heuristics. Using heuristics, we observed an exploration time reduction of 19%. Excluding the early termination condition, the reduction is about 36%.

5.3. qMaestro

This Section describes the methodology developed and the modifications made to MAESTRO. Subsection 5.3.1 describes the new architecture supported by qMaestro and Subsection 5.3.2 describes the new energy model used.

5.3.1. Configurable Accelerator Model

To implement mixed-precision quantization we reflected the changes made in qGamma to qMaestro. MAESTRO already implements three-level clustering so we start including the precision for every layer.

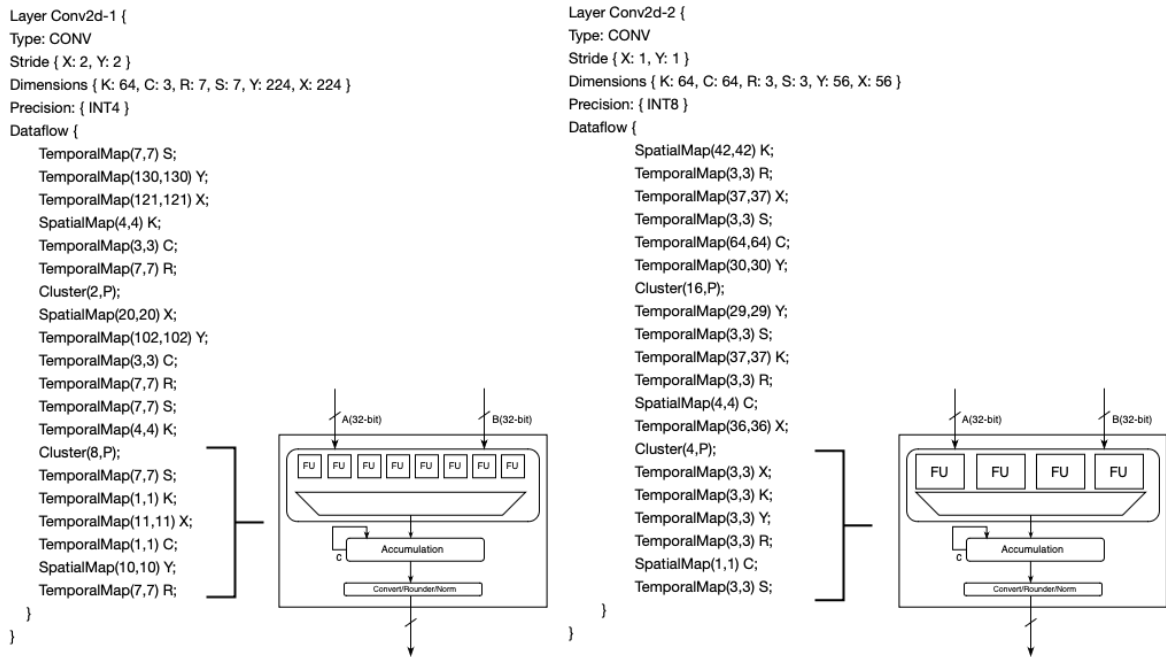


Figure 5.5: Example of dataflow file supported by qMaestro.

Have the possibility to change the precision in the same neural network means change the accelerator configuration at runtime. Therefore for each layer the accelerator is configured based on the target architecture. Figure 5.5 shows two examples of layers that computes INT4 and INT8 operations. Inside the layer specification there is now the precision, which defines the quantization of the activations and the filters. The layer body also contains the dataflow. The *SpatialMap* directive specifies how a dimension is distributed across PEs, the *TemporalMap* directive specifies how a dimension is distributed across time steps within a PE, and the *Cluster* directive groups PEs or creates nested subclusters if there is more than one cluster directive. The last part of the dataflow represents the configurable PE model and its assigned dataflow.

As part of the integration of the new accelerator model, there was an expansion of the dataflows accepted by the model, in particular for the input-centric representation now supported in qGamma, and we added support for different strides within the architecture.

5.3.2. Activity-Based Energy Model

qMaestro also implements a new activity-based energy model that takes into account memory access energy, MAC energy, and NoC energy.

The memory access energy for L1 and L2 memory is calculated based on the memory size calculated by the framework at runtime. Energy for the several memory elements were generated with CACTI-D [32], scaling the obtained results from a 32-nm to a 22-nm technology.

The MACs energy and are also calculated taking into account the precision for every layer. We have extracted and derived energy and performance values for the elements of the functional unit from the implementation results published in [3, 14, 17, 21, 31], scaled to 22nm technology and shown in Table 4.1 in Section 4.1.1.

The NoC energy is also calculate at runtime based on the quantization of the activation and filters. The on-chip interconnection was modeled on implementation results extracted from [4].

6 | Experimental Results

This Chapter describes the experimental setup used to perform the experiments and the results obtained. Section 6.1 describes the accelerator configuration used and also discusses the CNN models utilized as use cases. The Section 6.2 presents the experimental results.

6.1. Experiment Setup

This Section describes the experimental setup for the experiments, including the accelerator configuration, the CNN benchmarks used, and the metrics for evaluating the results.

6.1.1. Accelerator configuration and Setup

For running the experiments we used a MacBook Pro 2021 with 16GB of RAM and M1Pro Chip. In every experiment, we used qGamma and qMAESTRO to find and evaluate the best mapping for the CNNs. To simulate an on-the-edge DNN accelerator, we opted to run the simulation on an accelerator with 16 PEs, 512B of L1 memory maximum capacity, and 512KB of L2 memory maximum capacity.

We used qGamma for two different optimizations. The first is a per-layer optimization, in which the clusterization is explored for each layer and can vary based on the layer characteristics. In the second, we consider a clusterization that fits all layers since the clusterization is a hardware configuration and cannot be changed at runtime.

In the first optimization, we ran qGamma for 1000 generations with a population of 150 individuals for the INT32 and FP32 quantizations that require an L2 mapper, and with a population of 300 individuals for the other quantizations that require an L3 mapper, since this significantly increases the search space exploration.

In the second optimization, to find the best clusterization to use for the network we adopted the following procedure:

- Run qGamma for 1000 generations with a population of 150 individuals for the

INT32 and FP32 quantizations that implement an L2 mapper, and with a population of 300 individuals for the other quantizations that implement an L3 mapper.

- Select the most used cluster in the CNN to get an optimal result.
- Run qGamma for 1000 generations with the optimal fixed cluster found.

We then evaluate the resulting dataflow based on the evaluation metrics described in the 6.1.3 Subsection.

6.1.2. Benchmarks

To evaluate the work done in this thesis, we consider three CNNs: a MobileNetV1 [13] and a ResNetV1 [10] taken from the MLPerf Tiny benchmark, and a ResNet18 [11] taken from [38]. MLPerf Tiny is an open source benchmark suite for TinyML systems [2]. Its goal is to provide a representative set of deep neural networks and benchmarking code to compare the performance of embedded devices, including microcontrollers, DSPs, and tiny NN accelerators. The MLPerf Tiny Inference Benchmark Suite provides a set of four standard benchmarks selected by more than 50 organizations in academia and industry. These benchmarks are available for download at [34].

The tables 6.1, 6.2, and 6.3 show the overview of the CNNs. In these tables, it is possible to see the characteristics and dimensions of each layer of the CNNs. The notation used is the same one introduced in Subsection 2.1.4.

ResNetV1						
Layer	K	C	Y	X	R	S
1	16	3	32	32	3	3
2	16	16	32	32	3	3
3	16	16	32	32	3	3
4	32	16	32	32	3	3
5	32	32	16	16	3	3
6	32	16	32	32	1	1
7	64	32	16	16	3	3
8	64	64	8	8	3	3
9	64	32	16	16	1	1

Table 6.1: Layers' characteristics for ResNetV1

MobileNetV1						
Layer	K	C	Y	X	R	S
1	8	3	96	96	3	3
2	1	8	48	48	3	3
3	16	8	48	48	1	1
4	1	16	48	48	3	3
5	32	16	24	24	1	1
6	1	32	24	24	3	3
7	32	32	24	24	1	1
8	1	32	24	24	3	3
9	64	32	12	12	1	1
10	1	64	12	12	3	3
11	64	64	12	12	1	1
12	1	64	12	12	3	3
13	128	64	6	6	1	1
14	1	128	6	6	3	3
15	128	128	6	6	1	1
16	1	128	6	6	3	3
17	128	128	6	6	1	1
18	1	128	6	6	3	3
19	128	128	6	6	1	1
20	1	128	6	6	3	3
21	128	128	6	6	1	1
22	1	128	6	6	3	3
23	128	128	6	6	1	1
24	1	128	6	6	3	3
25	256	128	3	3	1	1
26	1	256	3	3	3	3
27	256	256	3	3	1	1

Table 6.2: Layers' characteristics for MobileNetV1

ResNet18						
Layer	K	C	Y	X	R	S
1	64	3	224	224	7	7
2	64	64	56	56	3	3
3	64	64	56	56	3	3
4	64	64	56	56	3	3
5	64	64	56	56	3	3
6	128	64	56	56	3	3
7	128	128	28	28	3	3
8	128	64	56	56	1	1
9	128	128	28	28	3	3
10	128	128	28	28	3	3
11	256	128	28	28	3	3
12	256	256	14	14	3	3
13	256	128	28	28	1	1
14	256	256	14	14	3	3
15	256	256	14	14	3	3
16	512	256	14	14	3	3
17	512	512	7	7	3	3
18	512	256	14	14	1	1
19	512	512	7	7	3	3

Table 6.3: Layers' characteristics for ResNet18

6.1.3. Evaluation Metrics

qGamma has the ability to optimize up to two metrics per execution. In our experiments, we used the following metrics:

- **Latency [Cycles]**, which is the number of cycles required to execute the layer.
- **Activt-Based Energy [nJ]**, which is the sum of the contributions from memory energy, MACs energy, and NoC energy.
- **Energy-Delay-Product (EDP)**, which is the product of the Activt-Based Energy [nJ] and the latency [Cycles].

EDP gives equal importance to both energy consumption and execution time. Any increase in either energy or delay will result in a higher EDP, making lower EDP values more desirable. The Energy-Delay Product illustrates the ability of a design to achieve an optimal balance between performance and execution time.

In Section 6.2, we introduced the Improvement Ratio (Imp. Ratio) to evaluate experiment results. The Imp. Ratio is calculated as the ratio of the metric obtained for the quantization with the highest bit precision to the metric associated with the quantization under consideration:

$$Improvement_Ratio = \frac{highest_bit_quantization_metric_value}{considered_quantizations_metric_value}$$

6.2. Results

This Section presents the results obtained during the experiments. Subsection 6.2.1 presents the results on ResNetV1 optimized for EDP. Subsection 6.2.2 describes the results obtained for the MobileNetV1 optimized for activity-based energy and Subsection 6.2.3 presents the results optimized for latency for the ResNet18.

6.2.1. Results on ResNetV1 optimized for EDP

The goal of this experiment is to compare ResNetV1 on three different quantizations, FP8, FP16, and FP32, applied homogeneously to all layers, all optimized by EDP. In this experiment, we consider the clustering explorable for each layer. Fig. 6.1 shows the EDP per layer. The Y-axis on the left shows the EDP, while the one on the right shows the number of MACs for each layer. The Figure shows how the EDP is affected by the

number of MACs. The higher the number of MACs, the higher the EDP. The Figure also shows how lower quantization affects the EDP, which is lower at lower bit quantization. The Imp. Ratio from FP32 to FP16 is 6.95x and the improvement from FP32 to FP8 is 21.44x. Fig. 6.2 shows the latency per layer, where the Y-axis represents the number of cycles executed. The latency improvement from FP32 to FP16 is 3.0x, and from FP32 to FP8 is 4.29x. Theoretically, the maximum latency improvement should be 2x from FP32 to FP16 and 4x from FP32 to FP8. However, since the framework optimizes for EDP, it may prioritize decreasing latency to detriment of the energy. In addition, the genetic algorithm may converge to suboptimal solutions that exploit a quantization level better than another for a given layer.

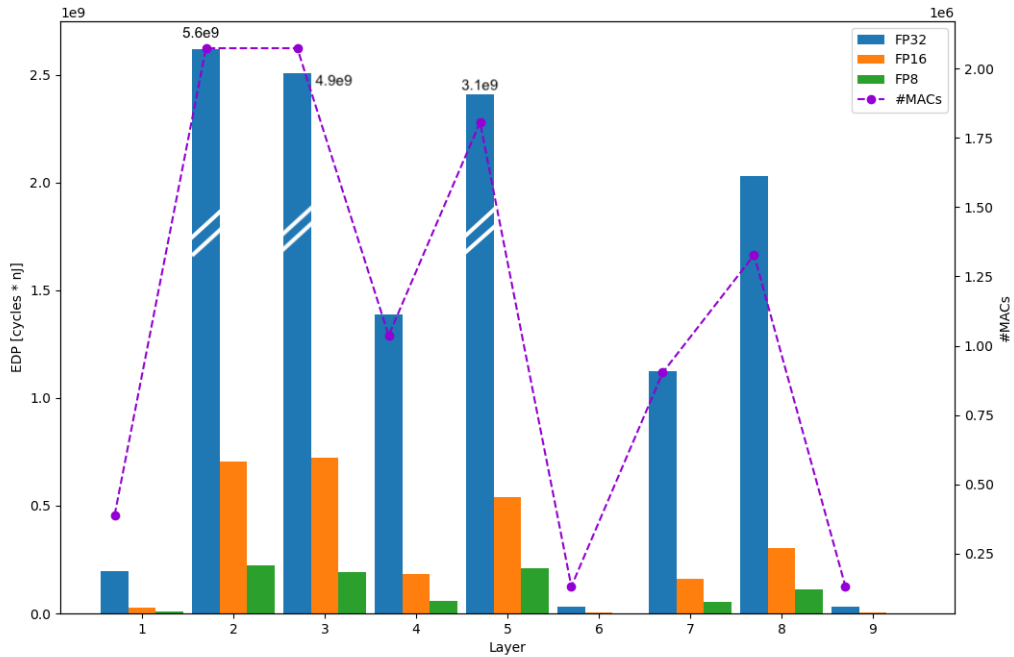


Figure 6.1: EDP results for ResNetV1. Number of MACs per-layer also reported.

Fig. 6.3 shows the total activity-based energy normalized to the MAC energy of the highest bit quantization. The improvement in activity-based energy is 2.32x from FP32 to FP16 and 4.94x from FP32 to FP8. The Figure shows that most of the energy is consumed by MACs due to the high pipeline cost, and that the FP32 MAC energy is more than twice that of FP16 and more than four times that of FP8.

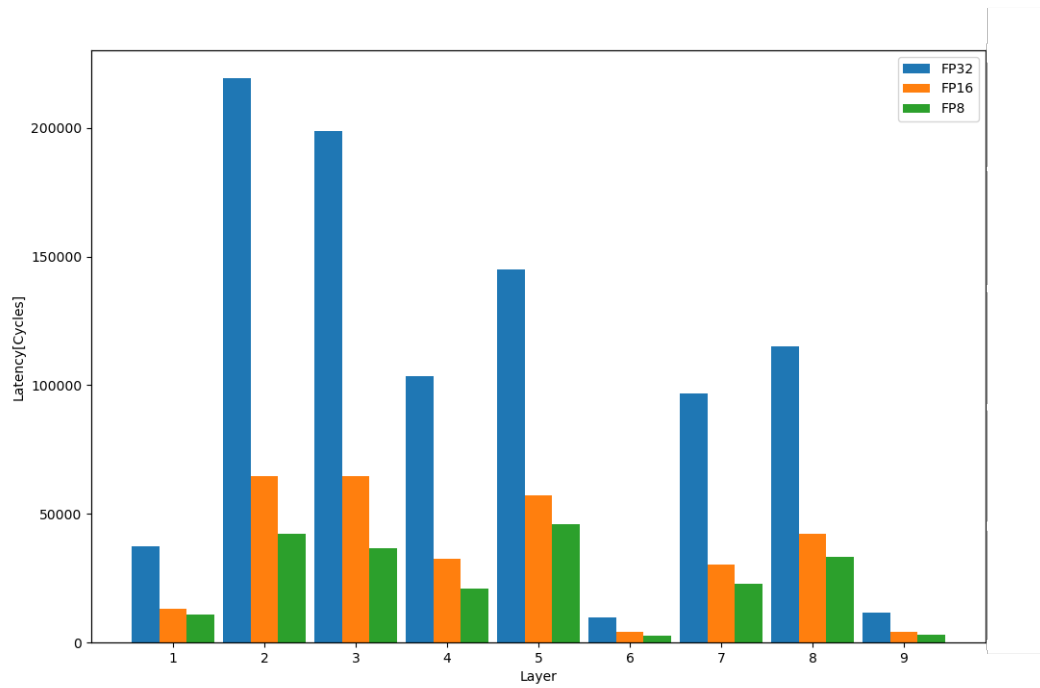


Figure 6.2: Latency results for ResNetV1.

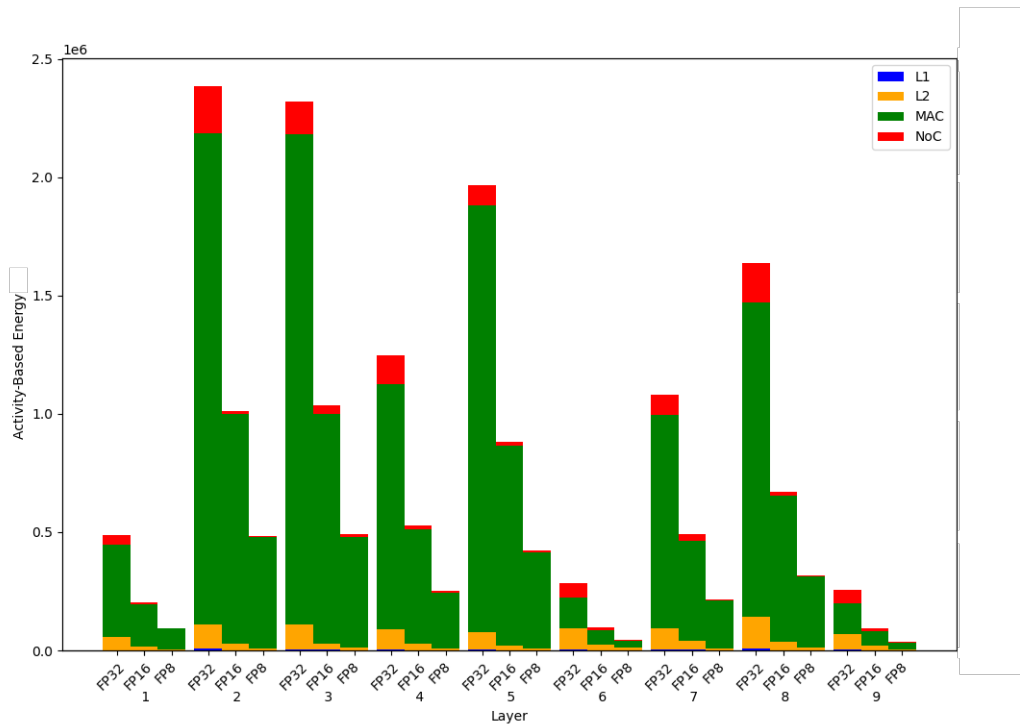


Figure 6.3: Activity-based energy results for ResNetV1.

Lower quantization levels result in lower values for both latency and energy because the EDP balances energy consumption and execution time, giving equal importance to both metrics.

6.2.2. Results on MobileNetV1 optimized for energy

This experiment compares MobileNetV1 on three different fixed-point quantizations, INT4, INT8, and INT16, applied homogeneously to all layers, all optimized by activity-based energy. In this experiment, we consider the clustering fixed for all layers. Fig. 6.4 shows the energy per layer and the quantizations. The Y-axis on the left shows the total energy normalized over the MAC energy of the quantization with the highest bit quantization considered, while the one on the right shows the number of MACs for each layer. The activity-based energy is divided into components: the blue is the L1 energy, the yellow is the L2 energy, the green is the MACs energy, and the red is the NoC energy. From an energy consumption perspective, L1 memory has minimal impact due to its smaller size, resulting in lower access energy, while L2 memory, being larger, contributes more significantly to the overall energy consumption. L1, L2, and NoC energy vary significantly based on quantization because the size of filters and activations affect memory accesses and bandwidth utilization. The Figure shows that the lower the quantization, the lower the energy consumption, and that the energy varies with the MAC number. The Imp. Ratio from INT16 to INT8 is of 1.96x and the improvement from INT16 to INT4 is of 3.033x.

Fig. 6.5 shows the latency per layer of the MobilNetV1. The Y-axis shows the number of cycles achieved per layer. The Figure shows how, if the optimization does not take latency into account, there could be higher latency values for lower quantizations. In this optimization, only the activity-based energy is considered, and the framework chooses the optimal activity-based energy even if the latency is not optimal.

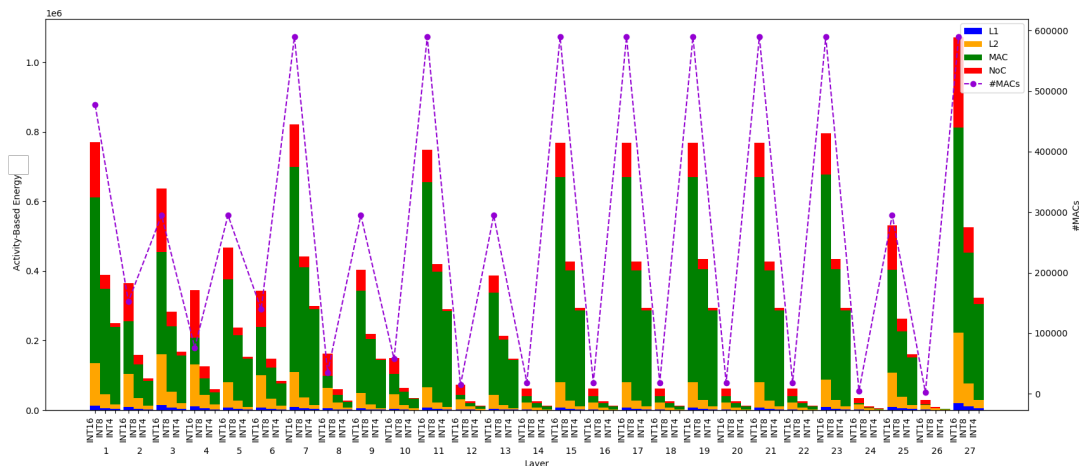


Figure 6.4: Activity-based energy results for MobilenetV1. Number of MACs per-layer also reported.

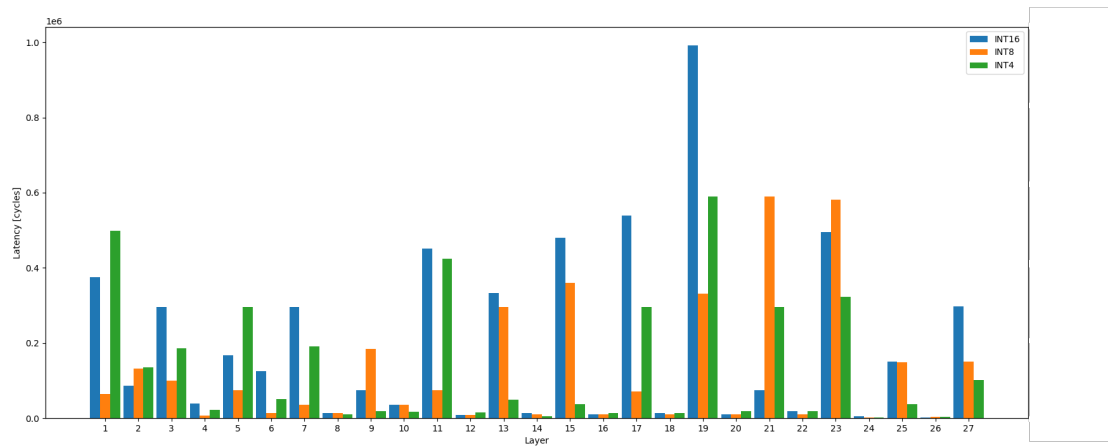


Figure 6.5: Latency results for MobilenetV1.

6.2.3. Results on ResNet18 optimized by latency

This experiment analyzes a ResNet18 in a mixed-precision configuration of INT8 and INT4 layers taken from [38]. In [38], they presented a mixed-precision integer-only quantization framework and proved its ability to improve performance while preserving accuracy. As a comparison for this mixed-precision configuration, they choose an INT8 single-quantization configuration that has similar accuracy.

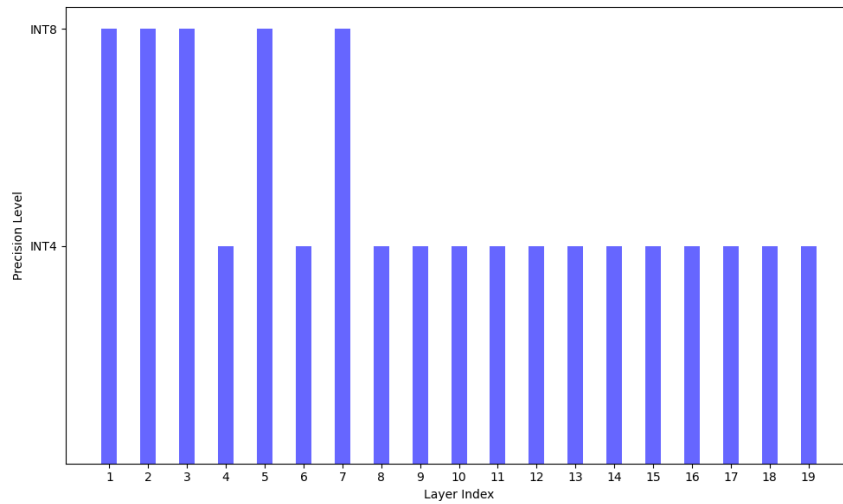


Figure 6.6: Quantization per layer in mixed-precision configuration.

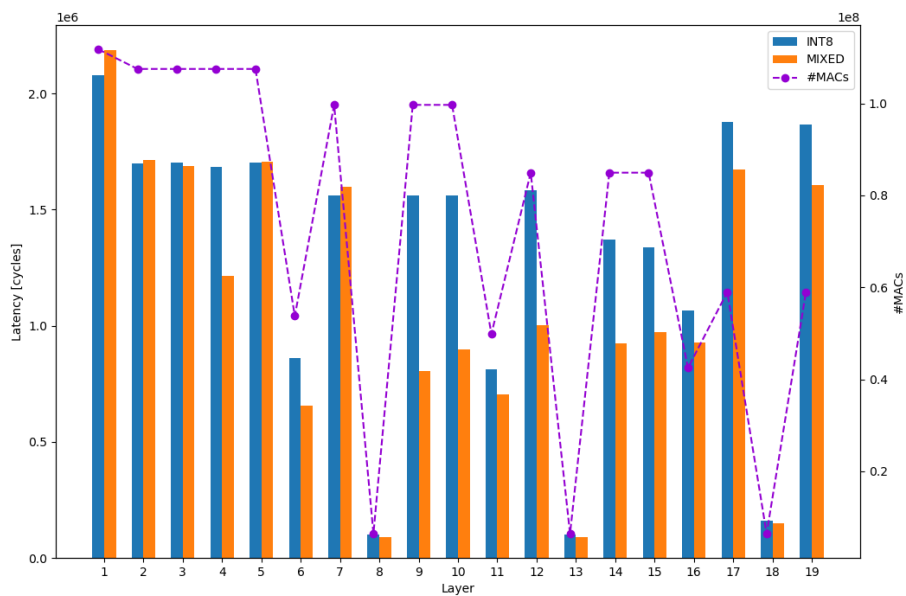


Figure 6.7: Latency results for ResNet18. Number of MACs per-layer also reported.

Both configurations are optimized for latency. In this experiment, we consider the clustering fixed for all layers. Fig. 6.6 shows in detail the precision of each layer in the case of mixed-precision quantization.

Fig. 6.7 shows the latency per layer. The Y-axis on the left shows the latency expressed in cycles, while the Y-axis on the right shows the number of MACs for each layer. The Figure shows how latency is affected by MACs and quantizations. In the layers where the quantization is INT4, latency is reduced than the corresponding INT8 quantization. The slight difference in latency results for INT8 between runs on the same layer is due to the nature of the genetic algorithm, which can for some layers converge on different suboptimal solutions, resulting in performance differences. The improvement ratio from INT8 single-quantization to mixed-precision quantization is of 1.20x.

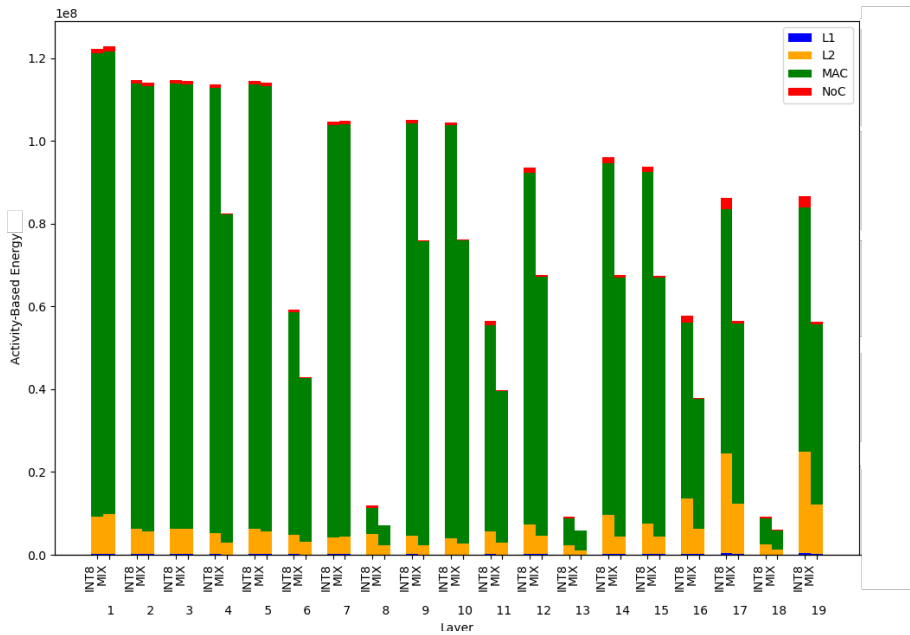


Figure 6.8: Activity-based energy results for ResNet18.

Fig. 6.8 shows the activity-based energy per layer. The Y-axis shows the total energy normalized to the MAC energy of the quantization with the highest bit quantization considered. It can be seen that even with latency optimization, the activity-based energy will be lower at lower quantizations. The largest component in the total energy is the MAC energy given by the high number of MACs performed by this CNN. The improvement ratio from INT8 single-quantization to mixed-precision quantization is of 1.23x.

7 | Conclusions and future developments

Given the need to always optimize the performance in DNN and accelerators, this thesis proposes qGamma, a framework that provides through a genetic algorithm a mixed-precision mapper for DNN accelerators.

In Chapter 3, we described GAMMA and MAESTRO, which we used to create our frameworks. GAMMA is a framework that uses a genetic algorithm to find the optimal mapping based on the metrics to be optimized, and MAESTRO is a cost analysis framework needed to evaluate the mappings on the accelerators.

In Chapter 4, we proposed the target architecture considered. This architecture is a spatial architecture consisting of an array of PEs, where each PE is equipped with an L1 scratchpad memory and a global L2 buffer shared among the PEs. It also defined the PE structure with the different pipelines based on quantization.

In Chapter 5, we proposed our methodology to modify the mentioned tools and create qGamma and qMAESTRO.

In Chapter 6 we exposed the results running the framework with different quantization and with different CNNs.

In conclusion, our framework has demonstrated that finding the optimal mapping is crucial for improving the efficiency of mixed-precision quantization DNNs on hardware accelerators. By optimizing latency, energy consumption, and energy-delay product, it ensures better utilization of hardware resources and better performance. The effectiveness of this framework is validated by the experimental results presented.

Future developments may include the following:

- Extending DiGamma [16] to mixed-precision quantization. DiGamma, discussed in Section 3.3, co-optimizes mapping and accelerator design for a given DNN. This extension will allow DiGamma to explore mixed-precision mappings and co-design

mixed-precision accelerators.

- Including in qMAESTRO a model for Digital In-Memory Computing (D-IMC). D-IMC has been proposed as a technology for implementing DNN accelerators to reduce energy and achieve optimal performance. Even with this new architecture, it is critical to optimize hardware configuration and mapping strategies. The inclusion of D-IMC in the cost model analysis framework further enables qGamma to optimize mapping strategies for this new architecture.
- Integrate qGamma into a DNN architecture search framework to explore the architecture of DNNs not only for accuracy optimization but also for optimizing hardware resource parameters and accelerator efficiency, such as latency and energy consumption.

Bibliography

- [1] M. Andersch, G. Palmer, R. Krashinsky, N. Stam, V. Mehta, G. Brito, and S. Ramaswamy. NVIDIA Hopper Architecture In-Depth. Technical report, NVIDIA, 2022. URL <https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/>.
- [2] C. Banbury, V. J. Reddi, P. Torelli, J. Holleman, N. Jeffries, C. Kiraly, P. Montino, D. Kanter, S. Ahmed, D. Pau, et al. Mlperf tiny benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.
- [3] S. Ben Ali, S.-I. Filip, and O. Sentieys. A Stochastic Rounding-Enabled Low-Precision Floating-Point MAC for DNN Training. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2024*, pages 1–6, Mar 2024.
- [4] G. Chen, M. A. Anders, H. Kaul, S. K. Satpathy, S. K. Mathew, S. K. Hsu, A. Agarwal, R. K. Krishnamurthy, V. De, and S. Borkar. A 340 mv-to-0.9 v 20.2 tb/s source-synchronous hybrid packet/circuit-switched 16×16 network-on-chip in 22 nm tri-gate cmos. *IEEE Journal of Solid-State Circuits*, 50(1):59–67, 2015. doi: 10.1109/JSSC.2014.2369508.
- [5] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016. doi: 10.1109/ISCA.2016.40.
- [6] R. Correa, A. Ferreira, and P. Rebreyend. Scheduling multiprocessor tasks with genetic algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 10(8): 825–837, 1999. doi: 10.1109/71.790600.
- [7] S. Galal and M. Horowitz. Energy-Efficient Floating-Point Unit Design. *IEEE Transactions on Computers*, 60(7):913–922, 2011. doi: 10.1109/TC.2010.121.
- [8] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. A survey of quantization methods for efficient neural network inference, 2021.

- [9] M. Harris. Mixed-Precision Programming with CUDA 8. Technical report, NVIDIA, 2016. URL <https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/>.
- [10] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.
- [11] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90.
- [12] E. Hou, N. Ansari, and H. Ren. A genetic algorithm for multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113–120, 1994. doi: 10.1109/71.265940.
- [13] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, 2017.
- [14] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson. Ten Lessons From Three Generations Shaped Google’s TPUv4i : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2021. doi: 10.1109/ISCA52012.2021.00010.
- [15] S.-C. Kao and T. Krishna. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.
- [16] S.-C. Kao, M. Pellauer, A. Parashar, and T. Krishna. Digamma: Domain-aware genetic algorithm for hw-mapping co-optimization for dnn accelerators. pages 232–237, 03 2022. doi: 10.23919/DATE54114.2022.9774568.
- [17] M. S. Kim, A. A. Del Barrio, H. Kim, and N. Bagherzadeh. The Effects of Approximate Multiplication on Convolutional Neural Networks. *IEEE Transactions on Emerging Topics in Computing*, 10(2):904–916, 2022. doi: 10.1109/TETC.2021.3050989.
- [18] A. Kuzmin, M. V. Baalen, Y. Ren, M. Nagel, J. Peters, and T. Blankevoort. FP8 Quantization: The Power of the Exponent, 2024.

- [19] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 754–768, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358252. URL <https://doi.org/10.1145/3352460.3358252>.
- [20] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE Micro*, 40(3):20–29, 2020. doi: 10.1109/MM.2020.2985963.
- [21] S. K. Lee, A. Agrawal, J. Silberman, M. Ziegler, M. Kang, S. Venkataramani, N. Cao, B. Fleischer, M. Guillorn, M. Cohen, S. M. Mueller, J. Oh, M. Lutz, J. Jung, S. Koswatta, C. Zhou, V. Zalani, M. Kar, J. Bonanno, R. Casatuta, C.-Y. Chen, J. Choi, H. Haynie, A. Herbert, R. Jain, K.-H. Kim, Y. Li, Z. Ren, S. Rider, M. Schaal, K. Schelm, M. R. Scheuermann, X. Sun, H. Tran, N. Wang, W. Wang, X. Zhang, V. Shah, B. Curran, V. Srinivasan, P.-F. Lu, S. Shukla, K. Gopalakrishnan, and L. Chang. A 7-nm Four-Core Mixed-Precision AI Chip With 26.2-TFLOPS Hybrid-FP8 Training, 104.9-TOPS INT4 Inference, and Workload-Aware Throttling. *IEEE Journal of Solid-State Circuits*, 57(1):182–197, 2022. doi: 10.1109/JSSC.2021.3120113.
- [22] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2018. URL <http://dx.doi.org/10.1109/IPDPSW.2018.00091>.
- [23] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* 5, page 115–133, 1943.
- [24] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.
- [25] NVIDIA. Automatic mixed precision for deep learning. URL <https://developer.nvidia.com/automatic-mixed-precision>.
- [26] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019. doi: 10.1109/ISPASS.2019.00042.
- [27] B. Parchamdar and M. Reshadi. Data flow mapping onto dnn accelerator considering

- hardware cost. In *2020 IEEE 14th Dallas Circuits and Systems Conference (DCAS)*, pages 1–5, 2020. doi: 10.1109/DCAS51144.2020.9330673.
- [28] S. Russell and P. Norving. *Artificial Intelligence A Modern Approach*. Pearson, 2021.
- [29] P. Shroff, D. W. Watson, N. S. Flann, and R. F. Freund. Genetic simulated annealing for scheduling data-dependent tasks in heterogeneous environments. In *5th Heterogeneous Computing Workshop (HCW'96)*, volume 970, pages 98–117, 1996.
- [30] H. Singh and A. Youssef. Mapping and scheduling heterogeneous task graphs using genetic algorithms. In *5th IEEE heterogeneous computing workshop (HCW'96)*, pages 86–97, 1996.
- [31] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan. Ultra-Low Precision 4-bit Training of Deep Neural Networks. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1796–1807. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/13b919438259814cd5be8cb45877d577-Paper.pdf.
- [32] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In *2008 International Symposium on Computer Architecture*, pages 51–62, 2008. doi: 10.1109/ISCA.2008.16.
- [33] J. J. Tithi, N. C. Crago, and J. S. Emer. Exploiting spatial architectures for edit distance algorithms. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23–34, 2014. doi: 10.1109/ISPASS.2014.6844458.
- [34] M. C. v1.1. MLPerf Tiny Benchmark. <https://github.com/mlcommons/tiny/tree/master>, 2021. [Online; accessed 08-May-2024].
- [35] S. Venkataramani, J. Choi, V. Srinivasan, W. Wang, J. Zhang, M. Schaal, M. J. Serrano, K. Ishizaki, H. Inoue, E. Ogawa, M. Ohara, L. Chang, and K. Gopalakrishnan. Deeptools: Compiler and execution runtime extensions for rapid ai accelerator. *IEEE Micro*, 39(5):102–111, 2019. doi: 10.1109/MM.2019.2931584.
- [36] L. Wang, H. J. Siegel, and V. P. Roychowdhury. A genetic-algorithm-based approach for task matching and scheduling in heterogeneous computing environments. In *Proc. Heterogeneous Computing Workshop*, pages 72–85, 1996.

- [37] R. Yamashita, M. Nishio, R. K. G. Do, and et al. Convolutional neural networks: an overview and application in radiology. *Insights Imaging 9*, page 611–629, 2018.
- [38] Z. Yao, Z. Dong, Z. Zheng, A. Gholami, J. Yu, E. Tan, L. Wang, Q. Huang, Y. Wang, M. W. Mahoney, and K. Keutzer. Hawqv3: Dyadic neural network quantization, 2021.
- [39] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. 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*, page 161–170, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333153. doi: 10.1145/2684746.2689060. URL <https://doi.org/10.1145/2684746.2689060>.

List of Figures

2.1	Traditional Programming vs. Machine Learning.	5
2.2	Schema of an Artificial Neuron.	8
2.3	Activation functions usually applied to neural networks: (a) ReLu, and (b) sigmoid, (c) tanh [37].	9
2.4	An example of Deep Neural Network.	10
2.5	An overview of a CNN [37].	11
2.6	An example of convolution operation with a kernel size of 3x3, no padding and stride 1 [5].	12
2.7	An example of max pooling operation with a filter of size 2x2 no padding, and stride of 2 [37].	14
2.8	Abstract DNN accelerator architecture model [19].	15
2.9	High level block diagram spatial architecture for CNN with an off-chip DRAM [5].	16
2.10	Four-core mixed-precision AI chip architecture [21]	17
2.11	Mixed precision example in deep learning [25].	18
2.12	New DP4A and DP2A instructions in Tesla P4 and P40 GPUs provide fast 2- and 4-way 8-bit/16-bit integer vector dot products with 32-bit integer accumulation [9].	19
2.13	The taxonomy of data reuse types in DNN accelerators and the corresponding implementation options for each type [20].	20
2.14	Example of dataflow in MAESTRO notation.	22
2.15	Weight Stationary dataflow steps [27].	23
2.16	Weight Stationary computation steps [27].	23
2.17	Different OS datflow variant: (a) SOC-MOP, (b) MOC-MOP, and (c) MOC-SOP [5].	25
2.18	Processing of an 1D convolution primitive in the PE. In this example, $R = 3$ and $S = 5$ [5].	26
3.1	Decoded 2-level genome in MAESTRO's description [15].	29
3.2	Overview of GAMMA [15].	30

3.3	High-level overview of mapping a high-dimensional DNN layer (CONV2D in this figure) to an accelerator with 2-D PE array. (a) An Overview of Mapping CONV2D to an Accelerator. (b) High-level Tool flow of MAESTRO [20].	31
3.4	An overview of MAESTRO's analysis framework [19].	32
3.5	HW/Mapping Co-optimization Framework with the three technical contributions highlighted [16].	34
3.6	(a) Co-opt Framework, (b-c) the HW-Mapping encoding representation, and (d-e) the corresponding decoded accelerator configuration. (f) The formula for calculating minimum on-chip buffer requirement. (g) The definition of notations. [16].	34
4.1	Target architecture of our research.	37
4.2	Simplified block diagram of the configurable PE.	38
5.1	qGamma and qMaestro overview.	42
5.2	Example of 3 level-mapper genome.	43
5.3	Example of genomes with and without explorable clusterization.	45
5.4	Example of the cross tile algorithm with a cross over on two dimensions in every mapper.	46
5.5	Example of dataflow file supported by qMaestro.	47
6.1	EDP results for ResNetV1. Number of MACs per-layer also reported. . . .	53
6.2	Latency results for ResNetV1.	54
6.3	Activity-based energy results for ResNetV1.	54
6.4	Activity-based energy results for MobilenetV1. Number of MACs per-layer also reported.	56
6.5	Latency results for MobilenetV1.	56
6.6	Quantization per layer in mixed-precision configuration.	57
6.7	Latency results for ResNet18. Number of MACs per-layer also reported. . .	57
6.8	Activity-based energy results for ResNet18.	58

List of Tables

2.1	Shape parameters of a convolutional layer [5].	12
2.2	A list of parameters and hyperparameters in a CNN [37].	13
3.1	Terminology in Genetic Algorithm [15].	28
4.1	Values of MAC energy based on different quantization in 22nm architecture.	39
6.1	Layers' characteristics for ResNetV1	50
6.2	Layers' characteristics for MobileNetV1	51
6.3	Layers' characteristics for ResNet18	51

