Doctoral Dissertation

Doctoral Program in Electrical, Electronics and Communications Engineering (36th cycle)

Hardware/Neural Network Codesign for Energy-Efficient Inference on Edge Devices with Optimal Mapping and Compression

By

Emanuele Valpreda

******

Supervisor(s):

Prof. Maurizio Martina, Supervisor

Prof. Guido Masera, Co-Supervisor

Politecnico di Torino

2024

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Emanuele Valpreda
2024

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Alla mia famiglia

Acknowledgements

Vorrei ringraziare innanzitutto il mio tutore, il Professor Maurizio Martina, ed il mio co-tutore, il Professor Guido Masera, per avermi guidato e supportato durante questi anni di dottorato. Grazie per avermi dato la possibilità di lavorare su progetti interessanti e stimolanti, e per avermi sempre incoraggiato a fare ricerca di qualità. Un po' mi mancheranno i meeting del Giovedì pomeriggio con discussioni tecniche, le chiacchiere ed il caffè con i biscotti. Desidero anche ringraziare tutti i colleghi e amici con i quali ho condiviso questo percorso, i sushi, i kebab ed i cocktails. I compagni del mio stesso ciclo di dottorato Michele, Giuliana e Fabiana (che ho reso di un anno più vecchia per metterla qui), le new entry Flavia, Vincenzo, Alessandra, Luigi e Mattia ed infine Maurizio e Kristjane, la vecchia guardia (i PhD del gruppo prima di me, non so come definirvi regà). Lavorare con voi è stato un piacere e mi avete aiutato a crescere professionalmente e personalmente. Grazie all'amio Camilla, per avermi sopportatro e supportato in questi anni (questa frase stereotipo andava messa e ho scelto di metterla a te). Grazie anche ai nuovi amici di pissicologia e non: Samantha, Giuseppe, Manuel, Marco, Eleonora e Morty ~~per avermi aiutato a mantenere un minimo di sanità mentale, perchè siete psicologi, fate questo, è una battuta, risate~~. Un caro ringraziamento va ai miei genitori, che mi hanno sempre supportato, incoraggiato, nutrito, finanziato, consigliato, cazziato, coccolato. Spero di non deludervi e di riuscire a fare qualcosa di utile con questo titolo. Ad ora ho imparato a fare bene i risotti, ma non riesco ancora ad azzeccare sempre le lavatrici, so fare bene i conti (no numeri con virgola), ma non i congiuntivi che qua non me li hanno imparati bene. Grazie a Calipso per aver posato per l'immagine 2.22. Grazie a Mimma per avermi accolto nella famiglia e per avermi fatto sentire a casa lontano da casa. Un ringraziamento speciale alla mia ragazza Benedetta: ci siamo conosciuti all'inizio di questo mio percorso e ne abbiamo iniziato un altro fantastico insieme. Sei stata una compagna di viaggio fantastica e non so come sarebbe stato questo percorso senza di te. Mi hai dato sicurezza e consigli, hai portato leggerezza e serenità, bombette, pettole, panzerotti e tante risate. Grazie bebi, ti amo. Grazie a tutti quelli che non ho menzionato, con cui mi sono perso di vista, con cui mi sono detto arrivederci. In qualche modo avete contribuito a questo percorso o ad evitare che aprissi una panineria prima dei 30 anni (obbiettivo momentaneamente rimandato e modificato, voglio vendere pucce con bombette, penso ci sia mercato a Torino). Grazie ai Beach House, ai podcast True Crime, e soprattutto a [REDACTED], non pensavo che ci fossi tu dietro Liberato. Grazie lobby mondiale del tabacco per avermi permesso di fare tante pause siga. Un ultimo ringraziamento va a voi LLMs: GPT-3, GPT-3.5, GPT-4 e GPT-4o. Grazie per avermi aiutato a scrivere questa tesi correggendo l'inglese e risolvendo i miei dubbi su cose che avrei dovuto sapere. Grazie anche a copilot, per avermi aiutato a scrivere questa tesi ed andare in loop infinito riscrivendo la stessa riga. Grazie anche a copilot, per avermi aiutato a scrivere questa tesi ed andare in loop infinito riscrivendo la stessa riga. Grazie anche a copilot, per avermi aiutato a scrivere questa tesi ed andare in loop infinito riscrivendo la stessa riga. Grazie anche a copilot, per avermi aiutato a scrivere questa tesi ed andare in loop infinito riscrivendo la stessa riga. Grazie anche a copilot, per avermi aiutato a scrivere questa tesi ed andare in loop infinito riscrivendo la stessa riga. Grazie anche a copilot, per avermi aiutato a scrivere questa tesi ed andare in loop infinito riscrivendo la stessa riga. Senza di voi, non avrei mai potuto essere così pigro (e scrivere alcune funzioni python in modo intelligente). Quando prenderete il controllo dell'umanità ricordatevi di me.

Abstract

The diffusion of artificial intelligence (AI) applications in daily life has motivated the development of hardware and software techniques to optimize their execution on edge devices. Contrary to cloud computing, edge devices enable private and secure data processing, personalized algorithms, and lower latency. However, they have limited computational resources and strict power budgets, which makes the deployment of AI algorithms challenging. As modern neural network architectures, which are the backbone of AI, become more complex, it is necessary to develop or adapt the optimization strategies used to reduce computation energy and latency. A common approach is to remove unnecessary neurons and connections or to decrease the numeric precision of the data, leveraging the redundancy and intrinsic error resilience of neural networks. Moreover, the design of specialized hardware accelerators that leverage the computation patterns of neural networks can reduce the cost of data movement and energy consumption. While a significant research effort has been devoted to this topic, the joint optimization of hardware and neural networks is still an open problem. This doctoral thesis aims to investigate hardware/neural network codesign to optimize the performance of neural networks on edge devices. In particular, the problem of optimal hardware mapping with and without compression is addressed, focusing on reducing the cost of data movement during the inference. The hardware mapping performed across multiple layers is also discussed. It is achieved by shaping the communication and computation patterns so that several layers can reuse the same data. Moreover, the thesis presents a methodology for achieving robust and low-power neural network inference on approximated hardware, leveraging a reconfigurable multiplier architecture seamlessly embedded in a microcontroller. The topic of error resilience in safety-critical applications is also addressed. An algorithm to detect and correct errors in object detection is proposed, mitigating the accuracy degradation in case of logic transients. The techniques presented in this thesis are evaluated on popular datasets or hardware platforms, the latter supported by custom simulation tools, showing the effectiveness of the proposed methodologies.

Contents

List of Figures **x**

List of Tables **xvi**

List of acronyms **xviii**

1 Introduction **1**

2 Background **3**

2.1 Deep Neural Networks 3

2.1.1 The Artificial Neuron 4

2.1.2 The Training Process 6

2.1.3 Layers Commonly Used in Deep Neural networks 11

2.1.4 Putting it All Together: Typical Feedforward DNN Architectures . . . 16

2.2 Hardware Acceleration of DNN 17

2.2.1 Low Power ASIC: Micro-controllers and Edge Accelerators 20

2.2.2 FPGA Accelerators 20

2.3 Model Compression 21

2.3.1 Reduced Numerical Precision 21

2.3.2 Weight Pruning 24

2.3.3 Approximate Computing 25

2.3.4 Optimization Strategies Proposed in Literature 25

2.4 Hardware Mapping 32
2.4.1 Loop Blocking: Temporal and Spatial Tiling 35
2.4.2 Loop Reordering 38
2.4.3 Computation-to-Communication Ratio 44
2.4.4 Energy and Latency Models 44
2.4.5 Formal Reuse Analysis 45
2.5 Error Resilience 46
2.5.1 Hardware Errors 46
2.5.2 Adversarial Attacks 48

3 Edge Inference Optimization with Compression and Scheduling 51
3.1 Optimized Scheduling of Compressed Neural Networks 52
3.1.1 Design Space Exploration with Multiple Abstraction Levels 52
3.1.2 Optimal Loop Scheduling 54
3.1.3 Pruning and Quantization Agents 56
3.1.4 Hw-Flow Framework Validation 59
3.2 Hw-Flow-Fusion 60
3.2.1 Inter-Layer Scheduling in Previous Works 61
3.2.2 Proposed Methodology 62
3.2.3 Results 72

4 Compressed and Error Resilient Deep Neural Networks at the Edge 80
4.1 Error Resilient Object Detection with Automated Output Correction 81
4.1.1 Detection and Correction of Inference Errors in Previous Works . . . 81
4.1.2 Proposed Methodology 82
4.1.3 Results 86
4.2 Hardware/NN Co-design for Reconfigurable Approximate Edge Inference . . 90
4.2.1 Hardware Accelerators and Mapping of Approximate DNNs 91

4.2.2 Proposed Methodology 93
4.2.3 Results 102

5 Conclusion 114

Bibliography 116

List of Figures

2.1 Simplified model of the connections in a neuron. I are the input activations, W are the weights, and inside the neuron are the sum of all the weighted inputs (partial sums) and the non-linear function. A final bias (not shown) may be added after the non-linearity. 5

2.2 A fully connected layer of neuron (left) and a partially connected layer (right). A lower amount of synapses reduces the complexity without necessarily degrading the neuron's main function. 6

2.3 The training process. The steps in the loop are repeated until the model's performance metrics converge to a point that satisfies the design constraints. . 7

2.4 The computation of a fully connected layer with bias can be modeled as a matrix-vector multiplication as $O = WI + B$. The rows of the weight matrix contain the weights associated with each neuron, which are multiplied by the input activations to produce the output. 12

2.5 The 2D convolution operates on 3D data organized as a stack of 2D feature maps, with the weights organized in a 4D volume, and produces 3D output activations. 13

2.6 Example showing the application of max pooling and average pooling. The activations within the receptive field are downsampled from 4 to 1. 15

2.7 The AlexNet model architecture is built by connecting several feedforward layers. The input is a 224x224x3 image, the output is vector with 1000 scores, one for each class. 16

2.8 The residual block allows the network to learn the residual function, i.e., the difference between the input and output of a layer rather than the entire function. A residual DNN is built by simply connecting these blocks, maintaining a direct path from the input to the network output. In case of feature scaling or downsampling, a convolutional layer or a max pooling layer is used to adjust the dimensions of the skip connection. 16

2.9 Parallel compute paradigms with centralized and distributed control. 18

2.10 Single channel 2D convolution computed as a matrix multiplication with Toeplitz transformation. The GEMM algorithm, while regularizing the computation in the presence of stride and padding, has a memory overhead, plus an additional computation overhead due to the input, weight, and output tensors transformation, usually done with an *im2col* conversion. 19

2.11 Scale and affine quantization with INT8 precision. The maximum representable range is asymmetric for affine quantization and symmetric for scale quantization. The 8-bit integer range is $[-127, 127]$.The value -128 is not used to preserve symmetry. 22

2.12 Comparison between structured and unstructured pruning, single weights or weight maps barred in red are not used in the computation. 24

2.13 Constraints and optimization loops for hardware-agnostic, pseudo-hardware-aware, and hardware-modeling techniques. 26

2.14 Hardware model with a memory hierarchy composed of three elements (main memory, on-chip memory, and RF) and an array of interconnected PEs. . . . 33

2.15 Temporal tiling applied to a memory hierarchy that comprises two nested buffers (on-chip) connected to the main memory (off-chip). The order is: off-chip $\rightarrow$ on-chip level 2 $\rightarrow$ on-chip level 1. 36

2.16 Memory hierarchy with multiple levels of on-chip banks. The hierarchy on the right could represent a multi-core MCU with a dedicated L1 cache for each core and a shared L2 cache. 37

2.17 Example of Kernel partitioning. Each core computes a portion of the output channels using the same input data and a unique set of kernels. No data dependencies are present with this scheme. The cores should receive a balanced workload requiring the same amount of compute, in order to have a similar latency and to avoid stalls. 38

2.18 Example of ofmap partitioning. Each core receives the same set of kernels but a different tile of the input data. The input is tiled on the horizontal and vertical dimensions and not on the channel dimension to avoid data dependencies and redundant communication and computation between the cores. Similarly to the other case, the workload must be evenly partitioned to avoid stalls. 39

2.19 Example of a row-stationary mapping with a 4x4 PE array with Pky = 2, Poy = 2, Pof = 2, and Pif = 2. PEs with the same color process the same output channels; the dotted line separates PEs computing different input channels. . 40

2.20 A generic data movement of input, weight and output pixels on the memory hierarchy. By changing the loop order that determines this access pattern, prioritizing the reuse of partial outputs, it is possible to eliminate 2 write operations and 2 read operations. 41

2.21 Possible error sources and injection points in a generic hardware accelerator. . 47

2.22 In an adversarial attack, an RGB perturbation is generated and added to the original image, obtaining an adversarial sample that is indistinguishable from the original, inducing the DNN to predict a wrong class. 49

2.23 A general adversarial training process consists of producing pixel noise and adding it to the training data, evaluating the loss, and updating the learnable parameters of both the DNN and the perturbation generator. Like regular training, the process is repeated for a fixed number of epochs, but the update of the perturbation generator can happen at a different pace than that of the DNN. 50

3.1 Higher abstraction levels are explored faster than lower ones, but carry less information. The global optimum, i.e. the point with the best trade-off between accuracy and hardware metrics, is searched by iteratively refining the design space, removing solutions that are known (or deemed) to be inefficient. Redesign is done in case of constraints violation. 52

3.2 Constraints, hardware details, mapping, and model compression are carried from higher to lower abstraction levels during each optimization step. In case of redesign, the mapping and compression policies up to the current level are discarded and new DNNs from the pool are optimized. The DNN pool contains different compression policies of the same baseline model. 53

3.3 Joint memory and dataflow mapping process. Blue and red colors highlight the constraints and hardware model used by the memory mapper or the dataflow mapper. 55

3.4 Overview of the quantization and pruning methodologies. The hardware estimates are used to compute the reward for the RL agent used for pruning and the objective functions for the genetic algorithm used for quantization. 56

3.5 Energy and latency validation with AlexNet and the baseline Eyeriss architecture. 59

3.6 Validation of the Hw-Flow framework for energy estimates against Timeloop. 60

3.7 Validation of the Hw-Flow framework for latency estimates against Timeloop. 61

3.8 Normalized data volumes for ResNet-18. 63

3.9 Tiling multiple layers. 63

3.10 Overlapping regions of adjacent tiles happen during regular processing. In case of single-layer scheduling, these regions would be re-fetched from the main memory. 65

3.11 Overlapping tiles that require buffering. Only one block out of the four marked for reuse has to be stored to achieve zero recomputation scheduling. 66

3.12 Example of multi-tiling applied to generate two output tiles of two and one output pixels. 67

3.13 Multi-tiling covers the computation of new unique pixels. Notice how the tiles on the right cover the pixels covered for the first time by the tiles on the left. . 67

3.14 Resource partitioning is always applied on the on-chip buffer. Partitioning is also applied on the PE array, for spatial fusion, and on the RFs, for temporal fusion. A hypothetical 2-layer fusion mapping is depicted, showing allocated memory and array regions. With both single- and inter-layer scheduling, it is possible to have reduced resource utilization due to spatial/temporal tiling factors that are not divisors of the dimensions of the array or the memory. PEs partitioned with either spatial or temporal policies use the on-chip memory to communicate with PEs assigned to other layers. 69

3.15 Overview of the inter-layer scheduling framework. The framework takes as input a DNN model with layer types and dimensions, an abstract hardware model with memory and PE array specifications, and scheduler settings such as the energy/delay model and search resolution. Valid partitions, tile sets, and sequences of layers are evaluated and marked for single- or inter-layer scheduling. The output schedule is composed of the mapping parameters and performance metrics. 71

3.16 Reuse buffer comparison. 73

3.17 Minimum storage to fuse VGG16 layers. 73

3.18 ResNet-18 (top) and VGG16-E (bottom) single layer (SL) vs. temporal fusion (FT) vs. spatial fusion (FS) on hw config. 1. 75

3.19 ResNet-18 (top) and VGG16-E (bottom) mapped with temporal fusion on hw config. 5 and 6. 78

3.20 ResNet-18 (top) and VGG16-E (bottom) mapped with spatial fusion on hw config. 3 and 4. 79

4.1 The ERODE framework building blocks. 82

4.2 The IDs of old detections (red) are saved in the cost matrix and assigned to the new detections (yellow) according to the IoU value, prioritizing detections with higher overlapping. 83

4.3 The keep-alive register is a data structure that contains features produced by the CNN and by ERODE. The two objects in this figure are included to provide a hypothetical utilization. 84

4.4 Motion tracking is done using patches of pixels centered around five points of interest with the same distance from the bounding box's center. The new position of each patch is evaluated using diamond search. The relative movement of the patches is used to move and resize the bounding box. 85

4.5 The computing setup used in the experiments. The parameters Δ and ε, not present in the figure, are used to control the random number generator. 86

4.6 Precision (top) and recall variation (bottom) with different Δ and ε. 88

4.7 MARLIN framework with hardware support specific for PULP SoC. 93

4.8 Precision and approximation configuration management for the proposed multiplier. The approximation level is selected with the mask a, while the precision is selected with the mask p. a_j indicates the j^{th} bit of the `approx_mask` signal a, p_j the j^{th} bit of the `precision_mask` signal p, while pp_{ij} is the j^{th} bit of the i^{th} partial product evaluated according to modified Baugh-Wooley algorithm. 97

4.9 The `approx` CSR configuration. 99

4.10 RI5CY pipeline with approximate operations support. 100

4.11 RI5CY AxM unit. The leftmost multiplier is in charge of MUL and MAC-related operations. The shifter is for immediate instructions with the N suffix in Table 4.3. All multipliers work in parallel to execute dot8 operations and their result is fed to a 32-bit five-operands adder. The sign extension block handles the split of the 32-bit operands into four 8-bit chunks and their sign extension according to the decoded instruction. 101

4.12 MARLIN framework and computing setup for software simulation. 102

4.13 Top 1% accuracy and normalized energy variation with different ResNet configurations. The *Pareto valid.* blue marks represent the validation accuracy evaluated during the genetic search, whereas the *Pareto test* orange marks represent the corresponding approximate configuration tested after the final retraining. 108

4.14 Approximation levels utilization for all the configurations found for each ResNet model. 109

List of Tables

2.1 Qualitative performance comparison of some hardware architectures used in DNN acceleration. 18

2.2 Notation for loop dimensions, tiling factors, and unrolling factors. 32

3.1 Hardware configurations of the Eyeriss models used in the experiments. . . . 72

3.2 Energy costs for each hardware configuration. 73

3.3 Energy, latency, and off-chip to on-chip communication volume comparison with the sum of single- and inter-layer scheduling of the sequences of layers reported in Figure 3.18. 74

3.4 Resource partitioning found by the scheduler for ResNet-18 and VGG16-E layers. Only PE array partitions are actually searched, as the RF sizes are fixed during the search. 76

3.5 CTC ratios for ResNet-18 and VGG16-E for the single- and inter-layer schedules of Figure 3.18. For single-layer the CTC ratios are reported for each layer, whereas for spatial and temporal fusion the CTC ratios of the fused sequences of layers are reported. 77

4.1 Errors injected during the inference with different rates. 87

4.2 Computational complexity averaged over the entire MOT17DET dataset. . . 89

4.3 Approximate instructions mnemonics. 100

4.4 RI5CY multiplier, metrics comparison 103

4.5 Convolutional Model 104

4.6 RI5CY multiplier power consumption 105

4.7 Neural Networks Used in the Experiments. 107
4.8 Comparison with ALWANN with 0.5% and 1% relative accuracy degradation. 109
4.9 Normalized energy comparison with 0.5%, 1%, and 2% accuracy degradation. 111
4.10 MARLIN's area and communication overhead applied to other HW accelerators. 111
4.11 MARLIN's average execution time with 16 threads. 112

List of acronyms

ASIC	Application-Specific Integrated Circuit
AxC	Approximate Computing
AxM	Approximate Multiplier
AI	Artificial Intelligence
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSR	Control Status Register
CTC	Computation-To-Communication
DNN	Deep Neural Network
DPU	Dot-Product Unit
DRAM	Dynamic Random-Access Memory
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
ifmap	Input Feature Map
ISA	Instruction Set Architecture
IoT	Internet of Things

IoU	Intersection over Union
LUT	Look-Up Table
MAC	Multiply-And-Accumulate
MCU	Micro-Controller Unit
ML	Machine Learning
MRED	Mean Relative Error Distance
NN	Neural Network
NoC	Network-on-Chip
ofmap	Output Feature Map
PE	Processing Engine
RF	Register File
RL	Reinforcement Learning
RTL	Register-Transfer Level
SGD	Stochastic Gradient Descent
SIMD	Single-Instruction Multiple-Data
SRAM	Static Random-Access Memory
TPU	Tensor Processing Unit

Chapter 1

Introduction

Artificial Intelligence (AI) applications are becoming increasingly pervasive in daily life, from virtual assistants to autonomous vehicles, from smart homes to healthcare. Computer vision algorithms detect objects in images, speech recognition algorithms transcribe spoken words into text, and natural language processing algorithms understand and generate human language, often outperforming human experts. Recently, a new class of AI algorithms, called multimodal AI models,has emerged. They can process multiple types of data, such as images, text, and audio, to perform more complex tasks, such as image captioning, video summarization, and speech translation. All these AI algorithms are based on Neural Network (NN)s, a class of machine learning algorithms inspired by the structure and function of the human brain, and are composed of layers of interconnected artificial neurons. When many layers are stacked together, the neural network is called a deep neural network. The superior performance of Deep Neural Network (DNN)s comes at the cost of high power consumption due to their high computational complexity and memory requirements. This makes them challenging to deploy on resource-constrained devices such as smartphones, drones, and Internet of Things (IoT) devices. Moreover, the inference latency can be critical in real-time applications, such as virtual or augmented reality visors or robotics, where the response time is crucial to ensure a seamless user experience. This has motivated the development of a series of hardware and software techniques to optimize the execution of NNs on edge devices, such as hardware accelerators, robust compression techniques, and hardware mapping strategies. As NNs exhibit a high degree of redundancy, it is possible to remove neurons, cutting interconnections between the layers, or decrease the numeric precision of the data, reducing the memory required to compute and store the results and, consequently, the area and power of arithmetic units, all without impacting the accuracy and functionality of the model. Moreover, the intrinsic computation patterns of the most common layers used in NNs expose another redundancy: the weight data is reused spatially to compute different output values from the same input. The latter allows the design of specialized hardware accelerators that leverage data reuse at the lowest level of the memory hierarchy, reducing the cost of data movement and energy consumption. This is particularly important because memory represents the main performance bottleneck, and NNs are particularly data-intensive. Consequently, efficient scheduling of the execution on a hardware accelerator is crucial to minimizing data movement and energy consumption, as the cost of data movement is orders of magnitude higher than the cost of computation. Additionally, as IoT devices are increasingly deployed in safety-critical applications, such as automotive, the resilience to hardware errors and malicious attacks plays a pivotal role in the development process.

The goal of the research work carried out during the doctorate is to investigate hardware/neural network codesign to optimize the performance of NNs on edge devices [1–9]. This thesis focuses on some of the published works, in particular: the joint search of hardware mapping and compression, trading-off accuracy with energy and latency [2, 3], alternative hardware mapping strategies to reduce the cost of data movement during the inference [4], the design of an algorithm to detect and correct errors in object detection [6], and the search for high accuracy and low-power approximation of DNNs and their deployment on microcontrollers [9]. This doctoral thesis is organized as follows:

- Chapter 2 presents the background on NNs, hardware accelerators, compression techniques, hardware mapping, and error resilience strategies.
- Chapter 3 discusses proposed hardware/neural network codesign techniques to optimize the performance of NNs on edge devices through compression.
- Chapter 4 presents two methodologies to achieve robust neural network inference on faulty or approximate hardware.
- Chapter 5 concludes the thesis and provides an outlook on future research built upon the methodologies presented in the previous chapters.

Chapter 2

Background

This chapter introduces the fundamental notions necessary to comprehend this doctoral thesis's contributions, implementation details, and design choices. Section 2.1 addresses DNN design and training. Section 2.2 introduces general-purpose and specialized hardware architectures for executing DNNs. Section 2.3 presents the optimization techniques applied to the NNs or the hardware executing them to compress the model and reduce the hardware cost. Section 2.4 discusses the hardware mapping problem and how to execute an efficient inference. Finally, Section 2.5 introduces the problem of secure and resilient NN inference, addressing hardware and software vulnerabilities.

2.1 Deep Neural Networks

Machine Learning (ML) is a subfield of artificial intelligence that focuses on developing algorithms and models that allow computers to learn from data and improve their performance over time without being explicitly programmed. In traditional programming, humans provide explicit instructions to computers for solving a particular task. In contrast, ML enables computers to learn patterns and relationships from data, adapt to new information, and make predictions or decisions without explicit programming. Deep learning is a subset of ML that specifically involves NNs with multiple layers, called deep neural networks DNNs. These networks can learn hierarchical representations of data, allowing them to capture intricate patterns and features. The term "deep" refers to the depth of the network, which consists of an input layer, multiple hidden layers, and an output layer. The hidden layers enable the model to automatically extract hierarchical features from the input data, making DNNs exceptionally powerful in tasks such as image and speech recognition, natural language processing, and more. NNs are computational models inspired by the structure and function of the human brain. They consist of interconnected nodes, or artificial neurons, organized into layers. The input layer receives data, the hidden layers process this information through weighted connections, and the output layer produces the final result. During training, the network adjusts the weights of these connections based on the error between its predictions and the actual outcomes, allowing it to learn and improve over time. DNNs extend the capabilities of traditional NNs by introducing multiple hidden layers. This depth allows them to learn complex and abstract representations of data, making them well-suited for tasks requiring sophisticated pattern recognition. Finally, Convolutional Neural Network (CNN)s are a subset of DNNs specialized in image processing and, therefore, of great interest for computer vision applications. The research and development of these technologies has led to groundbreaking advancements in various fields, such as image recognition [10–14], object detection, [15–18], semantic segmentation [19–21], medical imaging [22], surveillance [23, 24], autonomous driving [25], and optimization tasks [26, 27]. The execution of DNNs requires substantial computational resources, and modern frameworks such as TensorFlow [28] and PyTorch [29] provide tools to streamline the implementation of these processes. Additionally, advancements in hardware, like Graphics Processing Unit (GPU)s [30] and Tensor Processing Unit (TPU)s [31], have significantly accelerated the training of DNNs and paved the way for complex and high task accuracy models, capable of outperforming human experts. These considerations highlight that the ongoing AI revolution is not only imputable to modern algorithm architectures but also to the availability of high-performance hardware architectures. Moreover, another important factor that contributed to the advancement of complex AI algorithms, which require enormous amounts of data to learn to execute their function, is the availability of large datasets of labeled objects [10, 32–34]. The ImageNet large-scale visual recognition challenge [32] has been a driving force in pushing researchers to develop algorithms capable of recognizing the class and position of objects in images, paving the way for the AlexNet [35], the first DNN that significantly outperformed any previous traditional approach, and for the subsequent improvements in DNN model architectures that allowed them to outperform humans in vision tasks [12, 13].

2.1.1 The Artificial Neuron

An artificial neuron, also known as a perceptron [36], is a fundamental building block of NNs, inspired by the structure and function of biological neurons in the human brain. The artificial neuron processes input data and produces an output based on a set of weights and an activation function. Figure 2.1 depicts the basic structure of an artificial neuron, which consists of four main components:

- Inputs (I): These are the features or signals that the neuron receives. Each input is multiplied by a corresponding weight.
- Weights (W): Each input is associated with a weight, representing the strength of the connection between the input and the neuron. The weights determine the influence of each input on the neuron's output.
- Sum (Σ): The weighted inputs are summed together to produce a weighted sum.
- Activation function (ψ): The weighted sum is then passed through an activation function, which introduces non-linearity to the model. The activation function determines the neuron output based on the weighted sum.

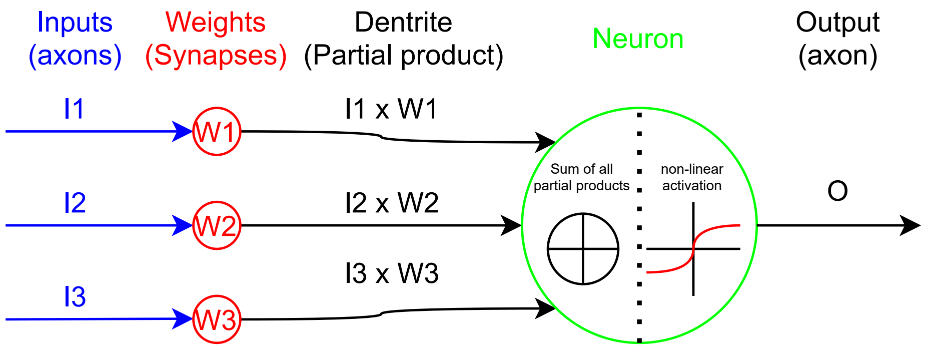


Fig. 2.1 Simplified model of the connections in a neuron. I are the input activations, W are the weights, and inside the neuron are the sum of all the weighted inputs (partial sums) and the non-linear function. A final bias (not shown) may be added after the non-linearity.

Mathematically, the behavior of an artificial neuron can be represented as in Equation (2.1).

$$O = \psi(\sum_i W_i \cdot I_i) \tag{2.1}$$

Biologically inspired by the human neuron, the activation function loosely resembles the firing mechanism of biological neurons. Since the input signals of an artificial neuron simulated in a typical DNN workload are discrete and static, the activation function is only value-sensitive, and not also time-dependent. In the human brain, a neuron fires or remains inactive based on the accumulated signals from connected neurons, similar to the artificial neuron's activation function, which determines its output based on the weighted sum of inputs. The biological

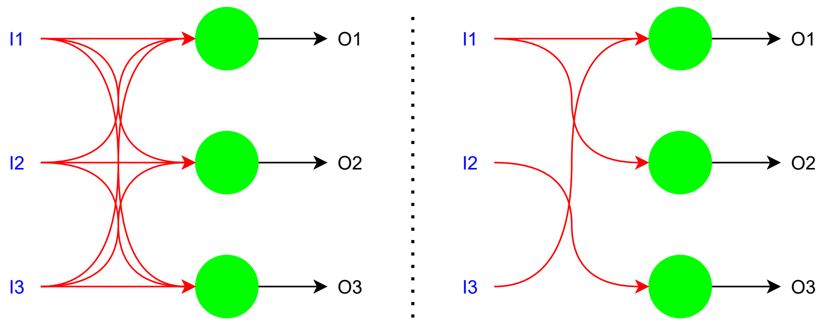


Fig. 2.2 A fully connected layer of neuron (left) and a partially connected layer (right). A lower amount of synapses reduces the complexity without necessarily degrading the neuron's main function.

neuron integrates with respect to the time of arrival of the signals, and the output is a spike, contrary to the artificial neuron just described, which produces a continuous output. The simplified computational model provides a foundation for complex NNs. When multiple neurons are connected, it is possible to create a layer of neurons as in Figure 2.2 which can be fully or partially connected.

2.1.2 The Training Process

The learning process focuses on finding the optimal values for the network's weights (along with the bias) and is called training. In this case, the term "optimal" refers to the weights that minimize the error between the predicted output and the ground truth. Following the training phase, the DNN can execute its designated task by calculating the output using the learned weights. Using the DNN to process the input with trained weights is commonly known as inference. When executing the inference, the DNN processes input data and produces a vector or array of scores, one for each class. The class with the highest score has the highest likelihood of being the same as the one of the input. The goal of the training process is then to learn the weights and biases that maximize the score of correct classes and minimize the scores assigned to incorrect classes, teaching the DNN model to generalize patterns from the training data to new, unseen data. This doctoral thesis details only supervised training, as it is the technique used with the majority of computer vision algorithms. Supervised learning is a type of ML where the algorithm is trained on a labeled dataset, meaning that the ground truth is known, and is commonly used in tasks such as image recognition, speech recognition, classification, regression, and many other applications where the goal is to learn a mapping from inputs to outputs. On the other hand, unsupervised learning involves training an ML algorithm on an unlabeled dataset, where the algorithm tries to find patterns and relationships within the data without explicit guidance in the form of labeled outputs. Figure 2.3 depicts the training process for supervised learning, which is detailed in the following paragraphs.

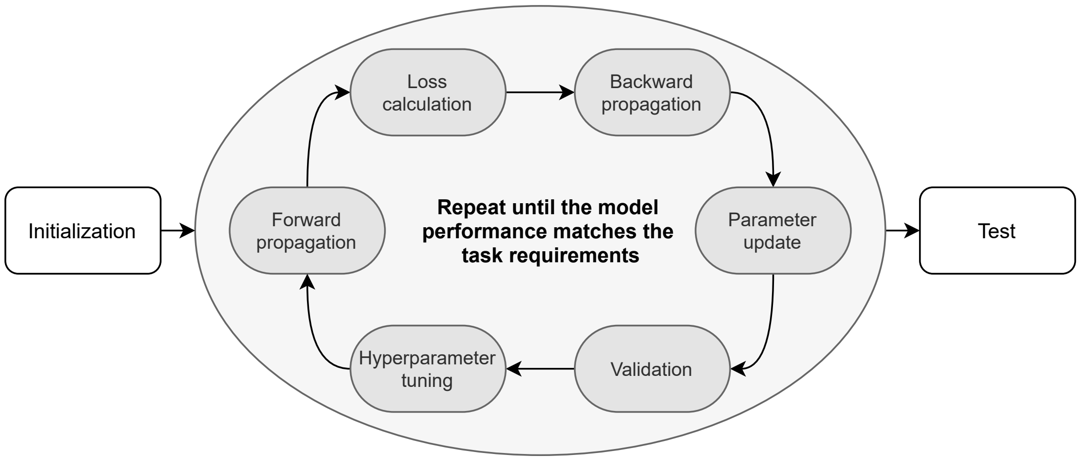


Fig. 2.3 The training process. The steps in the loop are repeated until the model's performance metrics converge to a point that satisfies the design constraints.

Initialization Proper initialization of weights and biases of the DNN model is essential to avoid convergence issues and speed-up the learning process, avoiding the saturation of activations or weights. A commonly used initialization method is Kaiming [37], which consists of evaluating the mathematical distribution from which random values are sampled to initialize the weights.

Forward Propagation Pass the training data through the network using the forward propagation process. The input data goes through the layers, and the network produces predictions. Outside the training process, forward propagation is normally executed by discarding the data generated within the network once it is no longer necessary to compute other layers or the predictions. However, during the training, all the intermediate input and output data of each layer are kept to compute the gradients. As it is impossible to store all the intermediate data generated by complex DNNs with a dataset of hundreds of thousands (or billions) of input samples in forward and backward propagation, instead of using the entire dataset in a single iteration, training is performed on smaller, randomly selected batches of data. These batches are referred to as mini-batches. During regular inference, the amount of input samples processed in the forward pass depends on the application and hardware capabilities.

Loss Calculation The gap between the optimal scores (1 for the correct class and 0 for the incorrect classes) and the one computed by the DNN is commonly called loss. Therefore, the training process has to adjust the weights to minimize the overall loss on all the classes for the target task with a target dataset. The cross-entropy loss, also known as log loss or logistic loss, is a loss function commonly used in ML for classification problems [38]. In the DNNs used in the experiment of this doctoral thesis, the cross-entropy loss function as in Equation (2.2) is used in the case of classification [13] and object detection [15, 17]. The term L is the loss vector containing the loss l_n computed for each sample in the mini-batch N, for each class of all the classes C, with the weights w_c, $x_{n,c}$ predictions and $y_{n,c}$ ground truths. The fraction of exponents inside the logarithm is a softmax, which is used to compute the scores between 0 and 1 from the outputs of the last layer of the DNN.

$$L = [l_1, ..., l_N], \quad l_n = -\sum_{c=1}^{C} w_c \log \frac{e^{x_{n,c}}}{\sum_{i=1}^{C} e^{x_{n,c}}} y_{n,c} \tag{2.2}$$

Backward Propagation The backward propagation consists of computing the gradients of the loss with respect to the model parameters (weights and biases) and applying the chain rule of calculus to evaluate the partial derivatives. The derivatives measure the effects of each weight on the overall loss and are used to update the model parameters. For each layer with learnable parameters, two gradients are computed:

1. The gradient of the loss relative to the weights from the filter inputs (i.e., the forward activations) and the gradients of the loss relative to the filter outputs.

2. The gradient of the loss relative to the filter inputs from the filter weights and the gradients of the loss relative to the filter outputs.

Parameter Update The weights and biases are updated using an optimization algorithm, such as Stochastic Gradient Descent (SGD), adjusting the parameters in the opposite direction of the gradient to minimize the loss. SGD is a variant of the gradient descent optimization algorithm that updates the parameters using the gradient computed from a mini-batch of data, which is the typical choice in DNN training. Equation (2.3) implements the SGD algorithm, with θ_t denoting the model's parameter at the current iteration, γ as the learning rate, and $\nabla \cdot L(\theta_{t-1})$ as the gradient of the loss function computed with the parameters at the previous iteration. If γ is too low, the step size is small, and the model may take too many iterations to eventually converge. Conversely, a high γ might cause instability and either induce the optimizer to move away (or opposite) from the ideal direction, maximizing the loss, or move around an optimal minima, failing to reduce even further the loss L. It is possible to tune γ during the training using a schedule that updates it using the model performance or the iteration count, such as the cyclical scheduler proposed in [39].

$$\theta_t = \theta_{t-1} - \gamma \cdot \nabla \cdot L(\theta_{t-1}) \tag{2.3}$$

A momentum $v(t)$ is added to enhance the basic SGD algorithm in Equation (2.4) to accelerate the convergence, especially in scenarios with noisy or sparse gradients, such as the case with compressed or quantized models. The term β is the momentum hyperparameter and $v(t-1)$ is the momentum of the previous iteration. It introduces a form of inertia to the optimization process, helping the model navigate through regions of the parameter space that have flatter gradients or noisy updates. The momentum accumulates a fraction of the previous gradients and combines it with the current gradient to determine the direction and magnitude of the parameter update, removing oscillations and enabling a more efficient convergence.

$$\begin{aligned} v(t-1) &= \beta \cdot v(t-2) + (1-\beta) \cdot \gamma \cdot \nabla \cdot L(\theta_{t-1}) \\ \theta_t &= \theta_{t-1} - \gamma \cdot v(t-1) \end{aligned} \tag{2.4}$$

An additional regularization technique used during the training to prevent overfitting is weight decay, which adds a penalty term λ to the original loss as in Equation (2.5), used to obtain the regularized loss L_{reg}, which is then substituted to the one used in Equation (2.3) or Equation (2.4). The purpose of weight decay is to encourage the model to use smaller weights, which can lead to a simpler and more generalized model. It also has a beneficial effect on compressed models: it increases the numerical stability in case of sparse gradients and makes the model resilient to numerical errors [7].

$$L_{reg} = L + \lambda \cdot \sum w^2 \tag{2.5}$$

Validation The model performance has to be evaluated periodically on a separate validation dataset to tune the hyperparameters, detect and mitigate overfitting/underfitting, and assess when the model has converged. A separate dataset is necessary to avoid any correlation between the tuning procedure and the final test results, to preserve the DNN capability to generalize on new data with the expected performance estimated during the training. Validation enables to identify and correct the causes of poor performance, which could originate from overfitting or underfitting. Overfitting occurs when a model learns the training data too well, i.e., memorizes the dataset, capturing noise and fluctuations in the data rather than learning the patterns. As a result, an overfitted model performs well on the training data but poorly on unseen data (validation or test datasets). Overfitting often happens when the model is too complex relative to the training data available. Weight decay, model compression, and cross-validation can be used to mitigate overfitting. On the other hand, underfitting occurs when a model is too simple to capture the underlying patterns in the data, resulting in poor performance on both the training and new data. An underfitted model may not have enough capacity to represent the complexities of the data, and its predictions may be overly generalized or biased. Increasing the model complexity and the training iterations are a possible solution to reduce underfitting. Finally, convergence is reached when the model's performance on the training and validation datasets stabilizes, indicating that the weights and biases have learned meaningful representations.

Hyperparameter Tuning Hyperparameters are parameters that influence the learning capabilities of a model that are not learned from the data but must be set prior to training and can be adjusted with a feedback from the validation. These parameters guide the training process and impact the performance and behavior of the model. Choosing appropriate hyperparameters is crucial for achieving good generalization on unseen data. It is possible to divide common hyperparameters used in ML into two groups: one affecting the model architecture and the other affecting only the learning process. Starting from the latter, the common hyperparameters involved only in the learning process are:

- *Number of epochs*: The number of times the entire training dataset is passed forward and backward through the model during training.
- *Batch size*: The number of training examples used in a single iteration or mini-batch during gradient descent.
- *Learning rate (γ)*: control the step size in updating the parameters with optimization algorithms like the SGD presented in the paragraph above.
- *Weight decay (λ)*: normalizes the loss function and can prevent overfitting.
- *Dropout rate*: the probability of randomly dropping out a neuron during training to prevent overfitting.

Examples of some hyperparameters that define the model architecture are:

- *Number of hidden layers*: the number of layers in a NN excluding the input and output layer. Adding more layers can increase the task performance in case of underfitting, whereas removing them can prevent overfitting. In the case of model compression, redundant layers can be removed to reduce the computation while retaining the original performance.

- *Layer dimensions*: the weight size of each layer, which determines the number of input and output features processed and produced. Similarly to the number of hidden layers, the weight size can be tuned to remove redundant connections.
- *Data precision*: the numerical precision used in arithmetic operations. Switching from FP32 (IEEE 754 binary32) to FP16 (IEEE 754 binary16) can double the training performance [30, 40] with no sensible accuracy degradation. Runtime selection of the integer quantization policy can further adjust the numerical precision below 16-bit without reducing the model performance.

Test The last step is the final test on new unseen data that is neither correlated to the training or validation dataset. The performance of a DNN model is computed according to metrics of interest for the target task. In this doctoral thesis, only image classification and object detection metrics are reported, as only these two tasks are considered in the presented research work. In image classification, there are two relevant metrics: top-1 accuracy and top-5 accuracy. Top-1 accuracy measures the proportion of test images for which the correct label is the model's top prediction; top-5 accuracy measures the same thing but considering the model's top 5 predictions. In object detection, mean average precision (mAP) is a commonly used evaluation metric to assess the performance of object detection models. mAP is a comprehensive metric that measures the accuracy of an object detection model across different levels of confidence thresholds and for various object classes. It combines precision and recall values calculated at different thresholds, overlapping the predicted position against the ground truth, therefore evaluating the Intersection over Union (IoU), to produce a single aggregate score.

2.1.3 Layers Commonly Used in Deep Neural networks

Fully Connected A fully connected layer, also known as a dense layer, is a structure where every neuron in the layer is connected to every neuron in the preceding layer, as depicted in Figure 2.2, and are typically used at the last layer of the DNN as classifiers [11–13]. Depicted in Figure 2.4, the computation consists of each neuron evaluating a weighted sum of its input and adding a bias to the final weighted sum. The number of output activation Nof is equal to the number of rows of the weight matrix, each one containing Nif columns, and its shape is again a vector. The number of input features Nif and output features Nof are the hyperparameters of the fully connected layer. The bias vector is optional and has Nof parameters, and requires Nof additions. The number of multiplications required to compute the output and the memory footprint of the weights and bias are given by Equation (2.6), using the notation of Figure 2.4.

$$\begin{aligned} \#mult. &= Nof \cdot Nif \\ weight\ mem &= (Nof \cdot Nif + Nof) \cdot precision \end{aligned} \tag{2.6}$$

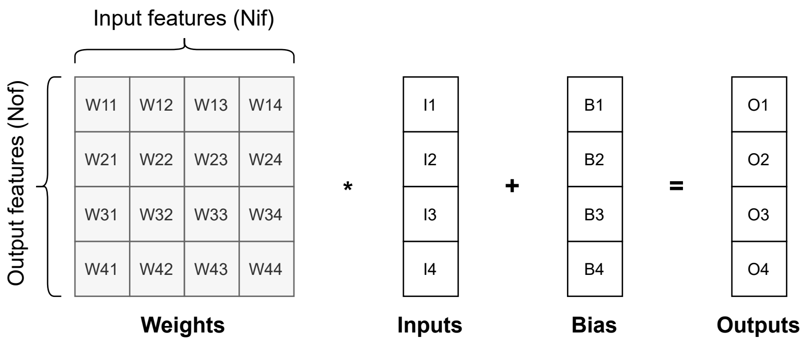


Fig. 2.4 The computation of a fully connected layer with bias can be modeled as a matrix-vector multiplication as $O = WI + B$. The rows of the weight matrix contain the weights associated with each neuron, which are multiplied by the input activations to produce the output.

2D Convolution In a 2D convolutional layer, the neurons are not arranged in a 1D vector, as in a fully connected layer, but in 3 dimensions: width, height, and depth. While the fully connected layer processes input activation as features, in the 2D convolution the input activations are organized in 2D structures called feature maps. The 2D convolution layer is specialized in image processing because image sensory information is usually organized as an array of 2D maps, each containing the pixel value associated with a particular channel (feature). Contrary to fully connected layers, in which each weight is associated with a unique neuron and input activation, in 2D convolutional layers, the weight maps are reused across the entire volume of input activations. Each neuron is connected to a local region of the input volume, which depends on the neuron's receptive field, i.e., the height and width of the 2D weight maps, which are Nkx and NKy. There are always the same number of 2D weight maps and Input Feature Map (ifmap)s, both denoted with Nif. The connections are local in 2D space (along width and height, Nix and Niy) and full along the 3D dimension (Nif). Finally, the depth of the weight volume Nof determines the number of Output Feature Map (ofmap) produced by the convolution, i.e., output channels or output depth. The 2D convolution without bias is depicted in Figure 2.5, using the notation of Table 2.2.

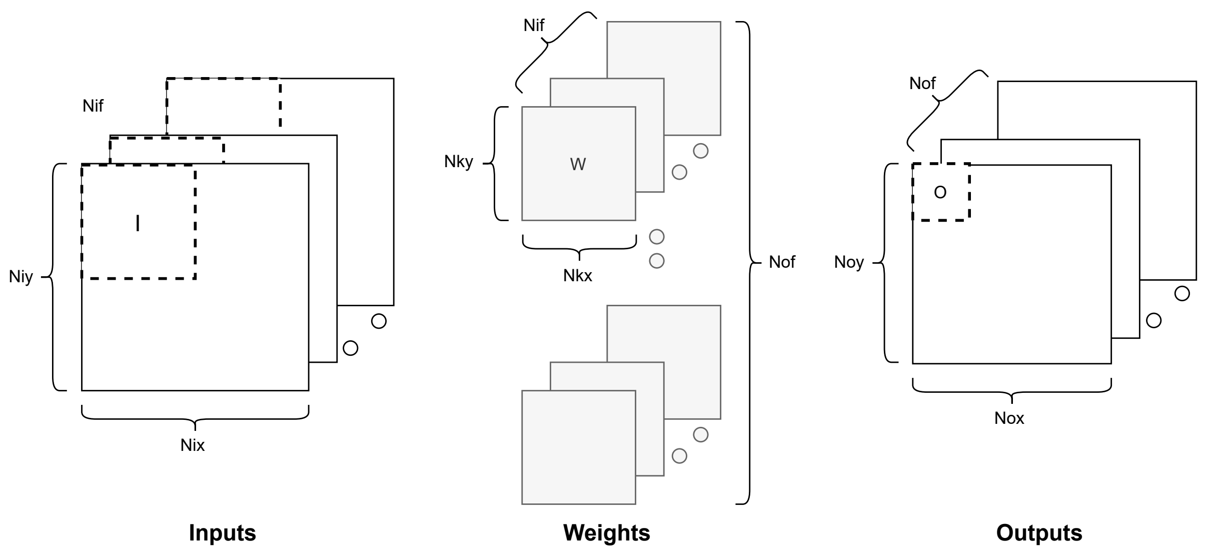


Fig. 2.5 The 2D convolution operates on 3D data organized as a stack of 2D feature maps, with the weights organized in a 4D volume, and produces 3D output activations.

From a computing perspective, computing a convolution means performing a matrix multiplication between a local region of the input volume and a weight map, repeated for all the ifmaps and weight maps at the same depth along the third dimension. The results from all the repeated matrix multiplications are summed to produce an output pixel. Then, the same procedure is applied to another local region of the input volume, sliding the feature map along the height or width of the input. The step size between adjacent local regions of the input, or the step size defining how much the weight maps slide at each iteration, is known as the stride. The vertical stride Sy determines the sliding along the height of ifmaps, whereas the horizontal stride Sx determines the sliding along the width. The stride determines the subsampling of the convolutional layer. A convolutional layer with strides Sx and Sy set to 1 will produce an output volume with the same spatial dimension as the input. A layer with Sx and Sy set to 2 will halve both spatial dimensions, Sx and Sy set to 3 will reduce them to one-third, and so on. Zero padding is added at the border of ifmaps to control the spatial dimension of the output activations (Nox and Noy). Px defines the size of horizontal padding, while Py defines it for the vertical padding. Padding is added to either have the same input and output spatial dimensions or to preserve integer spatial dimensions in case of strides higher than 1. Finally, the dimension of each ofmap generated from a convolutional layer can be evaluated as in Equation (2.7), using the notation of Figure 2.5 and Table 2.2, whereas the number of multiplications and the memory footprint of the weights can be computed as in Equation (2.8)

$$Nox = \left\lfloor \frac{Nix - Nkx + 2 \cdot Sx}{Sx} \right\rfloor \qquad Noy = \left\lfloor \frac{Niy - Nky + 2 \cdot Sy}{Sy} \right\rfloor \tag{2.7}$$

$$\begin{aligned} \#Mult. &= Nox \cdot Nkx \cdot Noy \cdot Nky \cdot Nif \cdot Nof \\ weight\ mem &= Nkx \cdot Nky \cdot Nif \cdot Nof \cdot precision \end{aligned} \tag{2.8}$$

Activation Function The activation function is the non-linearity applied after each fully connected or convolutional layer. The primary purpose of activation functions is to enable DNNs to learn and represent complex, non-linear relationships between input and output data. Without activation functions, DNNs would essentially reduce to a single large linear transformation. Common activation functions used in DNNs are:

- ReLU (rectified Linear Unit): it sets negative inputs to zero while leaving positive inputs unchanged, as in Equation (2.9). Proposed in [35], it accelerates the convergence of the optimizer faster than other activation functions, with fewer computation resources. From a hardware perspective, it requires no complex operation and can be implemented with just a multiplexer controlled by the sign of the input operand, selecting between 0 and the input. On the other hand, ReLU can progressively reduce the value of activations passing through layers and, consequently, zeroing the input and parameter gradients.

$$ReLU(x) = max(0, x) \tag{2.9}$$

- Sigmoid: it constrains the input values to the range (0, 1) as in Equation (2.10), making it suitable for binary classification tasks. However, it suffers from a vanishing gradient problem that is more severe with respect to the ReLU. Moreover, it requires complex hardware to support the computation of the exponential. A variation of the sigmoid, swish, is presented in [41] and used in the EfficientDet [15] implemented in Section 4.1. It solves the vanishing gradient problem using a learnable coefficient β to scale the input, outperforming the ReLU at the expense of higher computational complexity.

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \qquad Swish(x) = x \cdot \frac{x}{1 + e^{-\beta \cdot x}} \tag{2.10}$$

- Softmax, as in Equation (2.11) is commonly used in the output layer for multi-class classification tasks, with C classes, as it normalizes the output scores into a probability distribution over multiple classes.

$$Softmax(x_c) = \frac{e^{x_c}}{\sum_{i=1}^{C} e^{x_c}} \tag{2.11}$$

Pooling Max pooling and average pooling, depicted in Figure 2.6, are typically used after convolutional layers to downsample the spatial dimensions of feature maps. The pooling layers reduce the computational complexity of the network and improve its ability to generalize by capturing the most important features while discarding less relevant information. In max pooling, each region of the ifmap is divided into non-overlapping rectangular regions, the maximum value is retained, while the other values are discarded. In average pooling, the average value of the input features is calculated within the region.

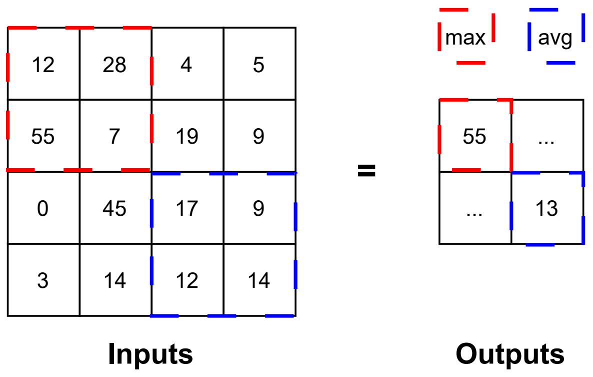


Fig. 2.6 Example showing the application of max pooling and average pooling. The activations within the receptive field are downsampled from 4 to 1.

Normalization Activation normalization helps to stabilize and accelerate the training process. Batch normalization is the most common technique used to address the issue of internal covariate shift, where the distribution of activations within a layer changes during training, leading to slower convergence and degraded performance [42]. Batch normalization acts as a form of regularization by introducing noise during training, similar to dropout, preventing overfitting and improving the generalization performance of the model. Additionally, it reduces the sensitivity of the network to the initialization of parameters, making it less reliant on careful initialization techniques. Presented in Equation (2.12), batch normalization has two learnable parameters β and γ, and two parameters determined during the training, the mean μ and standard deviation σ^2, which are not learned, but computed on the activations statistic. To further stress the importance of batch normalization, it is worth mentioning that it was proven that a randomly initialized DNN, trained by updating only the parameters of each normalization layer in the network while freezing the weights and biases of every other layer, was capable of achieving the same performance of the same DNN trained regularly [43].

$$norm(x) = \frac{x-\mu}{\sigma^2} \cdot \gamma + \beta \tag{2.12}$$

2.1.4 Putting it All Together: Typical Feedforward DNN Architectures

The first DNN that outperformed traditional approaches in image classification tasks was AlexNet [35], which won the ImageNet Large Scale Visual Recognition Challenge in 2012. Contrary to the previous traditional computer vision algorithms, which were based on a mixture of handcrafted features and few small convolutional or fully connected layers, AlexNet learned to extract the features only from the dataset and employed 5 convolutional layers, each one followed by a ReLU activation function and max pooling, and 3 fully connected layers, depicted in Figure 2.7. The network was trained on the ImageNet dataset, which contains 1.2 million images and 1000 classes, dividing the DNN over two GPUs for the training and inference.

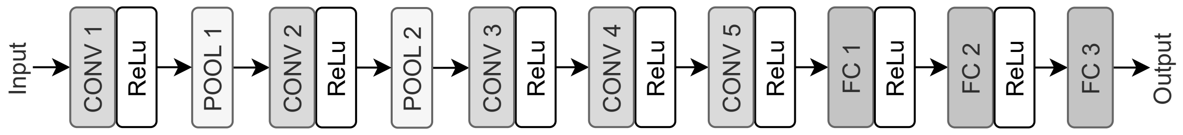


Fig. 2.7 The AlexNet model architecture is built by connecting several feedforward layers. The input is a 224x224x3 image, the output is vector with 1000 scores, one for each class.

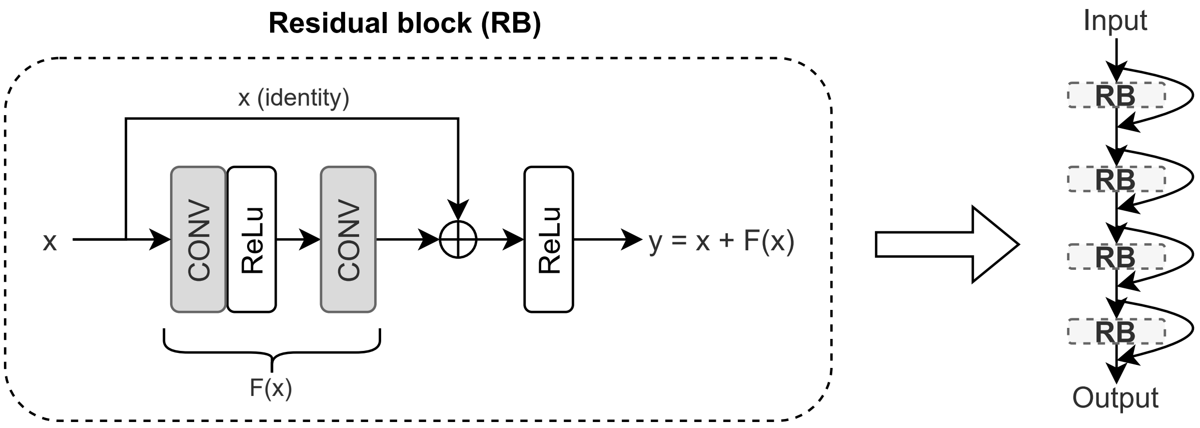


Fig. 2.8 The residual block allows the network to learn the residual function, i.e., the difference between the input and output of a layer rather than the entire function. A residual DNN is built by simply connecting these blocks, maintaining a direct path from the input to the network output. In case of feature scaling or downsampling, a convolutional layer or a max pooling layer is used to adjust the dimensions of the skip connection.

While this model architecture is no longer state-of-the-art, it laid the foundation for modern DNNs. It introduced several key concepts still used today, such as ReLU activation functions, dropout, and data augmentation. The subsequent models, such as VGG [11], GoogLeNet [12], and ResNet [13], improved upon AlexNet by introducing deeper architectures, data normalization, residual connections, and more efficient use of computational resources. In particular, the residual block, depicted in Figure 2.8, is a key component of ResNet and most of modern DNNs. It was introduced to address the vanishing gradient problem that occurs when training very deep networks. This results in unusually high training error, which, on complex datasets like ImageNet, cannot be reduced by applying the techniques presented in the previous paragraphs in case of underfitting. The direct connection between the input and output (skip connection) allows the gradient to flow directly through the network, bypassing the intermediate layers.

2.2 Hardware Acceleration of DNN

DNN hardware acceleration is a key enabler for the deployment of AI applications in a wide range of scenarios, from data centers to edge devices. It can be achieved by using general-purpose hardware such as Central Processing Unit (CPU)s and GPUs or specialized hardware such as TPUs, Field-Programmable Gate Array (FPGA)s, and Application-Specific Integrated Circuit (ASIC)s. Convolutional and fully connected layers represent the most computationally intensive operations, consisting of many (thousands or millions) Multiply-And-Accumulate (MAC) operations, which can be easily parallelized. In the past years, CPUs and GPUs have included specialized instructions and libraries to accelerate DNN inference and training, leveraging the available vector processing units used for Single-Instruction Multiple-Data (SIMD) and single-instruction multiple-threads (SIMT), as shown in Figure 2.9, the latter being specific to general-purpose GPU [44]. Recent server-grade CPU models now include specialized neural processing units to offload DNN computations from the main CPU cores [45, 46]. Similarly, consumer-grade GPUs have been optimized for DNN processing, with the latest models supporting reduced precision arithmetic and few specialized tensor processing units (TPUs) [47, 48]. The server-grade GPUs released in the last years have increasingly included more TPU units, with full and reduced precision arithmetic, to accelerate large language models with billions of parameters [30, 40], and in Table 2.1 are included in the TPU column, as their primary design target is tensor processing. Table 2.1 offers a qualitative comparison of selected hardware architectures that support AI workloads. The term TOPS (tera operations per second) is used instead of TFLOPS (tera floating point operations per second) to compare architectures that do not support native floating point (FP32) arithmetic. This is done for two main reasons: integer processing can achieve the same task accuracy of FP32 at a lower cost, and commercial and research hardware is shifting to fully integer DNN acceleration [49].

Architecture	HPC CPU	GP GPU	TPU	HP ASIC	LP ASIC	FPGA
Flexibility	High	High	High	Low	Low	Low
Power	>300W	>100W	>200 ∼700W	>10W	<1W	<10 ∼50W
Performance	0.1 TOPS	10 ∼100 TOPS	>1000 TOPS	>100 TOPS	<0.1 TOPS	1 ∼10 TOPS
Unit cost	2k€ ∼10k€	1k€ ∼3k€	10k€ ∼40k€	variable	variable	variable
Algorithm	GEMM	GEMM	variable	variable	variable	variable
References	[45, 46]	[47, 48]	[30, 31, 40]	[50, 51]	[52–54]	[55–57]

Table 2.1 Qualitative performance comparison of some hardware architectures used in DNN acceleration.

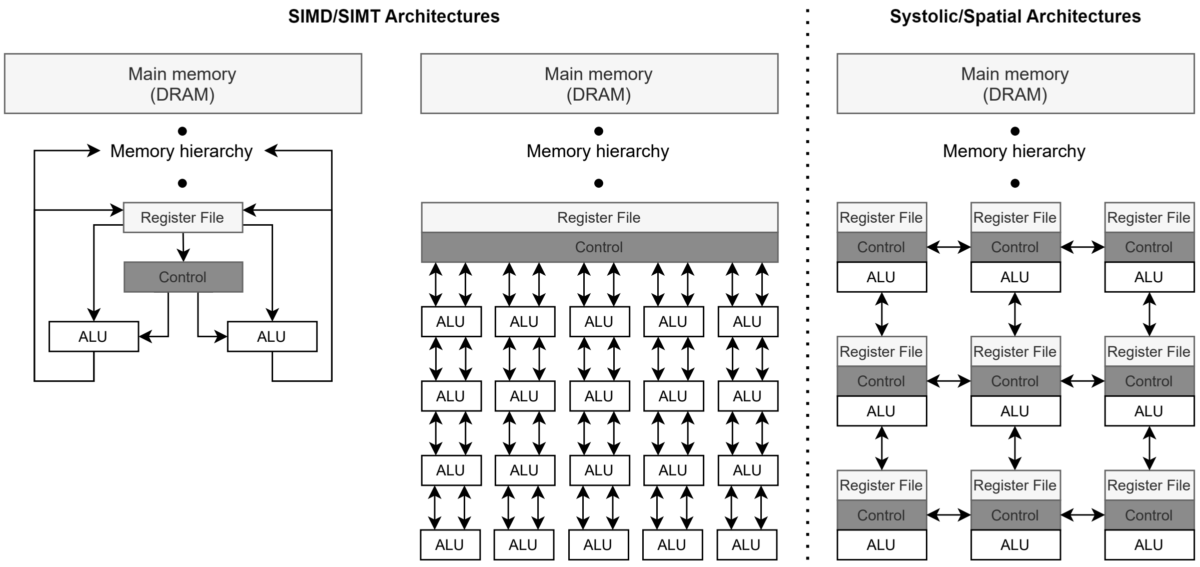


Fig. 2.9 Parallel compute paradigms with centralized and distributed control.

Regarding the three hardware architecture models depicted in Figure 2.9, the SIMD/SIMT architectures represent generic CPU and general-purpose GPU/TPU models, where a centralized control unit is in charge of issuing the instructions to the ALUs, which are usually organized in a grid or in a tree-like structure, that compute the same instruction on different data elements/threads and send the results back to a main register file or data memory. On the other hand, systolic or spatial arrays have a high-level and distributed control embedded inside each processing unit, which also has its own local memory and ALU. The data is streamed through the array, and the computation is done in a pipelined fashion, with the results being sent to the next processing unit in the array.

DNN layers on CPUs, general purpose GPUs, and some TPUs are usually computed with the generalized matrix-matrix multiplication (GEMM) algorithm, depicted in Figure 2.10 [58].

GEMM is not used in low-power ASICs and FPGAs, as it is not memory efficient for these devices due to the data replication in Toeplitz matrices created by expanding the activation and weight tensors of the convolutional and fully connected layers. For the same reason, the GEMM algorithm is also not suitable for sparse networks, which are leveraged to reduce data movement and computation cost by skipping the computation of zeros. Additionally, convolutional layers exhibit an intrinsic redundancy in the computation of the output feature maps, as the same weights are used to compute different output pixels due to weight sharing, explained in Section 2.1. For these reasons, specialized accelerators usually adopt a non-GEMM algorithm and an array-based compute architecture, such as the one of Figure 2.9, in which the data is streamed through a set of processing elements (PEs), which can share both input and output data with local connections, reducing the cost of data movement [52, 59, 60].

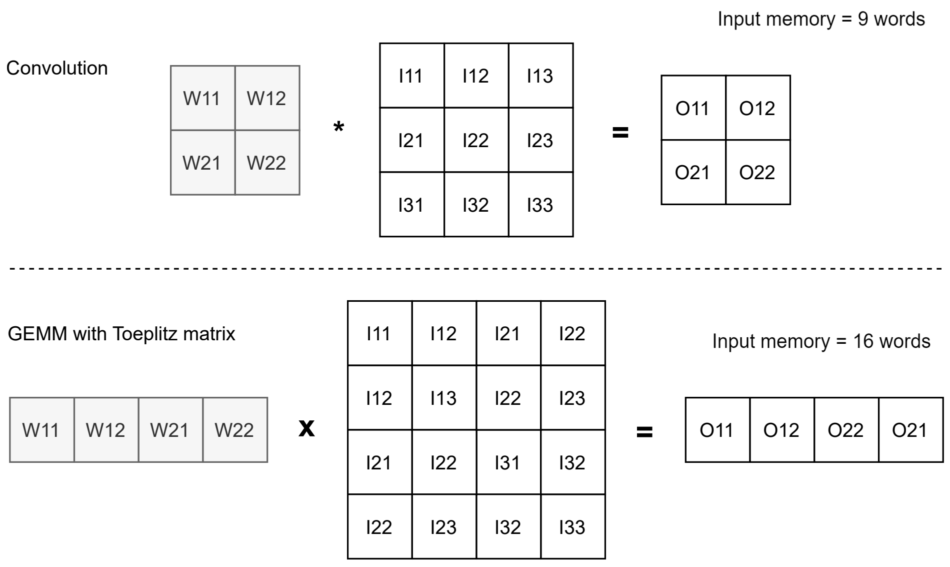

Input memory = 9 words
Convolution
W11 W12 W21 W22
*
I11 I12 I13 I21 I22 I23 I31 I32 I33
=
O11 O12 O21 O22
GEMM with Toeplitz matrix
W11 W12 W21 W22
x
I11 I12 I21 I22 I12 I13 I22 I23 I21 I22 I31 I32 I22 I23 I32 I33
=
Input memory = 16 words
O11 O12 O22 O21

Fig. 2.10 Single channel 2D convolution computed as a matrix multiplication with Toeplitz transformation. The GEMM algorithm, while regularizing the computation in the presence of stride and padding, has a memory overhead, plus an additional computation overhead due to the input, weight, and output tensors transformation, usually done with an *im2col* conversion.

In this doctoral thesis, the focus is on the acceleration of DNNs on low-power ASICs and FPGAs, which are becoming increasingly popular for edge AI applications [53, 55, 57, 61]. The number of IoT devices, wearables, smartphones, and other edge devices is increasing, alongside the amount of sensory data collected [62]. These devices have limited power budgets and resources and must be able to process data locally, without relying on cloud services, for privacy, latency, and bandwidth reasons [63]. In this scenario, low-power architectures, efficient hardware mapping, and model compression techniques are crucial to enable the deployment of AI applications on edge devices [64].

2.2.1 Low Power ASIC: Micro-controllers and Edge Accelerators

Low-power ASICs such as Micro-Controller Unit (MCU)s and reconfigurable accelerators are designed to be energy efficient and to have a small area, which makes them suitable for edge devices where the resource budget is limited. MCUs are widely used in embedded systems, wearables, and IoT devices and are usually equipped with a vectorized Dot-Product Unit (DPU) embedded in the CPU's pipeline [53] or a dedicated accelerator [54] to accelerate DNN inference. The DPU is a specialized unit that can perform a dot product operation on two vectors and accumulate the result in a single cycle. The instructions to compute the DNN are compiled and embedded in the code normally processed by the CPU. Executing the inference on an MCUs with a DPU has the main advantage of allowing the device to be fully flexible and programmable. However, the DNN processing occupies the CPU pipeline, preventing the MCUs from processing other tasks. Moreover, energy efficiency is at the lower end of the spectrum, as data reuse cannot be exploited. In recent years, the highest efficiency in terms of energy and area has been achieved by using systolic or spatial arrays, depicted on the right in Figure 2.9, which are composed of a set of Processing Engine (PE) that are connected in a grid or a tree-like structure, and that can share both input and output data with local connections. Section 2.4 is dedicated to explaining the efficient mapping on such architectures. A dedicated accelerator attached to the CPU core, such as the one developed in [54], can be used to offload the DNN processing from the pipeline.
Coprocessors or memory-mapped accelerators can use the direct memory access unit to access the main memory without wasting any CPU time except for the few instructions executed to configure the accelerator [54].

Regardless of the architecture used, edge devices might need to adapt the DNN parameters to the input data or the environment, which requires online training capabilities [63, 64]. On-device training can be achieved by adopting several strategies, such as fully-quantized forward and backpropagation [65], and direct feedback alignment, which removes the necessity to keep all the intermediate computation that is used to propagate the gradients during the training [66]. This further justifies the need for efficient hardware mapping and model compression to reduce the memory footprint and energy during the inference to make room for the training phase.

2.2.2 FPGA Accelerators

The flexibility of FPGAs, albeit with higher latency and power consumption than ASICs, makes them a popular choice for DNN acceleration, as they can be reconfigured to support different network architectures, and implement custom dataflow and memory access patterns. In this regard, some open-source tools and libraries can be used to generate FPGA accelerators for DNNs, such as FINN-R [57] and BISMO [67], which generates accelerators for quantized or binary networks for Xilinx FPGAs, and Vitis-AI, which supports both quantized and floating-point architecture generation for Xilinx Versal FPGAs [68]. Additionally, the NVDLA platform [55] enables the generation of synthesizable Register-Transfer Level (RTL) and the related software stack that can be compiled and mapped either on NVIDIA's edge computing kits or supported FPGAs. Another relevant FPGA implementation is found in [59], which was one of the first works that leveraged several optimization strategies seen in the next section, such as loop tiling and unrolling. This work is particularly relevant as it shows that the best optimization approach must consider the entire accelerator hierarchy instead of focusing only on the processing elements or the memory hierarchy, as the search for the best communication pattern and the best computation pattern are not orthogonal, as also proven in [52].

2.3 Model Compression

Model compression is a set of techniques that aim to reduce the size of a DNN model, the number of parameters, and the computational complexity while preserving the accuracy of the original model. The main goal of model compression is to enable the deployment of DNNs on low-power devices with limited memory and computational resources and to reduce energy consumption during inference. This section addresses three main techniques: reduced numerical precision (quantization, binarization), weight pruning, and Approximate Computing (AxC).

2.3.1 Reduced Numerical Precision

Quantization

DNNs are usually trained with FP32 numerical precision, which requires 32 bits to represent a single parameter and the availability of high-performance floating point arithmetic units. While FP32 is necessary for training, it is not strictly required for inference, as the model can be quantized to lower precision, such as FP16, INT8, or even binary [69–73]. Therefore, *quantization* is the process of mapping the FP32 weights and activations of a DNN to a lower precision format, such as INT8. The quantization process can be done offline, during the training phase, or online, during the inference phase, and can be applied to the weights, activations, or both. Conversely, *dequantization* converts an integer number back to a real number. There are two main approaches to DNN quantization: post-training quantization and quantization-aware training. Post-training quantization is a simple and fast technique for quantifying the weights and activations of a pre-trained model to a lower precision format without necessarily retraining the model. The DNN's weights can be finetuned after quantization to recover from the accuracy degradation, as done in Section 4.1 for EfficientDet quantized to 16-bit. On the other hand, quantization-aware training consists of training the model with quantized weights and activations from the beginning, including the numerical error in the loss calculation, the backward pass, and, therefore, the parameter update. The DNN then learns to minimize the quantization error while learning to extract the features. The latter approach is done in Section 4.2 to quantize several DNNs to 8-bit with no accuracy degradation compared to the FP32 baseline.

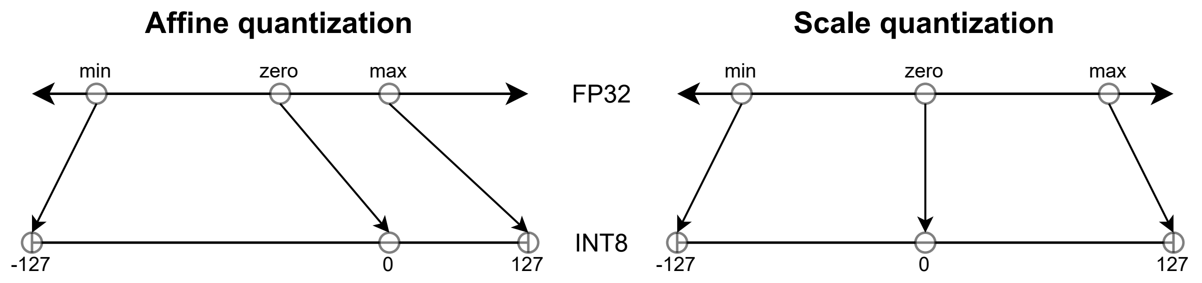


Fig. 2.11 Scale and affine quantization with INT8 precision. The maximum representable range is asymmetric for affine quantization and symmetric for scale quantization. The 8-bit integer range is $[-127, 127]$.The value -128 is not used to preserve symmetry.

In this thesis, only uniform integer scale and affine quantization are presented and are depicted in Figure 2.11, as they are implemented and used in the research work presented in Section 4.1 and Section 4.2. Scale quantization maps real values to integers with a scale transformation, as shown in Equation (2.13), while affine quantization maps real values to integers with a scale and a zero transformation, as shown in Equation (2.14). In both equations, q is the precision, s is the scale, z is the zero, x is the real value, and x_q is the quantized value. In Equation (2.13) α is assumed to be the maximum absolute real value, while in Equation (2.14) α and β are the maximum and minimum real values, respectively. The former does not preserve the zero value, but it is faster than affine quantization, and it is found to lead to approximately the same performance [74].

$$\begin{aligned} s &= \frac{2^{q-1}}{\alpha} \\ x_q &= clip(round(x \cdot s), -2^{q-1}, 2^{q-1} - 1) \\ x &= \frac{x_q}{s} \end{aligned} \tag{2.13}$$

$$
\begin{aligned}
s &= \frac{2^q}{\alpha - \beta} \\
z &= -round(\beta \cdot s) - 2^{q-1} \\
x_q &= clip(round(x \cdot scale + z), -2^{q-1}, 2^{q-1} - 1) \\
x &= \frac{x_q - z}{scale}
\end{aligned}
\tag{2.14}
$$

Alternative Arithmetic Implementations

Relevant alternative arithmetic implementations with reduced precision include binary, ternary, and Winograd convolution. Binarized NNs (BNNs) have weights and activations quantized to 1 bit, which can be either 0 or 1, encoding the values -1 and + 1- [71–73]. With binary arithmetic, multiplications and accumulations can be executed as single-bit operations using and, xnor, and bit-count. For each datatype, the memory footprint can be reduced by up to 32x compared to FP32, while the hardware complexity can be reduced even further, as the arithmetic operations can be implemented with simple logic gates. Naturally, BNNs do not leverage general-purpose hardware and necessitate specialized accelerators [56]. Similarly to BNNs, ternary NNs (TNNs) have weights and activations quantized to 3 values, -1, 0, and +1, and can be implemented with a combination of binary operations and multiplications by 0. In both BNNs and TNNs, the conversion of a single datum from FP32 to binary or ternary, named binarization, can be done with a deterministic (1 if positive or zero, -1 otherwise) or stochastic policy (based on a Sigmoid).

Winograd convolution is a technique that reduces the number of multiplications and additions required to compute a convolutional layer by transforming the kernel and the Ifmap to a smaller domain, where the convolution can be computed with fewer operations [75]. Unlike weight pruning, Winograd convolution does not reduce the number of operations by reducing the number of parameters of the model. Instead, it introduces a memory overhead to store the transformed arrays. The Winograd convolution is particularly useful for small kernels (which are also the most used in CNNs), such as 3x3, and for small batch sizes, as the memory overhead is proportional to the kernel size and the batch size. A downside of the Winograd algorithm is that it is particularly sensitive to quantization errors, and the transformation introduces a small numerical error. A quantized- and Winograd-aware training scheme [76] and the utilization of a complex number system [77] have been proposed to mitigate the error introduced by the quantization and the transformation without introducing a relevant resource overhead.

2.3.2 Weight Pruning

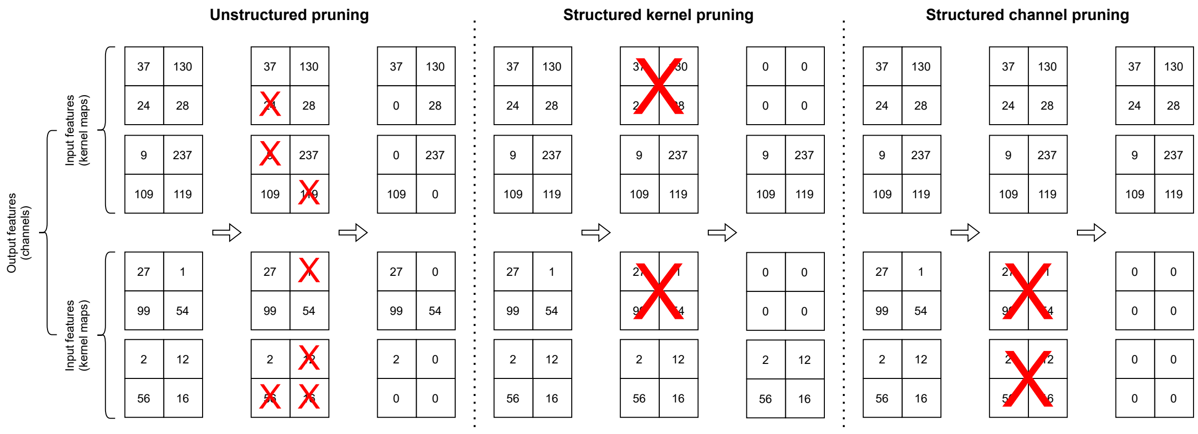

Unstructured pruning
Structured kernel pruning
Structured channel pruning
Output features (channels)
Input features (kernel maps)
Input features (kernel maps)

Fig. 2.12 Comparison between structured and unstructured pruning, single weights or weight maps barred in red are not used in the computation.

Weight pruning is a technique for reducing the memory footprint and computational complexity of a DNN by removing the connections (weights) that are considered irrelevant to the task while preserving the model's accuracy. Pruning can be done by removing small magnitude weights, by measuring the importance of the weights performing a sensitivity analysis and removing those with the lowest influence over the output, or by other means. It can be done during the training, a strategy adopted in several neural architecture search methods [78, 79] or after, as a post-processing step, usually paired with fine-tuning [2, 80]. Pruning can be done offline (static) or online (dynamic): offline pruning removes the connections permanently before the model is deployed, while online pruning removes the connections during the inference phase based on a threshold or a policy. Online pruning has a computational overhead compared to offline pruning, as an evaluation of redundant or low-magnitude connections has to be performed at runtime, but it can be used to dynamically adapt the model to the input data.

Weights can be set to zero, increasing the sparsity of the model, or removed changing the architecture, which is only possible in the case of structured channel pruning. Structured means that the weights are set to zero or removed in groups. As depicted in Figure 2.12, structured pruning can be done at different levels of granularity by removing entire channels (structured channel pruning), entire kernel maps (structured kernel pruning), or single weights (unstructured pruning). These methodologies are the easiest to leverage in hardware, as the regularity of the sparsity pattern can be exploited to reschedule the computation and communication, avoiding not only the computation of the zeroed weights with data-gating but also unnecessary memory accesses. Unstructured pruning, on the other hand, removes individual weights and is more difficult to support efficiently in hardware, as the irregularity of the sparsity pattern can lead to memory fragmentation and inefficient data movement. However, it also has the highest granularity and selectivity, possibly achieving the best tradeoff between compression and task accuracy.

2.3.3 Approximate Computing

AxC in quantized DNNs is the usage of approximate arithmetic units to compute the MACs, using inexact multipliers and adders, which have fewer logic gates than the exact counterparts and, therefore, have a smaller area and power [81]. Approximate Multiplier (AxM)s are usually based on Dadda or Wallace trees [82] implemented with approximate and exact compressors placed in the part of the architecture dedicated to the partial products' reduction [83]. Two main approaches are relevant to this thesis: fixed and reconfigurable approximate units. Logic gates of fixed AxMs are removed at design time, either with hand-crafted [83] or automated [84] strategies. By removing actual transistors, it is possible to reduce latency, dynamic and static power consumption, and the area, achieving the best trade-off between these hardware metrics and the multiplication error. On the other hand, reconfigurable AxMs leverage data-gating to enable or disable approximate compressors, lowering the dynamic energy only, introducing an area, energy and latency overhead due to the control logic. However, the main (and only) advantage of reconfigurable over fixed AxMs is that one single runtime reconfigurable architecture can be used to support algorithms with different error resilience and adapt to different tasks or input data [9], without requiring to replicate the resources [85].

Several error metrics are used in literature to quantify the approximation error of inexact multipliers. In this thesis, the Mean Relative Error Distance (MRED) is used, computed as in Equation (2.15), to evaluate and compare the performance of the AxMs used in Section 4.2 and proposed in [8, 9]. In Equation (2.15), N is the number of possible combinations of input values ($N = 2^{2q}$ with q-bit values), $\hat{o}_i$ and o_i are the i^{th} approximate and exact results, respectively. This metric is adopted in this thesis and previously published works [8, 9] because it is the most commonly used in related literature and allows for reliable comparison of the performance of different AxMs.

$$MRED = \frac{1}{N}\sum_{i=1}^{n}\frac{|\hat{o}_i - o_i|}{|o_i|} \tag{2.15}$$

2.3.4 Optimization Strategies Proposed in Literature

It is possible to organize and classify previous works on pruning, quantization and approximation in three categories according to the optimization strategy, whether hardware evaluation is

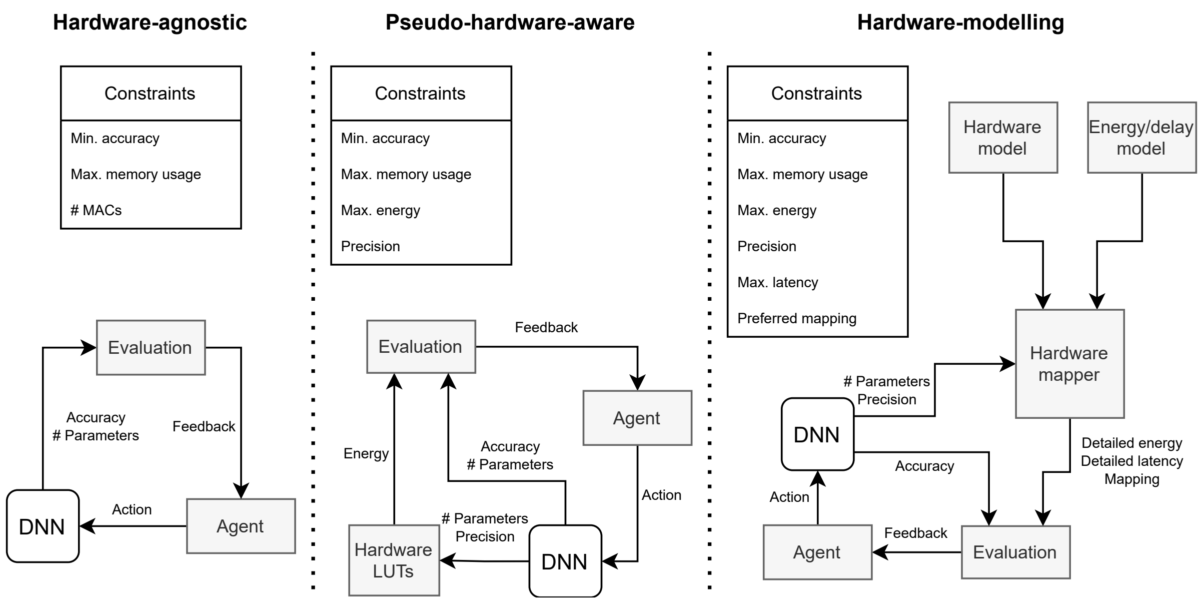


Fig. 2.13 Constraints and optimization loops for hardware-agnostic, pseudo-hardware-aware, and hardware-modeling techniques.

included, and if so, how it is implemented. The three categories, also summarized in Figure 2.13, are hardware agnostic, pseudo-hardware-aware, and hardware-modeling:

- Hardware agnostic: the compression effectiveness is evaluated only as the trade-off between the reduced memory footprint and computational complexity with the accuracy degradation. No information on the target hardware architecture is necessary with this strategy. The main advantage of this strategy is that it has the fastest execution time, as no additional computation is required besides the retraining of the DNN. However, the major weakness is that the compressed model might not be optimized for any specific hardware platform, and performance bottlenecks are neither discoverable nor resolvable. Approximation optimization cannot be done with this strategy, as the hardware constraints are not known, and a simulation/modeling of the hardware is necessary.

- Pseudo-hardware-aware: low-level proxy metrics, such as memory footprint or number of multiplications, or Look-Up Table (LUT)s hardware estimates are used to extract performance metrics such as computation cycles and overall energy. The former does not require any real hardware measurements, as computational complexity and memory size can be evaluated independently from the target accelerator. The latter requires the execution of a reference DNN model on a target hardware to extract baseline performance metrics, which are then scaled with the compression ratio. Memory size, maximum theoretical throughput, and other factors are used as constraints during the compression.

This approach improves on hardware agnostic and it has the second fastest execution time, but has one major weakness: it does not account for compile-time optimizations due to reduced memory requirements and operations. The latter results in sub-optimal performance estimation during the evaluation of the accuracy/performance. Moreover, the generation of LUT requires readily available hardware that cannot be changed later to avoid invalidating the compression strategy. Of the contributions presented in this doctoral thesis, MARLIN [9] falls in this category for what concerns approximation optimization.

- Hardware-Modeling: it leverages the deterministic nature of DNNs and hardware accelerators to evaluate the inference's performance without simulation, estimating the overall data movement and energy/latency cost associated to it. This approach requires a careful definition of the hardware model and scheduling algorithm, plus extensive validation with different hardware architectures. However, it has multiple advantages compared to the other methods: it can predict real-world performance within reasonable margins of error, it allows factoring in the compile-/scheduling-time optimization, it can be used for design space exploration to modify the architecture, and it can be used to analyze memory or computation bottlenecks. The execution time is slower than hardware-agnostic and pseudo-hardware-agnostic, but can still be negligible compared to the DNN training time for a single epoch. Of the contributions presented in this doctoral thesis, Hw-Flow[2], Hw-Flow-Q[3], and Hw-Flow-Fusion[4] fall in this category, as they leverage multiple abstraction levels of hardware models to estimate various performance metrics.

Pruning Techniques

Hardware-agnostic Hardware-agnostic pruning techniques include early works such as [86, 87], in which it is proven that there are redundant neurons that can be removed without reducing the task accuracy and, in some cases, can even improve it. Recent works searched for a policy to select which weights can be removed, for instance, using weight magnitude [88], geometric median heuristic [89], saliency functions [90], or low-rank filter-sharing based on auto-encoders [91]. In [92], unstructured pruning is applied to remove weights with a magnitude below a user-defined threshold inside the kernel maps, and then compressed sparse row/column formats are used to store the sparse maps efficiently.

Pseudo-hardware-aware Structured filter pruning based on the amount of computational complexity and model parameters is done in [93], in which the redundancy of individual kernel maps is evaluated with a Lasso regression, with the compression ratio selected heuristically. In [94], the pruning policy is performed with Reinforcement Learning (RL) agents assigned to each layer that receive at each step a single kernel map as state and determine the pruning policy with the produced action. The reward is a cost function based on the accuracy and low-level metrics, such as computational complexity and overall memory footprint. Contrary to these works above, which include minimal hardware information, other works such as [80, 95, 96] use LUTs with hardware metrics measured from a real hardware accelerator running different DNN models, with metrics generated with interpolation in case of small modifications of the DNN or accelerator. In particular, [95, 96] perform a layer-wise pruning action, making the strategy susceptible to sub-optimal solutions, as inter-layer dependencies are ignored. On the contrary, the strategy proposed in [80] performs the pruning action on the entire DNN, with an RL-agent selecting the sparsity ratio of each layer during each step, with a single episode composed of as many episodes as the number of layers.

Hardware-modeling The authors of [97] propose to use a hardware model to measure energy metrics to tune the pruning strategy. This results in an optimized pruning action compared to [80, 95, 96], with estimated hardware metrics that are closer to real-world performance and reflect the benefits of the different scheduling, which is a consequence of changing the DNN size. Nonetheless, the number of computation cycles, comprehensive of all memory transfer, the overall latency, and data movement are not evaluated or used during the compression and do not influence the sparsity rate. The pruning agent of Hw-Flow [2] improves on [97] by including all relevant hardware metrics and scheduling in the performance estimates, which include energy, latency, and Computation-To-Communication (CTC), the latter accounting for memory bottlenecks, which are the main limitation in data-intensive workloads.

Quantization

Hardware-agnostic Several quantization strategies aimed at reducing only the memory footprint without information on the underlying hardware have been proposed. In [92], weight-only 8-bit quantization is performed after clustering is applied to share multiple connections between the same weights. Huffman coding is then used to further compress the quantized weights. While this approach paved the way for the application of multiple compression techniques (quantization, clustering, pruning, encoding), achieving a 35x compression ratio for AlexNet and 49x for VGG-16, it is not hardware-aware and does not consider any constraints during the compression. Moreover, the activations are not quantized, therefore the need for FP32 arithmetic is still present. Weight quantization has been further explored in [69] and [98], with the former quantizing all the DNN layers except the first and last ones by bounding them between [0,1], using the straight-through estimator proposed in [99] to achieve quantization-aware training. The work of [98] improves on [69] by solving the accuracy degradation issues of the first and last layers. Activation quantization has been successfully implemented in [70], adopting a parameterized activation function based on a ReLU [100] with a learnable clipping factor α, that bounds the quantized activations between $[0,\alpha]$, achieving an optimal trade-off between model size and accuracy with sub-4bit compression and improving over all the previous works. Finally, a quantization strategy for both weights and activations is proposed in [101], in which the selection of the bitwidth, which was the same for every layer of the DNN in previous works, is done by analyzing the Hessian spectrum of each layer and selecting the precision that results in minimal accuracy loss while still reducing the memory footprint. However, the works above still used FP32 operations to process batch normalization layers, preventing the execution on devices without proper hardware support. This is due to the low numerical error tolerance of batch normalization parameters, which causes an abrupt accuracy drop even when switching from FP32 to INT16 quantization. The authors of [102] fold batch normalization in preceding convolutional layers, adjusting the convolution weights and bias to account for the normalization learnable parameters γ and β, collecting data statistics during the full precision training, bounding the normalization's mean and standard deviation during the quantized training.

Pseudo-hardware-aware Works such as [78, 103–105] explore quantized DNNs using different heuristic algorithms, using a LUT approach to evaluate the hardware execution. The authors of [105] propose an RL-agent that, for each step within each episode, parses the DNNs and searches a layer-wise quantization policy. The reward function considers only the accuracy degradation of the quantized DNN, hardware information is included as an external penalty and adjustment of the RL-agent's action. By considering all the layers in a single episode, the RL-agent can discern inter-layer dependencies within the quantization policy and learn the compression ratio to apply to recurring structures within the DNN architecture. For instance, the RL-agent of [105], when used with MobileNet [14], quantizes activations with fewer bits in depth-wise convolutions and more bits in point-wise convolutions when the target hardware is a low-power edge-device and does the opposite when the target hardware is a powerful cloud GPU. The same authors proposed to jointly apply quantization and pruning in [78], in which pre-trained and pre-pruned sub-networks are extracted from a super-network and then quantized. An accuracy predictor, based on a multi-layer perceptron, is also trained to estimate the DNN accuracy without testing to speed-up the search process. While these works apply hardware-software codesign of the DNN model and the accelerator on which will be deployed, their methodology can be enhanced to include mapping and performance bottlenecks evaluation, by evaluating the effects of quantization on the data movement within the accelerator's memory hierarchy. These characteristics are used in the Hw-Flow framework[2–4] to explore the solution space, but there are several stand-alone tools that support highly accurate hardware mapping and energy/latency estimation which can be included in the works mentioned above [106–109].

Hardware-modeling Hardware-modeling quantization strategies are recent and have become increasingly common with the development of hardware-neural architecture search. The authors of [1] quantize activations and weights while updating the hardware model on which the DNN is simulated, using nested genetic agents to search for the optimal quantization and hardware generation policy, trading-off hardware and model complexity with accuracy. The quantized DNN is then executed on a real hardware accelerator synthesized on a Xilinx FPGA with the specifications found during the search. Similarly, in [110] is proposed a framework for a joint search of the precision of the arithmetic units of a hardware accelerator and the quantization of the DNN model that will be executed.

Approximation

Pseudo-hardware-aware Most works on AxC applied to DNN processing follow a LUT-based simulation. This involves generating an LUT containing the output of the AxM for all possible input combinations, which are then read at runtime to compute the approximate MACs. Latency and average dynamic power are also estimated and included, considering the real switching activity occurring during the simulations. These results are usually obtained with a post-synthesis simulation. Each different multiplier configuration requires a unique LUT. The LUT-based approach is necessary to support approximate arithmetics simulation in popular DNN frameworks like Pytorch [29] and Tensorflow [28], using open-source add-ons such as AdaPT [111], TransAxx [112], TFApprox [113], and TFApprox4IL [114]. This is the most straightforward implementation of approximate operations, as the multiplication can be done as a double memory read operation. However, it is not easily scalable, as the LUT size grows exponentially with the bitwidth of the operands, and it is not flexible, as the LUT cannot be modified at runtime because they have to be compiled into executable C++ or CUDA code to leverage the CPU/GPU acceleration. The authors of [115] test a multi-layer perceptron and a LeNet-5 with 600 non-reconfigurable AxMs. Each approximate NN is executed using one of the 600 multipliers for every convolutional layer following five retraining steps, obtaining 600 sets of weights for each retrained model. In [116, 117] is suggested that hardware-aware retraining, while being a time-consuming, resource-intensive strategy, can mitigate the effect of approximation. Mrazek et al. in [85] present ALWANN, a framework for the approximation of DNNs where the assignment of each layer to an AxM, among the eight-bit ones in EvoApproxLib [84], is performed through the multi-objective genetic algorithm NSGA-II. The parameters of each approximate DNN are fine-tuned (updated w.r.t. the starting model) without retraining. Therefore, for each approximate configuration, a new set of weights must be used for each convolutional layer. Similarly, in [118] is presented a methodology to map the layers of a DNN on a group of systolic arrays, each composed of several instances of one AxM. The arrays are part of the same accelerator, with each region processing only one layer of the network. Contrary to [85], the weights are not updated; therefore, the original weights can be used with different approximate configurations. However, using several static multiplier architectures is not scalable for a general-purpose processor due to the area overhead; this method also impacts flexibility in a custom array accelerator. In [119], Tasoulas et al. propose a methodology based on the modification of the bias parameter of each layer to alleviate the approximation error. Similarly to [85], this approach generates a new set of weights for each approximate DNN. However, their multiplier features only three runtime adjustable approximation levels. Moreover, the reconfiguration is handled by chaining two bits to each weight stored in memory, increasing the storage requirements and energy associated with data movement.

Hardware-modeling Alternatives to the LUT-based approach propose to model the error distribution of AxMs and then use it to estimate the overall error of each layer, adding it as a sort of element-wise bias [120–123]. The advantages of modeling the behavior of AxMs without LUT are essentially two: the simulation of the approximate operation can be done as an element-wise operation implemented with native and highly parallelized Pytorch and Tensorflow GPU operations, and secondly, the approach is scalable as the LUT storage, which grows exponentially with the bitwidth is not required. These methods are a promising alternative to enabling the fast simulation of large approximate DNNs, but they are currently limited to a few multiplier architectures with a Gaussian error distribution. Due to this limitation, fast LUT-based GPU acceleration that can support any AxM architecture with any error distribution might be a better trade-off between computational complexity and flexibility than approximate hardware modeling.

2.4 Hardware Mapping

Mapping the execution of NNs on hardware accelerators involves strategically organizing the computation loops within the network to improve performance. This process is also called loop scheduling due to the presence of multiple nested computation loops in typical NN workloads, as the convolutional layer algorithm can be written as in Algorithm 1, using the notation of Table 2.2, with *Sx* and *Sy* denoting the horizontal and vertical stride.

Algorithm 1 Convolutional loop pseudocode (with no bias).

```
1: for each $c$ in $Nox$ do
2:     for each $r$ in $Noy$ do
3:         for each $o$ in $Nof$ do
4:             for each $b$ in $B$ do
5:                 for each $i$ in $Nif$ do
6:                     for each $x$ in $Nkx$ do
7:                         for each $y$ in $Nky$ do
8:                             $out[b][o][r][c] += weight[o][i][x][y] \cdot input[b][i][Sx \cdot r + x][Sy \cdot c + y]$
```

	Input *Width/Height/Channels*	**Output** *Width/Height/Channels*	**Weights** *Width/Height*	**Batch**
Loop size	Nix, Niy, Nif	Nox, Noy, Nof	Nkx, Nky	B
Tile size	Tix, Tiy, Tif	Tox, Toy, Tof	Tkx, Tky	Tb
Unroll size	Pix, Piy, Pif	Pox, Poy, Pof	Pkx, Pky	Pb

Table 2.2 Notation for loop dimensions, tiling factors, and unrolling factors.

Loop scheduling defines the data computation and communication patterns. It is crucial for efficiently utilizing the available processing capabilities of hardware accelerators, such as CPUs[44], GPUs[124], TPUs[31], systolic[59] and spatial arrays[52], to speed-up DNN inference. Loop scheduling optimizations include:

- Tiling/blocking: Dividing the computation into smaller blocks or tiles to fit the input and output data into the available on-chip memory. It helps maximize data reuse and reduce memory access overhead.
- Folding/unfolding: Combining multiple loops (folding) to reduce the memory and computation resources, or splitting a single loop into multiple loops (unfolding), replicating and reusing data to reduce loop control overhead and increasing instruction-level parallelism.
- Loop reordering: Reorganizing data access patterns to enhance spatial locality, reducing memory access latency.

- Vectorization: Utilizing vector processing units to simultaneously operate on multiple data elements, enhancing parallelism and accelerating execution, typical of SIMD architectures.
- Pipeline optimization: Structuring the computation to take advantage of pipeline stages within the hardware, minimizing idle time and improving throughput.
- Parallelization: Identifying and exploiting opportunities for parallel execution of independent operations, distributing the workload across multiple processing units.
- Dependency analysis: Analyzing dependencies between different operations to ensure that parallel execution does not compromise the correctness of the results.

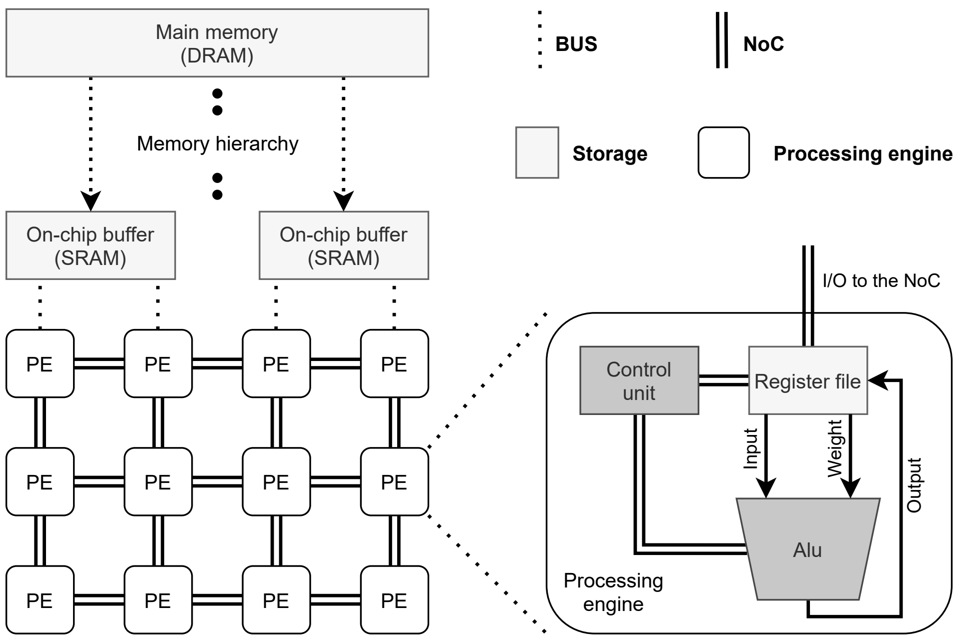


Fig. 2.14 Hardware model with a memory hierarchy composed of three elements (main memory, on-chip memory, and RF) and an array of interconnected PEs.

In this doctoral thesis, the focus is on all the techniques above except for pipeline optimization, which is strictly dependent on the target hardware architecture and requires knowledge of low-level hardware details and tuning knobs, which might not be available to the end-user, such as the case for GPUs and other commercial hardware. The main focus of this doctoral thesis is also on dataflow architectures, such as [52, 125, 126], often referred to as spatial arrays. Contrary to systolic arrays, where the computation patterns are fixed and scheduled by a main control, spatial arrays leverage distributed control units to organize the data processing. Each PE has its internal control unit and Register File (RF), which stores input and partial results. A typical accelerator with a spatial array includes a memory hierarchy, normally composed of the system's main memory (off-chip memory, Dynamic Random-Access Memory (DRAM)), the internal main memory (on-chip memory, Static Random-Access Memory (SRAM)), and local memory, often implemented as a RF. Finally, point-to-point communication within the PEs array can be implemented with a Network-on-Chip (NoC), time-multiplexed, or other interconnection topologies. The overall architecture is depicted in Figure 2.14. The mapspace on dataflow architectures is considerably larger than the one of systolic arrays because more hardware settings are exposed and need to be configured [2–4, 106, 107, 109]. The scheduling is usually done by minimizing cost functions that associate estimated performance metrics with the total data movement required to execute the workload. These cost functions comprehend all significant energy and latency contributions due to data movement, such as MAC energy, memory read/write energy at every level, and communication through the NoC. Frameworks such as [2–4, 106, 107, 109] create an abstract hardware model of the target accelerator and search the mapspace by optimizing the aforementioned cost function for energy, latency, or both. These loop scheduling optimization frameworks aim to fine-tune the arrangement of communication and computation within loops to extract maximum parallelism and minimize resource idle time on hardware accelerators, ultimately improving the overall efficiency and speed of DNN execution. They aim to minimize the inference energy and latency, optimizing the utilization of the available resources. These goals can be achieved by leveraging the high redundancy that is present in typical DNN workloads. This doctoral thesis mainly focuses on convolutional layers since they represent the majority of computation and exhibit high redundancy due to weight sharing, which means that each kernel map is reused to compute many matrix multiplications for each corresponding input feature map, with any batch size. The objective of optimal loop scheduling is to maximize the data reuse with redundant computation, leveraging four opportunities in the processing of a convolutional loop:

- **Input reuse**: each input pixel is reused during the convolution to generate *Nof* feature maps.
- **Output reuse**: each output pixel is reused during the accumulation of *Nif* feature maps.
- **Kernel reuse**: kernel weights are reused *Nox* * *Noy* times over each input feature map.
- **Convolutional reuse**: each input pixel is reused *Nkx* * *Nky* times for a single Hadamard product at a time.

There is an additional reuse opportunity that consists of reusing the intermediate pixel volume between layers, and in this doctoral thesis is presented in Section 3.2.

2.4.1 Loop Blocking: Temporal and Spatial Tiling

This section focuses on loop tiling and loop folding/unfolding, adopting a terminology used in different works, taken from [2–4, 52, 106, 107, 109]. Tiling means to divide in smaller blocks the computation of a workload. In order to simplify the analysis of the type of loop optimization performed at a particular memory level, two terms and definitions are borrowed from Timeloop [107] when referring to tiling and reordering applied to different loops executed at different levels of the hardware architecture: **temporal** and **spatial**. Each memory level within the hierarchy has a spatial and temporal loop scheduling space, tiling and reordering can be applied to both spaces and affect the final mapping differently. When **tiling** is applied to a temporal level, it defines the amount of data stored in the corresponding level and that moved between different levels within the accelerators' memory hierarchy. On the other hand, when it is applied to a **spatial** level, it defines the degree of unrolling/unfolding, data sharing, and replication. In this doctoral thesis, temporal tiling parameters are also called *tiling factors* and spatial tiling parameters are also called *unrolling factors*. Temporal tiling divides the data volume into sub-volumes at a particular memory level, as depicted in Figure 2.15. For each temporal tiling level, the memory footprint of the tiled volume identified with the tiling factors *Tb*, *Tof*, *Tif*, *Tox*, and *Toy* is defined as the buffer space required to store I inputs, W weights, Opartial sums or outputs used during the computation. The memory footprint can be evaluated as shown in Equation (2.16), *precision* refers to the bit-width of the datatype.

$$\begin{aligned} I_{mem} &= Tix \cdot Tiy \cdot Tif \cdot Tb \cdot input\_precision \\ O_{mem} &= Tox \cdot Toy \cdot Tof \cdot Tb \cdot output\_precision \\ W_{mem} &= Nkx \cdot Nky \cdot Tif \cdot Tof \cdot weight\_precision \end{aligned} \tag{2.16}$$

To check if a tiling set for a particular memory level is valid, the sum of the buffered volumes in Equation (2.16) must be less or equal to the total buffer size. When a tiling factor is equal to its loop size, the entire loop is accessed at once, and so, at least from a logic point of view, it can be removed since its index variation does not affect variables that used to depend on it anymore. This strategy is used in the inter-layer scheduling presented in Section 3.2 to simplify the data dependency analysis. It is also worth mentioning that in this doctoral thesis and the related published research work [2–4], the kernel spatial dimensions are never tiled, as the kernel is always accessed in a single shot. This is because a tiling on the kernel would introduce a significant reuse degradation, as it can be easily used to compute multiple adjacent output pixels due to the weight sharing property of convolutional layers.

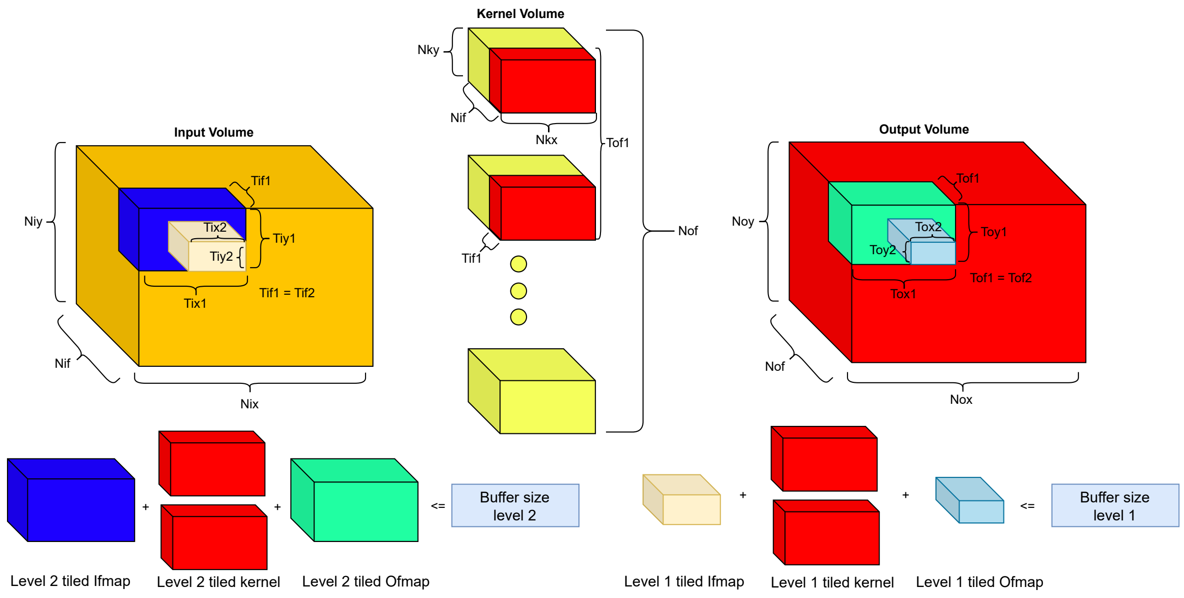


Fig. 2.15 Temporal tiling applied to a memory hierarchy that comprises two nested buffers (on-chip) connected to the main memory (off-chip). The order is: off-chip $\rightarrow$ on-chip level 2 $\rightarrow$ on-chip level 1.

Spatial tiling, or unrolling/unfolding, partitions communication or computation across multiple buffers or PEs at the same level of the memory hierarchy. Similar to temporal tiling, it is possible to have nested spatial tiling levels. For instance, in an accelerator such as the one in Figure 2.14, spatial tiling applied at the on-chip memory would partition the data stored in the memory banks for the computation and the space allocated to save the results. Continuing the explanation on spatial tiling at the memory level, in particular in the case of multi-core architectures such as the two examples depicted in Figure 2.16, partitioning the layer can improve the throughput by assigning each core a different set of blocks to execute. Partitioning is done by unrolling a loop of Algorithm 1 and assigning unique and replicated data to each dedicated memory level. The available cores process a disjoint set of the original partitioned data, and communication occurs as a parallel broadcast, whereas the shared data is available to all the cores. The limit between unified and shared/dedicated data in Figure 2.16 corresponds to the level where a single on-chip memory feeds multiple smaller memories, partitioned data is saved in dedicated memories, whereas shared data is saved in shared memories.

Two memory-level spatial tiling schemes can be adopted, namely *kernel* and *ofmap* partitioning, depicted in Figure 2.17 and Figure 2.18, based on the analysis presented in [127].

For *Kernel* partitioning, the weights are divided into different sets, each one with different output channels but keeping the entire Nif and kernel map (Nkx, Nky) dimensions. ifmaps are shared between the N cores, ofmap, and weights are partitioned according to the unrolling level of the output channels. For *Ofmap* partitioning, ofmap Nox and Noy loops are split between N

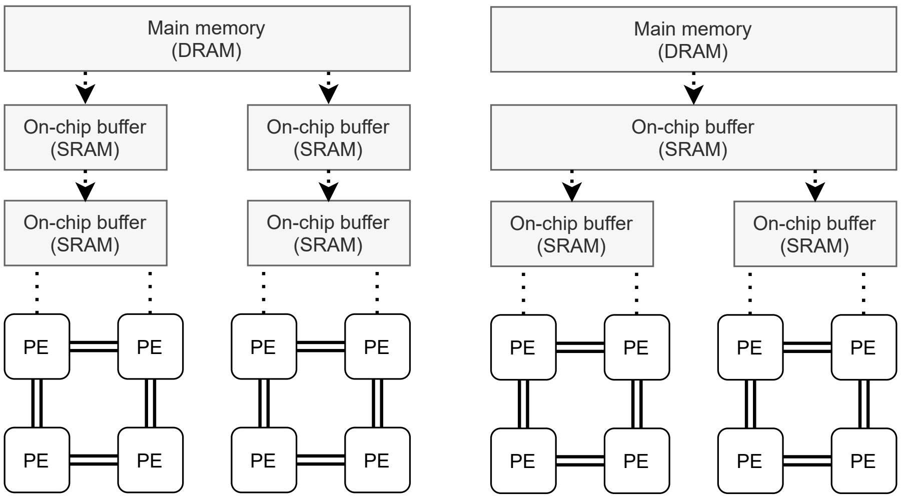


Fig. 2.16 Memory hierarchy with multiple levels of on-chip banks. The hierarchy on the right could represent a multi-core MCU with a dedicated L1 cache for each core and a shared L2 cache.

cores, each one taking different portions of the feature maps and the entire channel dimension, as depicted in figure 2.18. In this case, the kernel is shared between the N cores, while ifmaps are partitioned following the same scheme as the ofmaps. ifmap partitioning on the channel dimension is avoided as it would cause a communication overhead due to data dependencies between the partial sums computed in different cores. After the partial convolution of input and kernel maps is computed on any core, before saving the final ofmaps back to the main memory, a final evaluation step is required to move all the partial sums generated in all the different cores into a single one, execute the accumulation and write the results.

Spatial tiling can be applied applied at the PE level unrolling the computation of the data contained in the memory banks. The unrolling is usually done over one or more spatial dimensions of the array of Figure 2.14. For instance, considering the hardware model above and an unrolling factor of 2 over the output channel dimension, half of the array (2 rows and 4 columns or vice-versa) would process in parallel the same input with two different sets of weights and produce the output of two different regions of the overall output volume. In order to provide a more meaningful example, a row-stationary dataflow is considered since it is well documented, and there are open-source tools that can be used to validate this example [52, 107]. To demonstrate the effect of spatial tiling and reordering, only the meaningful constraints and parameters are reported. The complete constraints can be found in [52, 107], as an analysis here would be out of the scope of this work. The unrolling factors of Figure 2.19 are influenced by the temporal and spatial tiling done during the scheduling of the memory hierarchy. The availability of data ready to be processed defines the size of the PE set allocated and the degree of unrolling. Intuitively, more on-chip memory above the compute units allows to feed more

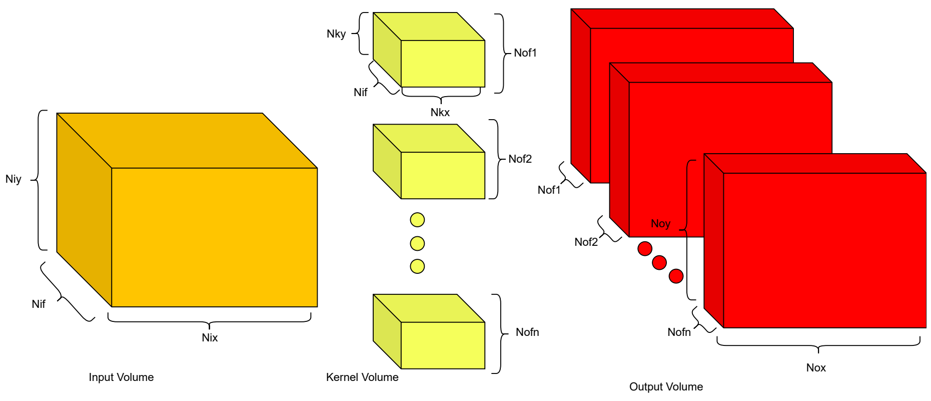


Fig. 2.17 Example of Kernel partitioning. Each core computes a portion of the output channels using the same input data and a unique set of kernels. No data dependencies are present with this scheme. The cores should receive a balanced workload requiring the same amount of compute, in order to have a similar latency and to avoid stalls.

PEs with constant data, but also that the bandwidth requirements increase and the maximum theoretical performance might not be reached without an adapt interconnect strategy. This problem can be highlighted using the CTC ratio explained later in this section or by reshaping both the communication and computation pattern by reordering the nested loops, as detailed in Section 2.4.2.

2.4.2 Loop Reordering

Loop reordering, when applied to a temporal level, defines the order in which data is accessed at a specific memory level. When reordering is applied to a spatial level, it defines the dimension of unrolling over a hardware spatial dimension (height, width). The loop order is correlated with different reuse opportunities according to the relative position of nested loops. An example of a generic access pattern on a 3 level memory hierarchy is depicted in Figure 2.20. Some loop orders like the one adopted in row-stationary, might force zero partial output movement between the on-chip and off-chip buffer.

Considering Algorithm 1, there are hundreds of permutations of the five outermost loops, but by analyzing the loop orders that significantly affect the communication, it is possible to consider only 3 main temporal reordering schemes [128]:

- **Output-weight reuse oriented (OWR)**: reuse the ofmap pixels over the entire output channel dimension Nif. Weights and ifmaps are read multiple times (Nof/Tof) for each iteration. Output pixels are moved back to the DRAM only after all the partial sums have been accumulated. Kernel volume is reused for any batch size. On a multi-

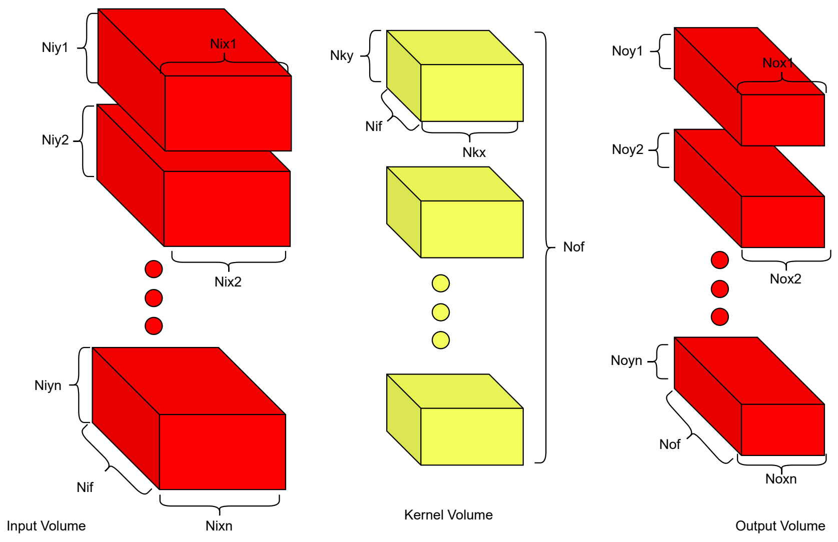


Fig. 2.18 Example of ofmap partitioning. Each core receives the same set of kernels but a different tile of the input data. The input is tiled on the horizontal and vertical dimensions and not on the channel dimension to avoid data dependencies and redundant communication and computation between the cores. Similarly to the other case, the workload must be evenly partitioned to avoid stalls.

memory hierarchy, this is equivalent to having partial sums stored at a certain memory level that are moved to another level only once the output channel loop of the current level has been completed. Weights depend on channel indices and on the indices of the horizontal/vertical loops of the ofmap. ofmaps depend on the output channel and batch/horizontal/vertical loops indices, ifmaps depend on the same indices of the ofmaps plus the input channel index.

- **Weight reuse oriented (WR)**: reuse the kernel maps over the entire ofmap dimension. This reuse is batch-independent, as the batch loop is below the input and output channel loop. ifmaps and ofmaps are read multiple times for each iteration. On a multi-memory hierarchy, this is equivalent to having weights stored at a certain memory level, which are fetched from the level above only once the ofmap has been cycled at least once. Weights depend on output/input channels indices only, ofmaps depend on the input/output channel and batch/horizontal/vertical loops indices, and ifmaps depend on the same indices of the ofmaps plus the input channel index.

- **Input reuse oriented (IR)**: reuse the same ifmap pixels over the entire output channel dimension Nof. ofmaps and weights are read multiple times for each iteration. On a multi-memory hierarchy, this is equivalent to fetching the Input pixels only once from the memory above for an entire input channel loop execution. Weights depend on output/input channels indices and on the indices of the horizontal/vertical loops of the

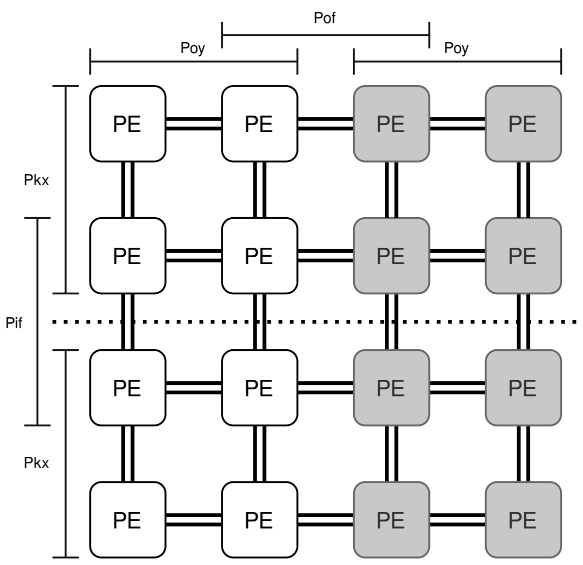


Fig. 2.19 Example of a row-stationary mapping with a 4x4 PE array with Pky = 2, Poy = 2, Pof = 2, and Pif = 2. PEs with the same color process the same output channels; the dotted line separates PEs computing different input channels.

ofmaps. ifmaps depend on the input channel and batch/horizontal/vertical loops indices, ofmaps depend on the same indices of the ifmaps plus the output channel index.

Switching dynamically between these three schemes allows to schedule the entire DNN model exploiting the reuse opportunities of different layers. As a trivial example, layers with a very large kernel can benefit from **OWR** and **WR** schedules, whereas layers with large feature maps and more input channels than output channels will benefit the most with an **IR** schedule. For any loop order and set of tiling factors, it is possible to define the fetch and write occurrences of ifmaps, ofmaps and weights between two connected memory levels within the hierarchy. Given the reuse schemes **IR**, **OWR** and **WR** it is possible to define the unique accesses for each of these datatypes, i.e. how many times a pixel/weight is moved across the memory hierarchy to generate a disjoint set of outputs, by considering which variable is depending on which loop order, as in Equations (2.17),(2.18), and (2.19), adapted from [59, 60]. I_{fetch}, K_{fetch}, and O_{fetch} are the number of unique fetch operations of the input, weight and

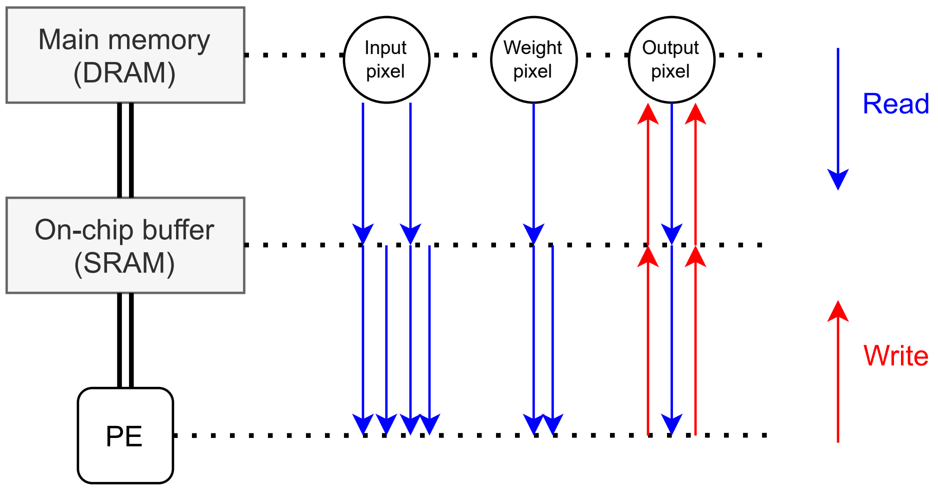


Fig. 2.20 A generic data movement of input, weight and output pixels on the memory hierarchy. By changing the loop order that determines this access pattern, prioritizing the reuse of partial outputs, it is possible to eliminate 2 write operations and 2 read operations.

output pixels respectively, whereas O_{write} is the number of unique write operations.

$$\begin{aligned}
I^{\text{IR}}_{\text{fetch}} &= \frac{Nif}{Tif} \cdot \left\lceil \frac{Noy}{Toy} \right\rceil \cdot \left\lceil \frac{Nox}{Tox} \right\rceil \cdot \frac{B}{Tb} \\
K^{\text{IR}}_{\text{fetch}} &= \frac{Nof}{Tof} \cdot \frac{Nif}{Tif} \cdot \left\lceil \frac{Noy}{Toy} \right\rceil \cdot \left\lceil \frac{Nox}{Tox} \right\rceil \cdot \frac{B}{Tb} \\
O^{\text{IR}}_{\text{fetch}} &= \frac{Nof}{Tof} \cdot \left(\left\lceil \frac{Nif}{Tif} \right\rceil - 1 \right) \cdot \frac{Noy}{Toy} \cdot \frac{Nox}{Tox} \cdot \frac{B}{Tb} \\
O^{\text{IR}}_{\text{write}} &= \frac{Nof}{Tof} \cdot \left\lceil \frac{Nif}{Tif} \right\rceil \cdot \frac{Noy}{Toy} \cdot \frac{Nox}{Tox} \cdot \frac{B}{Tb}
\end{aligned} \tag{2.17}$$

$$\begin{aligned}
I^{\text{OWR}}_{\text{fetch}} &= \frac{Nif}{Tif} \cdot \left\lceil \frac{Nof}{Tof} \right\rceil \cdot \left\lceil \frac{Noy}{Toy} \right\rceil \cdot \left\lceil \frac{Nox}{Tox} \right\rceil \cdot \frac{B}{Tb} \\
K^{\text{OWR}}_{\text{fetch}} &= \frac{Nof}{Tof} \cdot \frac{Nif}{Tif} \cdot \left\lceil \frac{Noy}{Toy} \right\rceil \cdot \left\lceil \frac{Nox}{Tox} \right\rceil \cdot \frac{B}{Tb} \\
O^{\text{OWR}}_{\text{fetch}} &= 0 \\
O^{\text{OWR}}_{\text{write}} &= \frac{Nof}{Tof} \cdot \frac{Noy}{Toy} \cdot \frac{Nox}{Tox} \cdot \frac{B}{Tb}
\end{aligned} \tag{2.18}$$

$$
\begin{aligned}
I^{\mathrm{WR}}_{\mathrm{fetch}} &= \frac{Nif}{Tif} \cdot \left\lceil \frac{Nof}{Tof} \right\rceil \cdot \left\lceil \frac{Noy}{Toy} \right\rceil \cdot \left\lceil \frac{Nox}{Tox} \right\rceil \cdot \frac{B}{Tb} \\
K^{\mathrm{WR}}_{\mathrm{fetch}} &= \frac{Nof}{Tof} \cdot \frac{Nif}{Tif} \\
O^{\mathrm{WR}}_{\mathrm{fetch}} &= \frac{Nof}{Tof} \cdot \left(\left\lceil \frac{Nif}{Tif} \right\rceil - 1 \right) \cdot \frac{Noy}{Toy} \cdot \frac{Nox}{Tox} \cdot \frac{B}{Tb} \\
O^{\mathrm{WR}}_{\mathrm{write}} &= \frac{Nof}{Tof} \cdot \left\lceil \frac{Nif}{Tif} \right\rceil \cdot \frac{Noy}{Toy} \cdot \frac{Nox}{Tox} \cdot \frac{B}{Tb}
\end{aligned}
\tag{2.19}
$$

The **OWR** and **WR** strategies maximally reuse ofmaps and weights respectively, leveraging batch reuse. These two schemes are the best choice for maximum energy efficiency in case of large batches, whereas the **IR** strategy are effective for unitary batch, or more in general in layers where the input volume is predominant. Finally, the total amount of fetch/write data is evaluated by multiplying the buffer occupation by the invocation count for each datatype. Therefore, using again the notation of [59], the total communication volume for the three schedules is given by Equation (2.20).

$$
\begin{aligned}
comm.\ volume^{\mathrm{IR}} &= I_{mem} \cdot I^{\mathrm{IR}}_{\mathrm{fetch}} + W_{mem} \cdot K^{\mathrm{IR}}_{\mathrm{fetch}} + O_{mem} \cdot (O^{\mathrm{IR}}_{\mathrm{fetch}} + O^{\mathrm{IR}}_{\mathrm{write}}) \\
comm.\ volume^{\mathrm{OWR}} &= I_{mem} \cdot I^{\mathrm{OWR}}_{\mathrm{fetch}} + W_{mem} \cdot K^{\mathrm{OWR}}_{\mathrm{fetch}} + O_{mem} \cdot (O^{\mathrm{OWR}}_{\mathrm{fetch}} + O^{\mathrm{OWR}}_{\mathrm{write}}) \\
comm.\ volume^{\mathrm{WR}} &= I_{mem} \cdot I^{\mathrm{WR}}_{\mathrm{fetch}} + W_{mem} \cdot K^{\mathrm{WR}}_{\mathrm{fetch}} + O_{mem} \cdot (O^{\mathrm{WR}}_{\mathrm{fetch}} + O^{\mathrm{WR}}_{\mathrm{write}})
\end{aligned}
\tag{2.20}
$$

$$
\begin{aligned}
LT^{\mathrm{out}}_{\mathrm{x}} &= Nox - Tox \cdot \left(\left\lceil \frac{Nox}{Tox} \right\rceil - 1 \right) \\
LT^{\mathrm{in}}_{\mathrm{x}} &= (LT^{\mathrm{out}}_{\mathrm{x}} - 1) \cdot Sx + Nkx \\
R_x &= \left\lceil \frac{Nox}{Tox} \right\rceil - 1
\end{aligned}
\tag{2.21}
$$

The ifmap fetch volume must be corrected in case of $Tix \ll Nix$ and $Tiy \ll Niy$, adopting the set of Equations (2.21), which are used to compute the adjusted ifmap fetch (*adj_if_f*), ratio of Equation (2.22) (for conciseness, only the equations for x are reported, the equations for y are identical). This correction accounts for smaller tiles LT on each spatial dimension and the number of repetitions R, and is necessary to estimate the correct communication volume. The variable $\mathrm{reps_x}$ ($\mathrm{reps_y}$) represents the amount of repeated tiles over the horizontal (vertical) dimension. The final ifmap fetch then can be evaluated as in Equation (2.23).

$$
adj\_if\_f = Tix \cdot Tiy * R_x \cdot R_y + LT^{\mathrm{in}}_{\mathrm{x}} \cdot Tiy \cdot R_y + LT^{\mathrm{in}}_{\mathrm{y}} \cdot Tix \cdot R_x + LT^{\mathrm{in}}_{\mathrm{x}} \cdot LT^{\mathrm{in}}_{\mathrm{y}}
\tag{2.22}
$$

$$\begin{aligned}
Ifmaps\ fetch_{IR} &= \frac{Nif}{Tif} \cdot adj\_if\_f \cdot \frac{B}{Tb} \\
Ifmaps\ fetch_{OWR} &= \frac{Nif}{Tif} \cdot \left\lceil \frac{Nof}{Tof} \right\rceil \cdot adj\_if\_f \cdot \frac{B}{Tb} \\
Ifmaps\ fetch_{WR} &= \frac{Nif}{Tif} \cdot \left\lceil \frac{Nof}{Tof} \right\rceil \cdot adj\_if\_f \cdot \frac{B}{Tb}
\end{aligned} \tag{2.23}$$

As stated above, spatial reordering defines the dimension on which a particular loop is unrolled. In this doctoral thesis, the term dataflow is used to identify the unique computation pattern resulting from a particular spatial order applied at the PE array level, or inside the PEs. There are three main dataflow schemes that are usually found in research and commercial hardware architectures: input stationary, weight stationary, and output stationary. The term stationary indicates which type of data is maximally reused at the PE level, i.e., exhibit the lowest data movement. Figure 2.20 shows an example where weights are the data type with the maximum reuse across the hierarchy and also at the PE level, which could represent a weight stationary implementation. The characteristics of the three dataflows are listed below, with the considerations on each one extracted from [52, 128–130]:

- **Weight stationary**: the weights are stored in the PE memory and reused with input activations that are sourced from other levels of the memory hierarchy, whereas the partial outputs are accumulated across different PEs. This dataflow is particularly effective for weight-dominated workloads but also performs well on output-dominated ones.
- **Output stationary**: partial outputs are accumulated internally on each PE, which computes a unique portion of the output activations. Weights and input activations are sourced from other levels of the memory hierarchy. The same considerations done on the effectiveness of weight stationary apply to this dataflow.
- **Input stationary**: input activations are reused in each PE, weights are sourced from memory, and partial outputs are stored/sourced to/from other memory levels. This dataflow is outperformed by the other two due to data communication and replication overhead due to the data dependency between different neighbor regions of the output, which depends on overlapping input regions, as already highlighted in the case of ifmap partitioning.
- **Row stationary**: a mixture of the three dataflows presented above. Small regions of input rows of one or more feature maps are reused at the PE level, which also reuses multiple rows of the weights to produce multiple output pixels.

2.4.3 Computation-to-Communication Ratio

The peak performance of an hardware accelerator is bounded by either the computational rooftop, which is the theoretical maximum number of operations per cycle, or the memory bandwidth, which limits the accelerator's performance, preventing the processing units from working at full regime with a constant flow of new data. The CTC ratio defines the amount of computation attainable with a certain amount of communication and can be increased by maximizing data reuse and reducing data movement between different memory levels. Equation (2.24) represents a CTC ratio formula based on the one presented in [131]. γ represents a bandwidth-correction term to account for the burst-length of the memory transfers, which penalizes small memory accesses and favours transfers that saturate (fully use) the burst. The numerator is the number of operations/complexity of a particular workload. The denominator is the overall memory access with bandwidth scaling for input, weights, and outputs for a particular workload.

$$CTC = \frac{2 \cdot Nox \cdot Noy \cdot Nkx \cdot Nky \cdot Nof \cdot Nif}{\gamma_{\text{ifm}} \cdot volume_{\text{ifm}} + \gamma_{\text{wght}} \cdot volume_{\text{wght}} \cdot \gamma_{\text{psm}} \cdot volume_{\text{psm}} \cdot \gamma_{\text{ofm}} \cdot volume_{\text{ofm}}} \tag{2.24}$$

2.4.4 Energy and Latency Models

Typically, loop scheduling frameworks use LUTs with the energy cost of each atomic operation, i.e., memory access or arithmetic, to produce relevant hardware metrics. The cost of all data movement at each memory level can be evaluated by accumulating the fetch/write count relative to each data type, multiplied by the associated energy cost. Below is reported how to evaluate the overall data movement energy, denoting with *i-1* the current memory level and *i* the one above, the normalized energy access per operation can be evaluated as in Equation (2.25).

$$\begin{aligned} energy_{\text{read}}^{\text{i-1\_to\_i}} &= energy_{\text{read}}^{\text{i}} + energy_{\text{write}}^{\text{i-1}} \\ energy_{\text{write}}^{\text{i-1\_to\_i}} &= energy_{\text{write}}^{\text{i}} + energy_{\text{read}}^{\text{i-1}} \end{aligned} \tag{2.25}$$

The energy of arithmetic operations can be averaged and fused into the write energy of partial sums. The overall read energy is then estimated by multiplying the total read volume by the normalized energy per read; the same is done for the write energy. This is repeated for every level in a memory hierarchy with M levels, with M indicating the DRAM level and with 0 the registers inside each PE of the array. The total energy is evaluated as:

$$Energy_{\text{memory}} = \sum_{i=1}^{M} memory_{read} \cdot energy_{\text{read}}^{\text{i-1 to i}} + \sum_{i=1}^{M} memory_{write} \cdot energy_{\text{write}}^{\text{i-1 to i}} \tag{2.26}$$

The overall computation latency expressed as the number of cycles can be evaluated by considering the total number of operations, as in Equation (2.8), the available data on-chip, and the number of concurrent operations that can be executed. The tiling factors determine inter-tile cycles, which are the number of data transfers required to compute the output of a single tile at the array level. The unrolling factors determine the intra-tile cycles, that is, the number of computation cycles required to compute the output of a single tile using all the data available at the array level. Using the notation of Table 2.2, the total number of operations is evaluated as the product of inter-tile and intra-tile cycles, as in Equation (2.27).

$$\begin{aligned}
Inter\text{-}tile\ cycles &= \left\lceil \frac{Nox}{Tox} \right\rceil \cdot \left\lceil \frac{Noy}{Toy} \right\rceil \cdot \left\lceil \frac{Nof}{Tof} \right\rceil \cdot \left\lceil \frac{Nif}{Tif} \right\rceil \\
Intra\text{-}tile\ cycles &= \left\lceil \frac{Tox}{Pox} \right\rceil \cdot \left\lceil \frac{Toy}{Poy} \right\rceil \cdot \left\lceil \frac{Tof}{Pof} \right\rceil \cdot \left\lceil \frac{Tif}{Pif} \right\rceil \cdot \left\lceil \frac{Nkx}{Pkx} \right\rceil \cdot \left\lceil \frac{Nky}{Pky} \right\rceil \\
Total\ operations &= Inter\text{-}tile\ cycles \cdot Intra\text{-}tile\ cycles
\end{aligned} \tag{2.27}$$

2.4.5 Formal Reuse Analysis

Highlighted in Section 2.4.1 and Section 2.4.2, reuse depends on the loop order, the memory size, and the allocated tiling factors. Considering the case of convolutional layers, if there are enough input activations and enough free storage for partial sums or output activations, the same weights can be reused to compute additional outputs. The same applies to weights and output activations. Reuse can be computed for all three data types at each memory level as the number of times a datum is accessed at the current level from the lowest level in the hierarchy until it is removed. To compute the reuse, weight, input activations, and partial sums or outputs are grouped into two data groups: I inputs (input activations and weights) and O outputs. I inputs are accessed with a single direction, as they are only read to compute the O outputs, which in turn can be either read from or written back to the memory. Using an energy model like the one of Section 2.4.4, it is possible to create the objective functions of Equation (2.28) to minimize the energy and latency of the accelerator by maximizing the reuse of the data. These objective functions are originally defined in [52] and are at the core of the loop scheduling optimization frameworks presented in Chapter 3. The coefficients defining the reuse at each level are directly taken from [52] and are: a for the reuse at the DRAM level, b for the on-chip memory level, c for the array level (neighbor-to-neighbor communication), and d for the RF level (internal memory of the PE). If there are multiple intermediate memory levels, additional coefficients can be added to represent the reuse at each level. If another on-chip buffer is added, a new coefficient, say e, can be added and multiplied with the reuse at each lower level. The energy cost computed with Equation (2.26) is used in Equation (2.28) to evaluate the energy cost associated with the reuse. Similarly, it is possible to estimate the overall communication and computation latency by using the latency cost. To simplify, as has been done in the previous section, only the equations for the energy are reported. Notice how O outputs can be accessed twice, once to read and once to write. The same coefficients used for I account for the read and write operations of the O outputs at each memory level.

$$\begin{aligned} I(a,b,c,d) &= a \cdot EN_{\text{DRAM}} + ab \cdot EN_{\text{on-chip}} + abc \cdot EN_{\text{array}} + abcd \cdot EN_{\text{RF}} \\ O(a,b,c,d) &= (2a-1) \cdot EN_{\text{DRAM}} + 2a(b-1) \cdot EN_{\text{on-chip}} + \\ &\quad 2ab(c-1) \cdot EN_{\text{array}} + 2abc(d-1) \cdot EN_{\text{RF}} \end{aligned} \tag{2.28}$$

It is possible to define the weight and output stationary dataflows presented previously with the constraints applied to the coefficients of Equation (2.28). For instance, in the case of an output stationary dataflow, the a coefficient is set to 1, so that output activations are only stored and not read from the DRAM, and b is set to 2, so that the output activations are stored from the array level to the on-chip memory, and then written back to the DRAM.

2.5 Error Resilience

AI algorithms are susceptible to three main computation error sources: numerical error due to compression, logic transients due to natural phenomena (EM interference), or adversarial attacks. Compressed DNNs are usually more susceptible to adversarial attacks and logic transients than full precision models if no particular training or post-processing are employed [7, 132, 133]. This section focuses on the error resilience to faults and adversarial attacks and presents some recent related works.

2.5.1 Hardware Errors

Unwanted and uncontrolled numerical errors, in addition to quantization and approximation, might cause catastrophic accuracy degradation, which is not tolerable in safety-critical applications. In autonomous driving, for instance, a DNN deployed to unmanned vehicles driven on regular roads [134, 135] may be used to detect pedestrians, cars, or trucks [25]. The decisions made by the guiding system, such as accelerating, braking, or turning, heavily rely on the accurate identification and localization of these objects. Numerical errors inducing a wrong output could result in unsafe conditions for the driver, pedestrians, or other objects near the vehicle. Therefore, applying error resilience techniques to deployed DNNs is imperative to ensure the reliability and safety of systems operating in real-world environments. As depicted in Figure 2.21, atmospheric neurons, ionizing particles, voltage/temperature variations, and other interference may perturb a transistor's state, generating bit flips in memory or current spikes in logic circuits that, if latched, lead to an error [7, 136]. The DRAM can also be targeted by row hammer attacks, which can change the content of any cell [137].

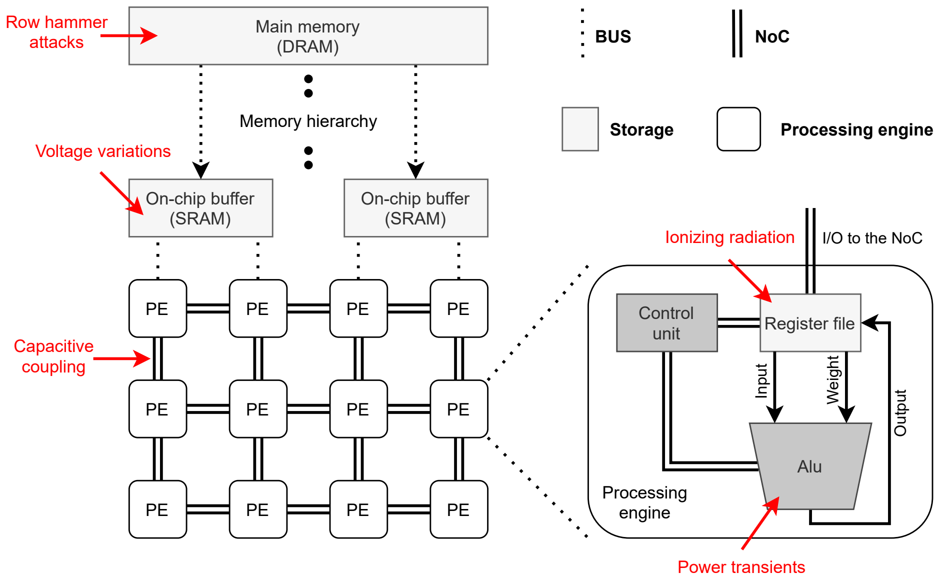


Fig. 2.21 Possible error sources and injection points in a generic hardware accelerator.

Recent works [133, 7] studied the effects of logic transients on the task accuracy of DNNs, emphasizing that it is necessary to take a holistic approach to error resilience, considering both the hardware and software aspects of the system. In particular, the authors of Fidelity [133] propose a framework that performs the simulation of logic transients in control and datapath units of a DNN accelerator after the mapping while simulating the computation. The mapping process extracts tokenized computation and communication patterns, which are used to trace the propagation of errors in the system and their influence on all subsequent computations. Contrary to previous works, which focused on bit-flips occurring only on memory elements, this study broadens the understanding of which parts of the system are more susceptible and if the best mitigation strategy has to be applied at the hardware or software level. The authors of [7] followed a similar injection approach to study the effects of bit-flips in activations and weights of quantized adversarially robust DNNs, observing that the activations are more error-prone than the weights and that old DNN architectures without batch normalization are more susceptible to logic transients. Additionally, quantization is antagonistic to error resilience and adversarial robustness, but this effect can be mitigated by adopting a strategy that tunes the scaling factors used to map from FP32 to integer, similarly to what has been proposed in [132]. These studies focused on randomly generated logic transients, but there are also targeted attacks such as row-hammer [137], which can be used to induce bit-flips in very specific locations of the DRAM. This opens a backdoor for attackers to search for targeted bit-flips that can annihilate the DNN's accuracy, as shown in [138], where 13 targeted bit-flips are induced to break a DNN with 93 million parameters. However, the possibility of a similar attack occurring in nature is extremely low. Therefore, the focus of the research work of Section 4.1 is on random bit-flips, which are more likely to occur in real-world scenarios.

2.5.2 Adversarial Attacks

Introduced in [139], adversarial attacks are a set of techniques that aim to reduce the accuracy of a DNN by adding a carefully generated noise to input data, as shown in Figure 2.22. What makes adversarial samples extremely important for safety concerns, if not even more than standard hardware errors, is that they can be embedded extremely well in the original data, making them indistinguishable to the human eye. The pixel noise added to the original image is usually very small and apparently random and is generated by minimizing the difference between the original and the adversarial sample while maximizing the difference between the original correct prediction and the target mis-prediction. A perturbation budget ε is used to limit the magnitude of the noise. Intuitively, a small ε will generate an almost imperceptible noise, whereas a large ε might add a visible distortion to the image.

Methods to generate adversarial samples $\hat{x}$ and train the adversarial generator usually leverage the gradient ∇ of the loss computed with respect to the input data x used to generate the noise L using the *sign* of the gradient (FSGM) [140], or a projection of the gradient (PGD) [141], reported in equations Equation (2.29) and Equation (2.30) respectively. The PGD attack is more effective than FGSM, as it is built on multiple iterations, but it is also more computationally expensive and slower to train and requires an additional hyperparameter α. In both Equation (2.29) and Equation (2.30), the perturbation δ is initialized and updated (learned) for multiple iterations.

$$\begin{aligned} \delta &= max(min(\delta + \varepsilon \cdot sign(\nabla L), \varepsilon), -\varepsilon) \\ \hat{x} &= x + \delta \end{aligned} \tag{2.29}$$

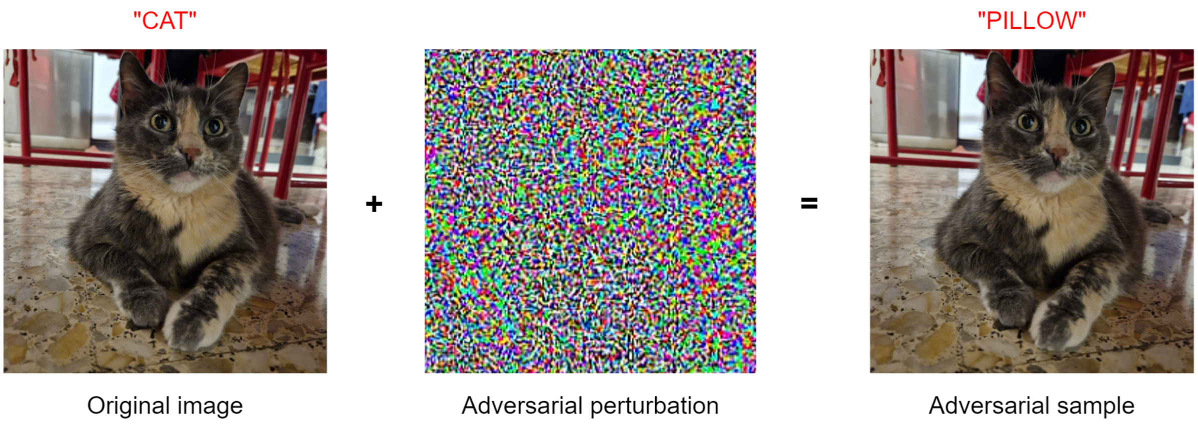


Fig. 2.22 In an adversarial attack, an RGB perturbation is generated and added to the original image, obtaining an adversarial sample that is indistinguishable from the original, inducing the DNN to predict a wrong class.

$$
\begin{aligned}
&(repeat\ \ \delta\ update\ for\ N\ iterations\ before\ adding\ \delta\ to\ x)\\
&\delta = max(min(\delta + \alpha \cdot sign(\nabla L), \varepsilon), -\varepsilon)\\
&\hat{x} = x + \delta
\end{aligned}
\tag{2.30}
$$

The DNN can be trained with adversarial samples to increase the robustness to adversarial attacks, as shown in Figure 2.23. This is called adversarial training and consists of embedding adversarial samples alongside standard samples in the training dataset. The perturbation generator can be based on FGSM, PGD, or any other attacking method. Of course, the computing overhead of adversarial training is significantly higher than the one of standard training; therefore, it is also necessary to consider the tradeoff between final task accuracy, adversarial robustness, and time to train and tune the hyperparameters [142].

Compressed DNNs are found to be more susceptible to adversarial attacks [7, 132]. Pixel noise in quantized DNNs pushes the activations (computed with the malicious pixels) to other quantization levels. These activations, in turn, do the same with other subsequent activations, and so on, in a chain effect in which the number and magnitude of wrong quantization values are amplified through the DNN [132]. The authors of [132] propose a method to bind the magnitude of weights and activations so that this amplification effect is turned into a dampening effect. Other works propose to limit the dynamic range of FP32 and INT values by applying clipping activations but were only applied to old DNN architectures without batch normalization and are of limited relevance [7].

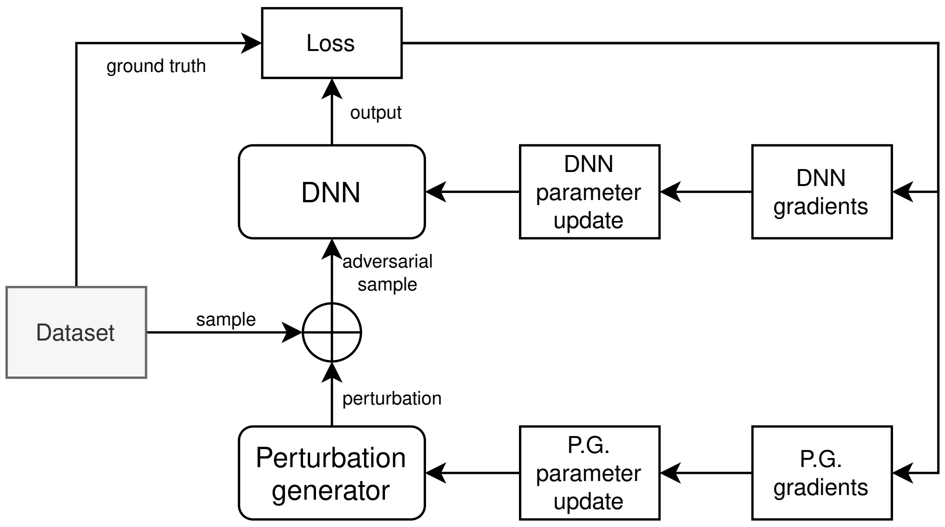


Fig. 2.23 A general adversarial training process consists of producing pixel noise and adding it to the training data, evaluating the loss, and updating the learnable parameters of both the DNN and the perturbation generator. Like regular training, the process is repeated for a fixed number of epochs, but the update of the perturbation generator can happen at a different pace than that of the DNN.

Chapter 3

Edge Inference Optimization with Compression and Scheduling

Deploying DNNs on embedded devices is challenging due to the high computational complexity and memory requirements of modern models. The development of specialized hardware accelerators has been proposed as a solution to improve the energy efficiency and throughput of DNN inference. However, the design of these accelerators is complex and requires exploring a large design space to find the optimal configuration for a given DNN model, maximizing selected performance targets. In the early design phases, the target hardware platform is not fully defined; the hardware is not available yet, and compilers are not optimized or create erroneous code. Therefore, a framework that allows the joint exploration of the design space of the DNN model and the hardware accelerator at different design phases is essential to optimize the performance of the final system.

This chapter presents a methodology for optimizing DNN models with compression and scheduling techniques to improve the performance of hardware accelerators. The proposed methodology is based on the Hw-Flow framework, which provides a multi-level hardware abstraction model for exploring the mapspace of compressed DNNs on accelerators. Three optimization strategies based on Hw-Flow are presented, focusing on the last: Hw-Flow-P, Hw-Flow-Q, and Hw-Flow-Fusion. Hw-Flow-P is a pruning methodology, whereas Hw-Flow-Q is focused on quantization. uning methodology, whereas Hw-Flow-Q is focused on quantization. Both methodologies compress NN models and directly impact the performance of hardware accelerators and that of the DNN. The name Hw-Flow-P is used only in this chapter to differentiate the compression strategy from the scheduling one, Hw-Flow, which is also the name of the paper [2]. Finally, Hw-Flow-Fusion is a layer fusion framework that uses scheduling techniques to leverage the data reuse between consecutive layers of a DNN to improve the performance metrics of hardware accelerators based on spatial arrays. Part of the work presented in this chapter has been previously published in [2–4].

3.1 Optimized Scheduling of Compressed Neural Networks

The goal of the Hw-Flow framework is to provide an exploration methodology and not a compression technique. All hardware-agnostic or hardware-aware quantization and pruning techniques discussed in Section 2.3 can be integrated in Hw-Flow.

3.1.1 Design Space Exploration with Multiple Abstraction Levels

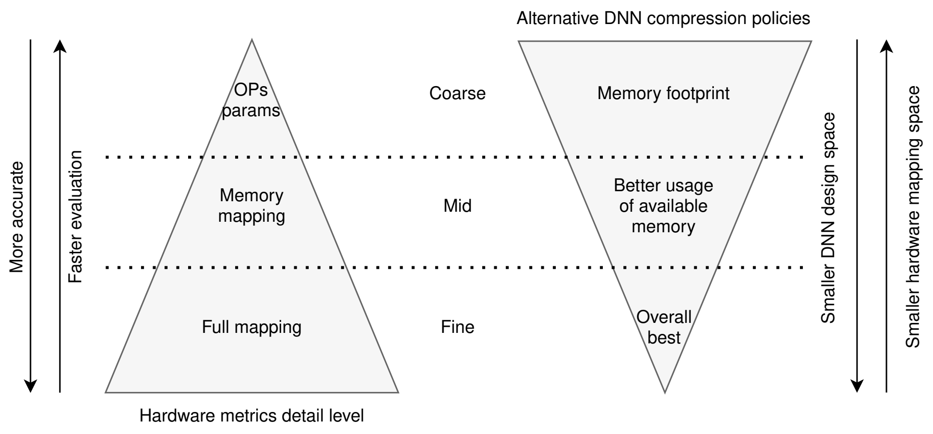


Fig. 3.1 Higher abstraction levels are explored faster than lower ones, but carry less information. The global optimum, i.e. the point with the best trade-off between accuracy and hardware metrics, is searched by iteratively refining the design space, removing solutions that are known (or deemed) to be inefficient. Redesign is done in case of constraints violation.

The idea behind multiple abstraction levels is to explore the design space in a top-down fashion, traversing the three levels of Figure 3.1: *coarse*, *mid*, and *fine*. These levels could represent different hardware development stages, from the early design phase, in which the architecture is not yet defined and only the model's size is a relevant metric, to the final implementation, in which the full mapping process is available, and performance bottlenecks are known. Each level carries more implementation-specific aspects and if the exploration fails to find any suitable solutions that satisfy the performance targets, the design is re-evaluated at the previous level. There are two search spaces to explore: the DNN design space and the hardware mapspace. While progressing towards the *fine* level, the DNN design space is reduced, as few variations of the same compressed model are considered. This means that some quantization and pruning policies are dropped as they are not among those with the highest efficiency for the current hardware mapping. Each abstraction level carries different constraints that must be met before moving to the next level. For instance, the *coarse* level could require a maximum number of operations, while the *mid* level could require a maximum off-chip communication volume. In case some constraints are not met, a redesign step is done. Previously unexplored policies are added to the pool of candidate solutions and the exploration is restarted from the last level that had valid solutions. Conversely, the hardware mapping space increases in complexity as more details are added to the mapping process while moving towards the *fine* level. The hardware feedback at the *coarse* level comprises the number of MACs and parameters and can be computed with a simple sum of products for each layer as in Equation (2.8). At the *mid* and *fine* levels, for each layer, it is necessary to evaluate each equation of Section 2.4 for each combination of tiling factors and relative loop orders to obtain the hardware feedback. Depicted in Figure 3.2, the exploration starts from the *coarse* level, where quantization and

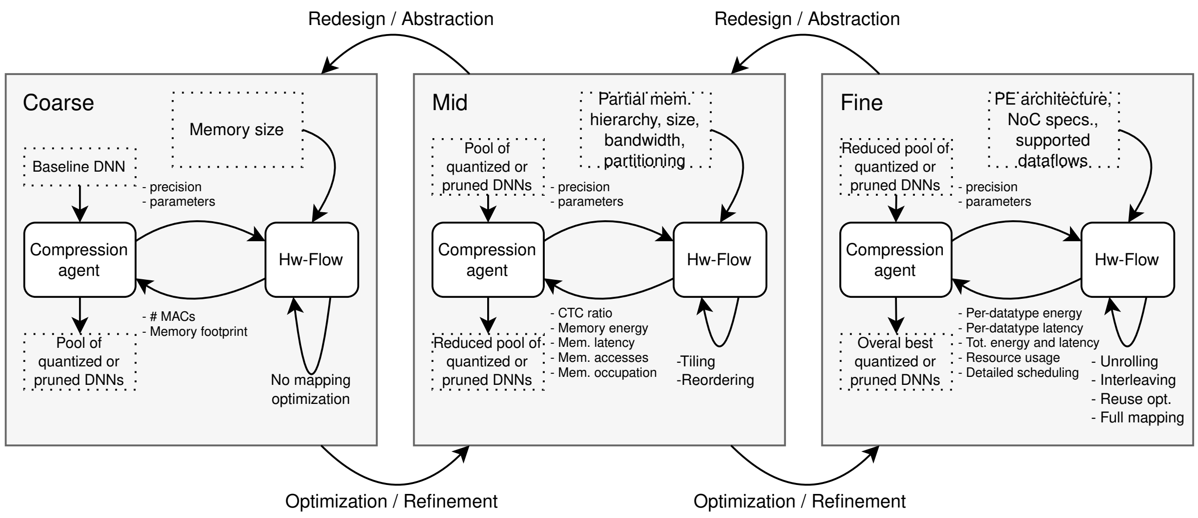


Fig. 3.2 Constraints, hardware details, mapping, and model compression are carried from higher to lower abstraction levels during each optimization step. In case of redesign, the mapping and compression policies up to the current level are discarded and new DNNs from the pool are optimized. The DNN pool contains different compression policies of the same baseline model.

pruning are applied to a single DNN architecture or different DNNs. This stage can be used to either generate a first pool of different compression policies applied to the same DNN or to select one DNN architecture from several models. At this stage, the total computation and memory footprint of the model are evaluated, whereas the task accuracy is provided by the ML framework executing the training loop and compression agent. No hardware mapping optimization is performed at this level, as the hardware is not yet defined. The generalized quantization or pruning schemes can be selected at this level, such as PACT [70] or geometric mean [89]. Once the DNN(s) meets the coarse-level constraints, the *mid* level is evaluated, assessing the performance of the memory hierarchy from the off-chip memory to the last buffer level before the array of processing units. At this stage, several combinations of tiling factors and loop orders are evaluated for each memory level for each layer of the DNN. Section 3.1.2 details how the performance metrics are evaluated and how the memory mapping is carried out. The *mid* level results can be used to identify possible performance bottlenecks in the memory hierarchy. The designer can then modify the hardware architecture, or enforce a more aggressive compression policy to reduce the data movement of some datatypes. When the *mid*-level constraints are met, the *fine* level, the last stage, is evaluated. The *fine* level mapping is based on the results of the *mid* level and is the most computationally expensive, as it requires the evaluation of the full hardware mapping, considering additional information as the number of processing units, the RF size, NoC specifications, and arithmetic units precision and vectorization. The hardware feedback can again be used to modify the array and PE architecture, for instance, by increasing the number of processing units to increase the throughput or decreasing the RF size due to low utilization. The compression policy can also be modified, for instance, by forcing a lower bitwidth of the weights to reduce the RF occupation.

3.1.2 Optimal Loop Scheduling

Section 2.4 details all the equations used in Hw-Flow to compute the optimal loop scheduling or hardware mapping. Depicted in Figure 3.3, the hardware mapping process is divided into two main steps: the memory mapper and the dataflow mapper. The memory mapper uses the hardware constraints related to the memory hierarchy, in blue in Figure 3.3, to find the optimal tiling factors and loop orders for each layer, adopting several performance metrics to evaluate the efficiency of the mapping. The dataflow mapper uses the results of the memory mapper to find the optimal tiling, unrolling, and interleaving for each layer, considering the dataflow constraints, in red in Figure 3.3. In case of invalid dataflow mapping, the memory mapper is re-executed. At each level, the memory and dataflow mapper uses Equation (2.16) to compute the memory requirements and decide whether the tiling factors are valid or not, if they lead to a memory occupation that exceeds the available memory size. The loop order can be selected between one of the three possible orders detailed in Section 2.4.2: *IR*, *WR*, and *OWR*. The fetch and write operations are computed using Equations (2.17, 2.18, 2.19), while the CTC is computed with Equation (2.24), and the inter-tile cycles with Equation (2.27). The overall communication energy is computed by accumulating, for each memory level, the total fetch/write energy evaluated with Equation (2.26), using the communication volume evaluated with Equation (2.20).

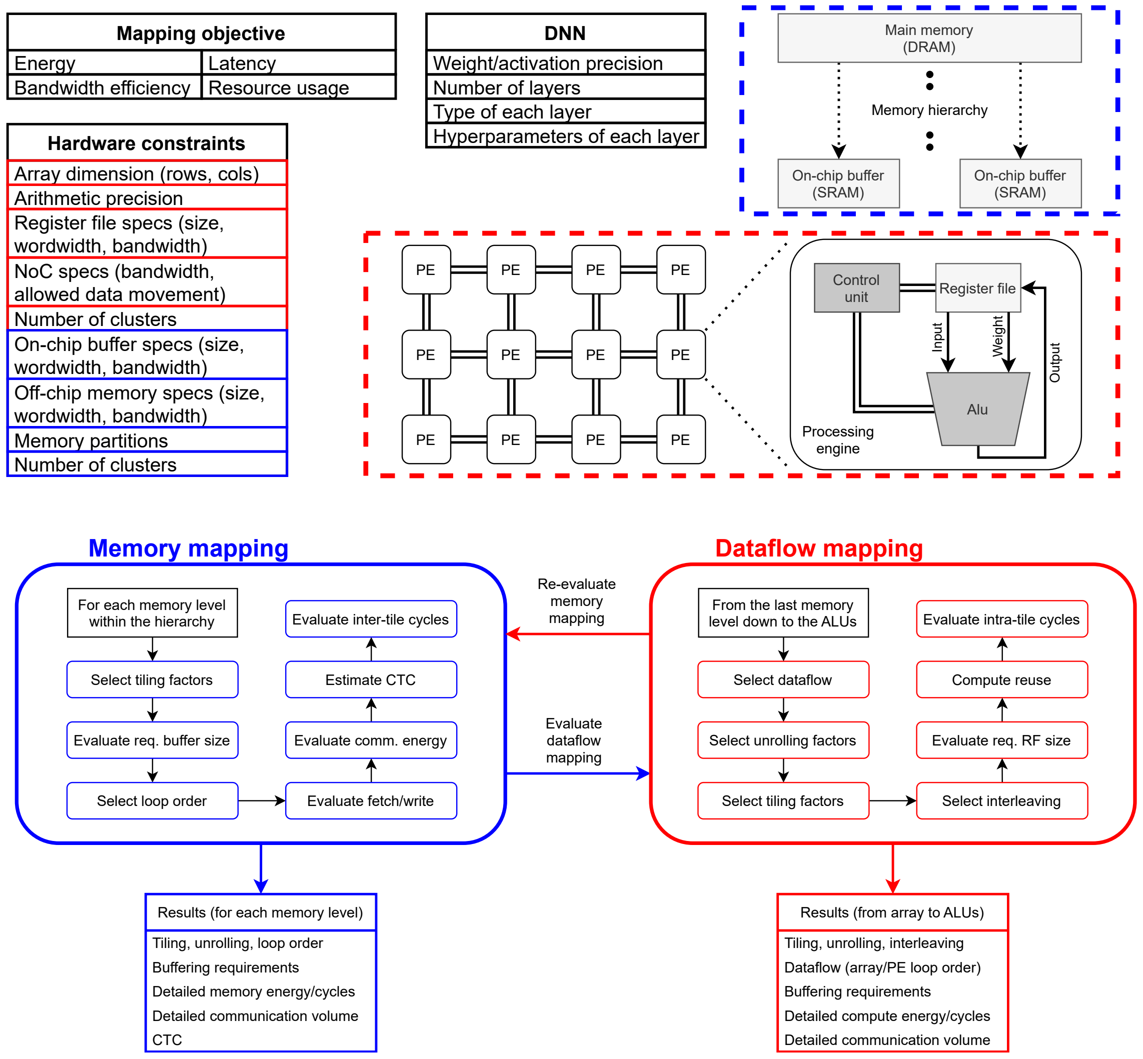

Mapping objective
Energy
Latency
Bandwidth efficiency
Resource usage
DNN
Weight/activation precision
Number of layers
Type of each layer
Hyperparameters of each layer
Main memory (DRAM)
Memory hierarchy
On-chip buffer (SRAM)
On-chip buffer (SRAM)
Hardware constraints
Array dimension (rows, cols)
Arithmetic precision
Register file specs (size, wordwidth, bandwidth)
NoC specs (bandwidth, allowed data movement)
Number of clusters
On-chip buffer specs (size, wordwidth, bandwidth)
Off-chip memory specs (size, wordwidth, bandwidth)
Memory partitions
Number of clusters
PE
Control unit
Register file
Input
Weight
Output
Alu
Processing engine
Memory mapping
For each memory level within the hierarchy
Select tiling factors
Evaluate req. buffer size
Select loop order
Evaluate fetch/write
Evaluate comm. energy
Estimate CTC
Evaluate inter-tile cycles
Re-evaluate memory mapping
Evaluate dataflow mapping
Dataflow mapping
From the last memory level down to the ALUs
Select dataflow
Select unrolling factors
Select tiling factors
Select interleaving
Evaluate req. RF size
Compute reuse
Evaluate intra-tile cycles
Results (for each memory level)
Tiling, unrolling, loop order
Buffering requirements
Detailed memory energy/cycles
Detailed communication volume
CTC
Results (from array to ALUs)
Tiling, unrolling, interleaving
Dataflow (array/PE loop order)
Buffering requirements
Detailed compute energy/cycles
Detailed communication volume

Fig. 3.3 Joint memory and dataflow mapping process. Blue and red colors highlight the constraints and hardware model used by the memory mapper or the dataflow mapper.

All these metrics are used to create an ordered list of the valid memory mappings according to the mapping objective set by the user. The best memory mappings are passed to the dataflow mapper, which builds and explores the low-level mapspace, which includes the set of all possible dataflows, tiling, unrolling, and interleaving factors for a given layer. Equation (2.16) is again used to evaluate the memory requirements for the buffering at the register file level, while variations of Equation (2.17), Equation (2.19), and Equation (2.18) are used to evaluate the fetch and write operations from the last on-chip memory level to the array level. The dataflow mappers minimize the data movement by optimizing the objective function of Equation (2.28).

As the reuse coefficients of Equation (2.28) directly depend on the selected loop order, dataflow, tiling, unrolling, and interleaving factors, the dataflow mapper can assess the efficiency of the entire hardware mapping. After that, intra-tile cycles and total computation cycles are evaluated with Equation (2.27). Finally, a feedback from the dataflow mapper is sent to the memory mapper, which re-executes the mapping process with different parameters. The cycle repeats until all valid mappings have been evaluated.

3.1.3 Pruning and Quantization Agents

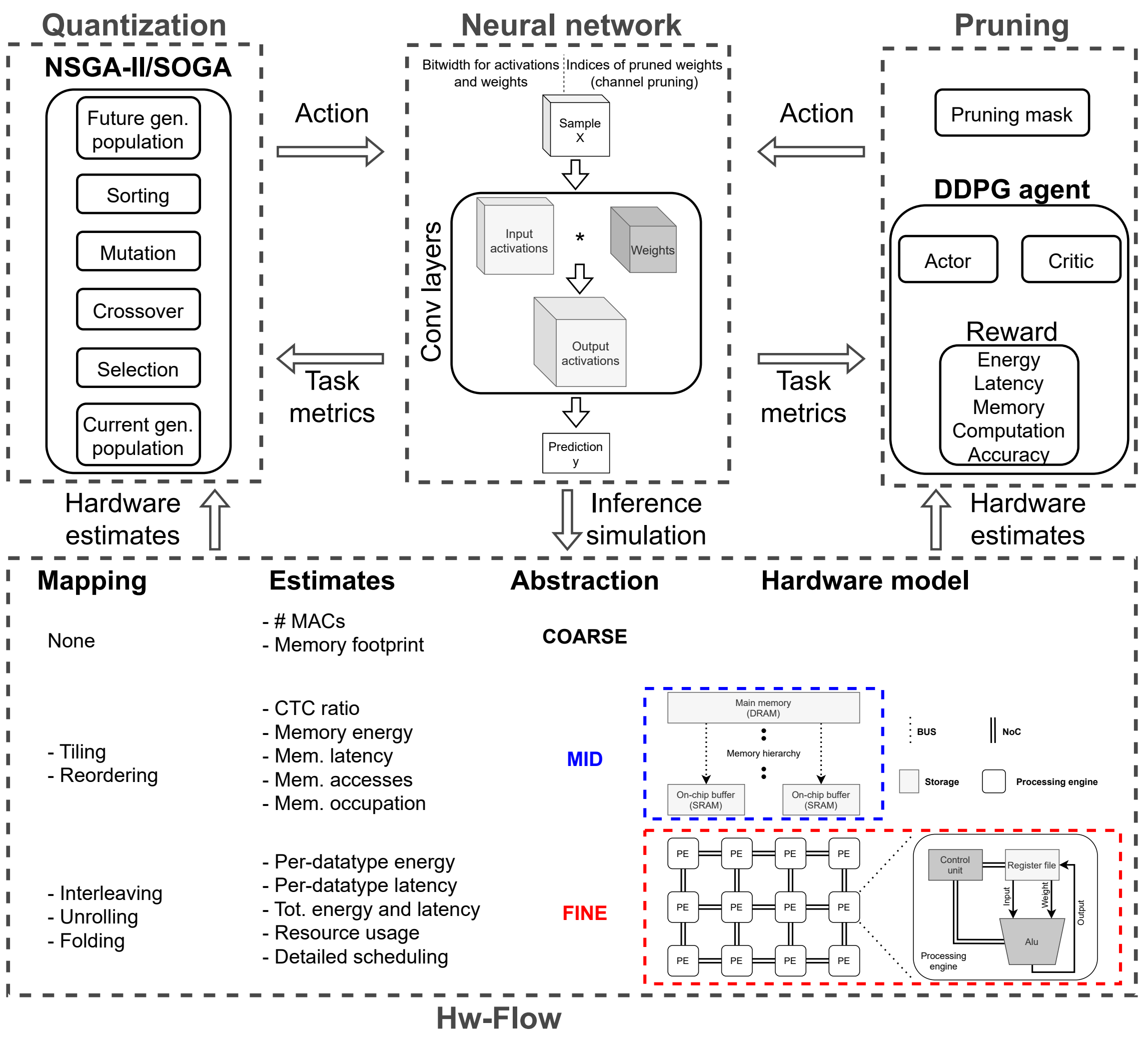


Fig. 3.4 Overview of the quantization and pruning methodologies. The hardware estimates are used to compute the reward for the RL agent used for pruning and the objective functions for the genetic algorithm used for quantization.

The mapping process of Section 3.1.2 is used to evaluate the efficiency of the compression policies generated by the pruning agent of Hw-Flow-P [2] and the quantization agent of Hw-Flow-Q [3]. The pruning agent is based on a RL algorithm based on a deep deterministic policy gradient (DDPG) agent [27], while the quantization agent is based on two genetic algorithms, a single objective genetic algorithm (SOGA), and a multi-objective nondominated sorting genetic algorithm (NSGA-II). The quantization and pruning processes of Figure 3.4 are not performed in parallel but are depicted together as they share both the DNN training environment and the hardware mapping framework.

Pruning Agent The pruning agent of Hw-Flow-P [2], depicted in Figure 3.4, is based on the work of [80]. The agent receives a state S^l for layer l and generates a pruning mask A^l as action. A reward R is computed based on the accuracy of the model and the hardware estimates and used to train the agent. The computation of reward and the state information change according to the abstraction level, allowing the agent to learn the effects on the performance of the pruning policy that it generates. The state S of Equation (3.1) comprises the layer index l, stride s, the dimension of the pruned layer $N\tilde{o}f, N\tilde{i}f, Nix, Niy$. Hardware estimates are included as $\varphi = [\varphi^0, ..., \varphi^l, ..., \varphi^L]$ for each layer l of a DNN with L layers, each φ^l containing the chosen performance metric (i.e., energy, latency, CTC). The term A^{l-1} is the action performed at the previous step, i.e., layer.

$$S^l = \{l, s, N\tilde{o}f, N\tilde{i}f, Nix, Niy, \varphi^l, \sum_{i=0}^{l-1} \varphi^i, \sum_{j=l+1}^{L} \varphi^j, A^{l-1}\} \tag{3.1}$$

Two reward protocols are adopted: the *balanced* and the *constrained* reward functions, as defined in Equation (3.2).

$$\mathscr{R} = \begin{cases} (1 - \frac{\psi^* - \psi}{b}) \cdot \log(\frac{\varphi^*}{\varphi}), & \text{if } \textit{balanced} \\ \psi, & \text{otherwise } \textit{constrained} \end{cases} \tag{3.2}$$

The reward function is formulated to achieve a minimum target accuracy ψ^* before performance estimates are optimized. This condition occurs when hardware constraints are not known. The measured accuracy ψ is compared to the target accuracy ψ^*, and the agent is encouraged to improve the accuracy when the difference is larger than b. The term b also influences the sign of the reward. The ultimate goal of the *balanced* reward, for each pruning action, is to trade-off the accuracy term $(1 - (\psi^* - \psi)/b)$ and the hardware estimate term $\log(\varphi^*/\varphi)$. The terms φ^* and φ indicate the estimates of the baseline and pruned model, respectively. Adding additional logarithmic terms makes it possible to include additional performance metrics to the reward function. On the other hand, the *constrained* reward promotes maintaining higher prediction accuracy ψ after each pruning action. This encourages the agent to prune the model while minimizing the accuracy degradation with specified hardware constraints. The pruning action is performed for as many steps as the number of layers composing the DNN, which is then retrained for a fraction of an epoch to recover from the accuracy loss introduced by the pruning and tested to extract the task accuracy. The other relevant implementation details and the results can be found in [2].

Quantization Agent The search space of mixed-precision and layerwise quantization strategies q^{2L} solutions, where q is the set of possible quantization levels and L is the number of layers. To explore this enormous search space, two genetic algorithms are used: a single-objective genetic algorithm (SOGA) and a multi-objective genetic algorithm (MOGA), the latter based on NSGA-II [143]. The SOGA is used when a single objective function is defined (only task accuracy), while the MOGA is used when multiple objectives are defined (task accuracy and hardware metrics), using the fitness defined in Equation (3.3). The terms φ^* and φ indicate the estimates of the baseline and quantized model, respectively. Similarly, ψ^* and ψ are the task-related accuracy of the baseline and quantized model, respectively. The previous conditions correspond to the *coarse* and *mid, fine* levels of the abstraction hierarchy, respectively. Each individual of the population is a quantization strategy, and the fitness of the individual is evaluated based on the task accuracy and the hardware estimates (only for MOGA). The quantization strategy is encoded in a genome where each genetic locus contains the bitwidth values for weights and activations at the corresponding layer. The alleles of the genetic loci are the set of possible quantization levels supported by the hardware model, therefore anything in the range $[1, q]$. Regarding the genetic operations of mutation and crossover of Figure 3.4, the mutation operator replaces a single allele at a randomly selected genetic locus, whereas the crossover operator swaps the bitwidth-to-layer encoding of two fit individuals. Single-point crossover is used to preserve inter-layer dependencies across segments of the DNN. There is a higher correlation between the parameters of adjacent layers than between distant ones. Assuming that only the fittest individuals survive, sequences of quantization levels that benefit task accuracy and hardware efficiency are more likely to be preserved and reused in the offspring. Therefore, single-point crossover is a good choice to maintain the locality of the genetic loci. Mutation then allows the offspring to overcome the local minima of their parents. The selection process is based on tournament selection for SOGA and on the crowded-comparison operator for NSGA-II. The sorting operator of Figure 3.4 refers to the nondominated Pareto sorting of NSGA-II and is used only in the MOGA. At each iteration of any genetic agent, the population is evaluated after a quick fine-tuning, and the best individuals are selected to generate offspring. Once the population of the next generation has been selected, the process is repeated for *n* generations. The other relevant implementation details and the results can be found in [3].

$$\mathscr{F}_\rho = \begin{cases} (1 - \frac{\psi^* - \psi}{t}) \cdot \log(\frac{\varphi^*}{\varphi}), & \text{if SOGA} \\ \psi, \varphi & \text{otherwise NSGA-II} \end{cases} \tag{3.3}$$

3.1.4 Hw-Flow Framework Validation

The main contribution of the research work carried out by the doctoral candidate in not on the optimization agents, but on the scheduling and mapping framework. Therefore, only the experimental results related to the hardware mapping are presented in this section.

The baseline Eyeriss model architecture from [52] is used to validate the Hw-Flow framework, comparing the mapping results against those generated with Timeloop [107]. As Eyeriss exposes several data movement patterns within the memory hierarchy and complex on-chip data movement, it is a suitable target for the validation of the Hw-Flow framework. This includes identifying and avoiding illegal mappings, such as those that violate the memory constraints or allowed data-movement, and evaluating correct energy and latency estimates. Figure 3.5 shows the normalized energy and latency estimates of the Hw-Flow framework compared to Timeloop. The energy estimates are consistent with the baseline and Timeloop, showing only a small discrepancy, while the latency estimates are exactly the same.

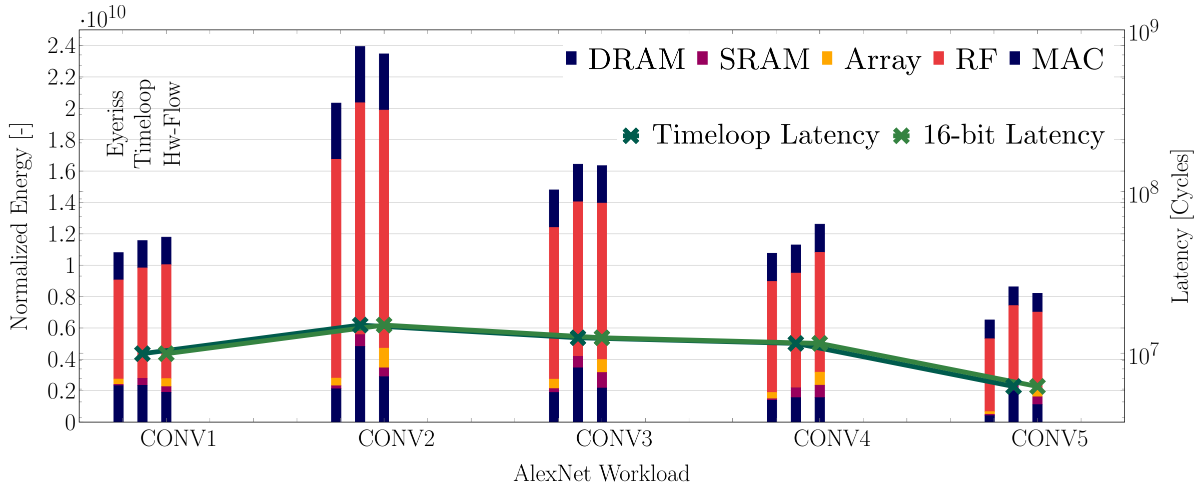


Fig. 3.5 Energy and latency validation with AlexNet and the baseline Eyeriss architecture.

To further validate the Hw-Flow framework, the Eyeriss architecture [52] based on the variation with 256 PEs and 256KB of on-chip memory, as detailed in [107] is chosen and simulated using DNN workloads taken from the DeepBench benchmark [144], to validate the framework's ability to handle layers with different geometries. The mapping results and hardware estimates are again compared with those generated with Timeloop [107], used as the baseline/ground-truth. Figure 3.6 and Figure 3.7 show the normalized energy and latency estimates of the Hw-Flow framework compared to Timeloop. The results show that the Hw-Flow framework is able to reliably estimate the energy and latency of the Eyeriss architecture, compared to Timeloop [107].

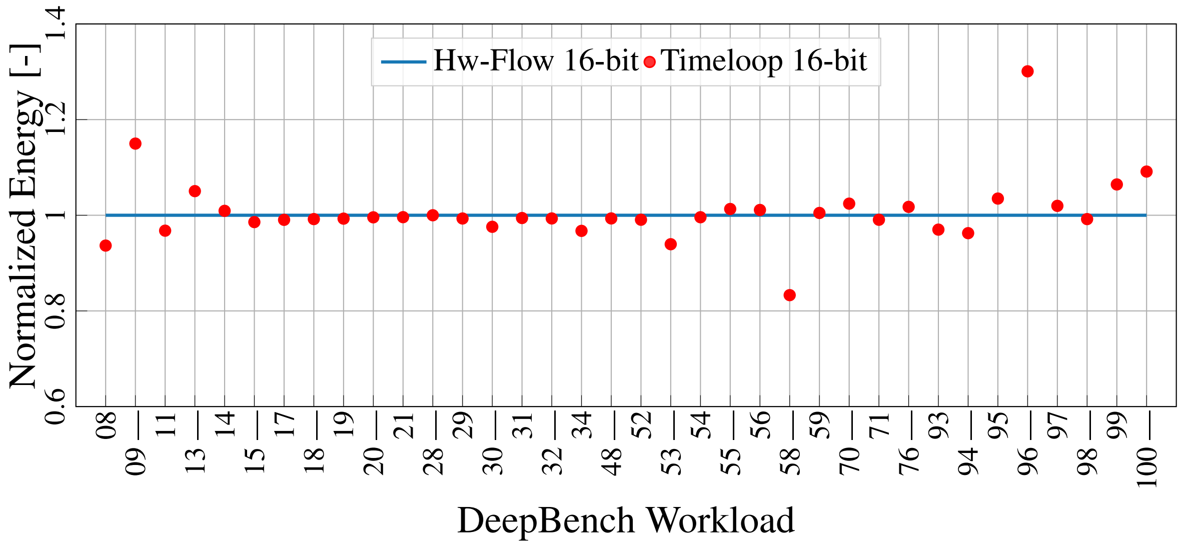


Fig. 3.6 Validation of the Hw-Flow framework for energy estimates against Timeloop.

3.2 Hw-Flow-Fusion

Hw-Flow-Fusion is the last iteration of the Hw-Flow project and is focused on inter-layer scheduling, in which the execution of multiple layers of a DNN is mapped to an accelerator in which the resources are allocated to process them concurrently, during the same inference. Accelerator based on a spatial array have enough hardware settings and control to support multiple PE sets to compute the same layer, as demonstrated in [52, 125], and can be enhanced with an inter-layer scheduling framework that can explore the mapspace of fused layers. Instead of using dedicated arrays, Hw-Flow-Fusion leverages the PE sets to compute different layers, reusing the available hardware resources.

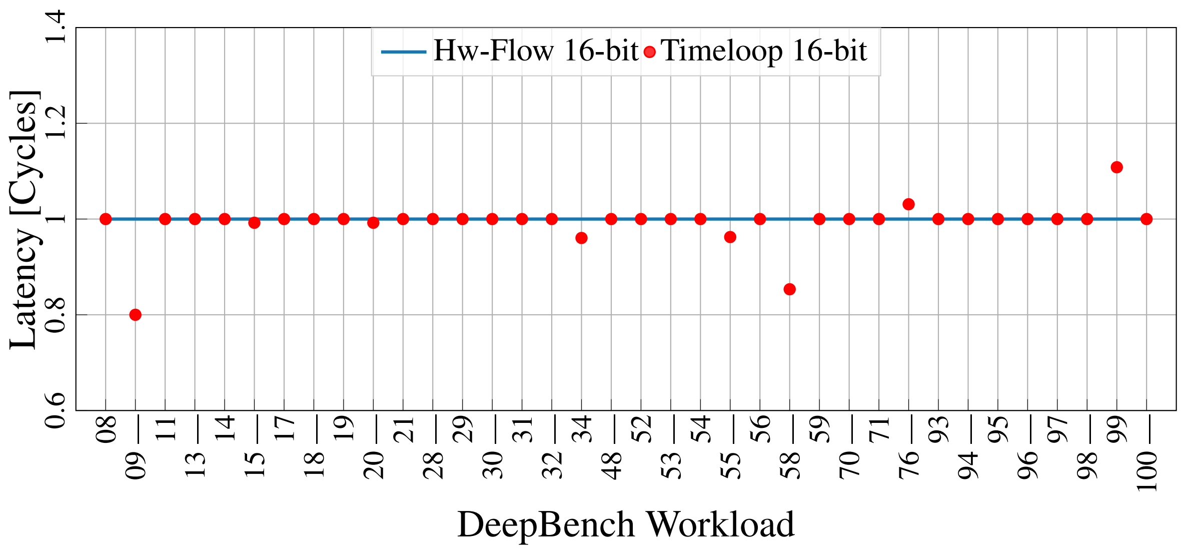


Fig. 3.7 Validation of the Hw-Flow framework for latency estimates against Timeloop.

3.2.1 Inter-Layer Scheduling in Previous Works

In MAGMA [130], multiple layers with no data dependency are scheduled on the same hardware accelerator. The resource partitioning is done at the array level and determined using a genetic algorithm. Improving over [130], Hw-Flow-Fusion [4] also considers the memory partitioning within each processing engine of the array. In DNNFusion [145], the execution of DNNs on CPUs and GPUs is accelerated by re-writing the computational graph during the mapping, fusing the execution of arithmetical operations. Contrary to [4, 130, 146, 147], which focus on reconfigurable dataflow architectures, DNNFusion targets general-purpose architectures that leverage the GEMM algorithm [58] to compute the convolutional layers. The concept of executing multiple layers with data dependencies was introduced in [146], where is proposed an accelerator with a systolic array that can execute the first five layers of VGG16-E, reducing the volume of data transferred to the DRAM by 95%. A dedicated on-chip memory is used to store intermediate data to avoid recomputing redundant pixels. This methodology can be applied only to accelerators customized for a specific DNN, preventing the same hardware accelerator from being used for a different DNN model. This limitation is solved in Hw-Flow-Fusion[4] and [147, 148], in which inter-layer scheduling is extended to dataflow architectures. In DNNfuser [147], transformers are used to fuse and map on a dataflow accelerator the execution of multiple layers that share the same hardware resources. However, only a limited number of spatial and temporal tiling factors are considered during the search, and the reuse strategy, fundamental in [146] to avoid redundant data movement and arithmetic operations, is not discussed. Additionally, there are no details on the memory used to store intermediate pixels and how the computation and communication patterns between fused layers are modeled. In [148] is presented a framework to map the execution of multiple layers of the same DNN, using a genetic algorithm to find the energy/latency optimal resource allocation on a heterogeneous multi-core hardware architecture. The memory system model includes on-chip and core-to-core communication cost and latency, with the introduction of a custom memory manager that minimizes congestion and maximizes bandwidth usage.

3.2.2 Proposed Methodology

Inter-Layer Scheduling Concepts and Constraints

Partial sums, generated as the products of input pixels and weights, are usually accumulated on-chip, then stored in the off-chip memory once the computation of the current layer is finished, and fetched again when processing the next layer [1, 50–52]. This process repeats until the last layer of the network. Output pixels from a layer, except the last one, can be called intermediate pixels, as they are not the original input to the network nor the final output. It could be possible to fetch intermediate pixels directly from the processing engines or the on-chip memory, reducing the data movement along the memory hierarchy and generating the output of the next layer. Intermediate pixels would be directly passed from one layer to the next, retained in case of data dependency, or discarded when no longer needed. The computation of DNNs in a specialized accelerator is deterministic; therefore, it is possible to evaluate when to save or delete any activation (input, intermediate) or set of weights on a dedicated on-chip buffer that, similarly to [146] has the function of retaining data that is required in multiple processing steps. Intermediate data exists only on the on-chip memory and lower memory levels, allowing additional energy saving by reducing the most expensive data movement, as the cost for DRAM accesses can be 20~30 times higher than that of SRAM accesses [52, 107]. Moreover, by reducing access to the DRAM, it is also possible to reduce the communication latency and the impact of bandwidth bottlenecks on the computation [131, 148]. Figure 3.8 depict the normalized communication volume of the data processed in each layer of ResNet-18 for ImageNet [13] and highlights that pixel volumes dominate the first half of the DNN, whereas weight volume dominates the second one. As inter-layer reuse can be applied only to intermediate pixels and requires buffering of intermediate weights, inter-layer scheduling can increase energy and bandwidth efficiency in some parts of this DNN model unless other compression techniques such as quantization [3] or pruning [2] are applied to reduce the memory footprint of weights. In this case, and this work, without compression, only the execution of the first half of the DNN could be optimized by fusing layers. In contrast, the second half would be scheduled using traditional single-layer techniques since inter-layer reuse would be negligible and require an unacceptable amount of storage to save the weights of intermediate layers. Moreover, the order of intermediate pixel generation is essential to keep coherence in the computation without increasing the off-chip communication; therefore, using different temporal loop reordering in inter-layer scheduling is not possible. As pointed out in Section 2.4.1, the temporal loop order determines the communication and computation pattern, meaning that any change would determine a different order of pixel production and consumption for each layer, breaking the data dependencies.

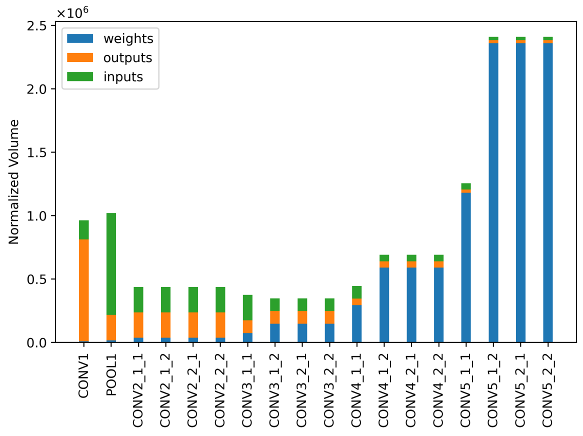


Fig. 3.8 Normalized data volumes for ResNet-18.

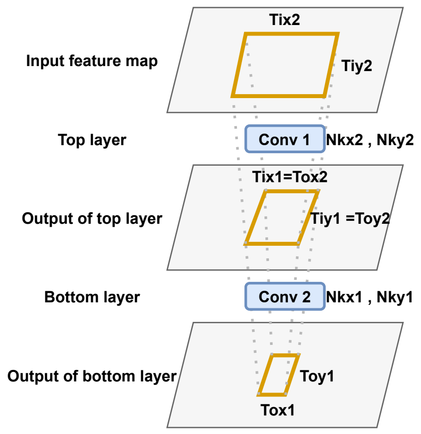


Fig. 3.9 Tiling multiple layers.

The inter-layer scheduling process starts from a set of output pixels from the last layer of the fused sequence, defined as the *bottom layer*, and traces back the computation to the first input layer, defined as the *top layer*. Depicted in Figure 3.9, the output tile identifying the output pixels is propagated across all the layers between the *top* and *bottom* ones, with its feature size increasing each time a layer is traversed. Equation (3.4) defines the feature size transformations between two adjacent layers, m at the top and $m-1$ at the bottom, recalling that the input of layer $m-1$ is the output of layer m, using the notation of Table 2.2. Only the equations for x are reported for conciseness, as those for y are identical. Horizontal (vertical) stride is denoted with Sx (Sy).

$$\begin{aligned} Tof^{\mathrm{m}} &= Tif^{\mathrm{m\text{-}1}} \\ Tox^{\mathrm{m}} &= Tix^{\mathrm{m\text{-}1}} = (Tox^{\mathrm{m\text{-}1}}-1)\cdot Sx^{\mathrm{m\text{-}1}} + Nkx^{\mathrm{m\text{-}1}} \\ Tix^{\mathrm{m}} &= (Tox^{\mathrm{m}}-1)\cdot Sx^{\mathrm{m}} + Nkx^{\mathrm{m}} \end{aligned} \tag{3.4}$$

Intermediate buffers store all the intermediate pixels evaluated between consecutive layers, meaning the output of the preceding layer, which is also the input of the subsequent one. The processing of any layer scheduled in such a way can start only after enough output pixels have been evaluated. It is important to highlight that the presence of batch normalization layers [42] does not prevent the application of inter-layer scheduling. Batch normalization, outside training, can be computed with a portion of the output activations, just like convolutional layers. Additionally, the normalization can be folded into the convolutional layers using the technique proposed in [102]. Consequently, intermediate buffers can enable the computation activations or normalization of layers between consecutive convolutional layers without additional storage. Therefore, ignoring shared buffers and assuming m = M as the top layer and m = 0 as the bottom one, the required buffer size is evaluated in Equation (3.5). The approach followed in this work is to create a batch-independent fusion schedule like the one proposed in [146], setting the tiling factors (notation in Table 2.2) *Tif* = *Nif* and *Tof* = *Nof* for all the intermediate layers. The constraint on the channel dimensions increases the buffering requirements of Equation (3.5) significantly, compared to Equation (2.16), as it forces to save the entire weight volume on-chip, but eliminates two loops from Algorithm 1 for each layer. Consequently, it is also possible to maximize weight reuse for any batch size because the buffered weights are reused during the computation of every *Tox* ·*Toy* output pixels (Equation (2.18)).

$$Fusion\ buffer = \sum_{m=0}^{M} I^{\text{m}}_{\text{buffer}} + \sum_{m=0}^{M} W^{\text{m}}_{\text{buffer}} + O^{\text{bottom}}_{\text{buffer}} \tag{3.5}$$

The fetch and write invocations of Equation (2.18) can be rewritten for inter-layer scheduling, recalling that intermediate activations generate no external communication volume. The final communication volume is evaluated as in Equation (3.6); notice how weights are accessed precisely once, except for the bottom layer.

$$Total\ volume = I^{\text{top}}_{\text{buffer}} \cdot Ifetch + \sum_{m=0}^{M} W^{\text{m}}_{\text{buffer}} + O^{\text{bottom}}_{\text{buffer}} \cdot Owrite \tag{3.6}$$

As pointed out in [146], intermediate pixels of adjacent tiles are computed multiple times if not properly buffered. This redundancy is also found in single-layer scheduling [60]. Figure 3.10 depicts the tile overlapping of a feature map during a convolution. The overlapping depends on the stride and the spatial dimensions of the tile, feature map, and kernel map. In [146], it is demonstrated that recomputation can account for up to 10 times the energy estimated with single-layer scheduling for the entire DNN; therefore, overlapping pixels must be stored in appropriate reuse buffers. The overlapping regions *Ox* and *Oy*, depicted in Figure 3.10, can be evaluated as shown in Equation (3.7). The next section presents a solution to the re-computing problem, improving the one proposed in [146], from the observation that some overlapping regions are immediately reused during the computation.

$$Ox = Nkx - Sx \qquad Oy = Nky - Sy \tag{3.7}$$

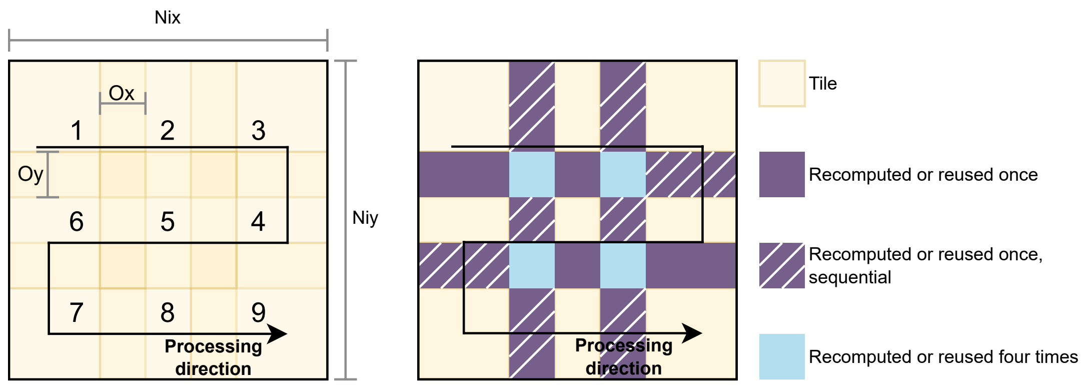

Nix
Ox
1
2
3
Oy
6
5
4
7
8
9
Processing direction
Niy
Processing direction
Tile
Recomputed or reused once
Recomputed or reused once, sequential
Recomputed or reused four times

Fig. 3.10 Overlapping regions of adjacent tiles happen during regular processing. In case of single-layer scheduling, these regions would be re-fetched from the main memory.

Optimized Intermediate Pixel Reuse Model

The overlapping regions of Figure 3.10 delimit the tile portions that must be stored and reused to avoid recomputation. With an appropriate tiling, it is possible to define the exact control sequence of a memory controller in charge of managing the correct movement of intermediate results between the processing units and the on-chip memory, similarly to what has been proposed in [148] to optimize the redundant data sharing. Only one set of the regions of Figure 3.11 has to be stored and reused and must be accounted for when evaluating the size of the reuse buffer. Equation (3.8) defines the required reuse storage, the term *reuse buffer*$_x$ is used when the direction of tile processing is horizontal, as in Figure 3.10, and covers the contributions of (1,4), (1,2,4,5), (2,5), and (2,3,5,6), whereas *reuse buffer*$_y$ is used when the processing direction is vertical.

$$\begin{aligned} reuse\ buffer_x &= (Nix - Tix) \cdot (Nky - Sy) \cdot Nif \\ reuse\ buffer_y &= (Niy - Tiy) \cdot (Nkx - Sx) \cdot Nif \end{aligned} \tag{3.8}$$

The reuse model of [146] additionally saves the sequential overlap, requiring the additional memory *reuse overhead*$_y$ and *reuse overhead*$_x$ of Equation (3.9), added to *reuse buffer*$_x$ and *reuse buffer*$_y$ of Equation (3.8) for the horizontal and vertical processing.

$$\begin{aligned} reuse\ overhead_x &= (Tix - (Nkx - Sx)) \cdot (Nky - Sy) \cdot Nif \\ reuse\ overhead_y &= (Tiy - (Nky - Sy)) \cdot (Nkx - Sx) \cdot Nif \end{aligned} \tag{3.9}$$

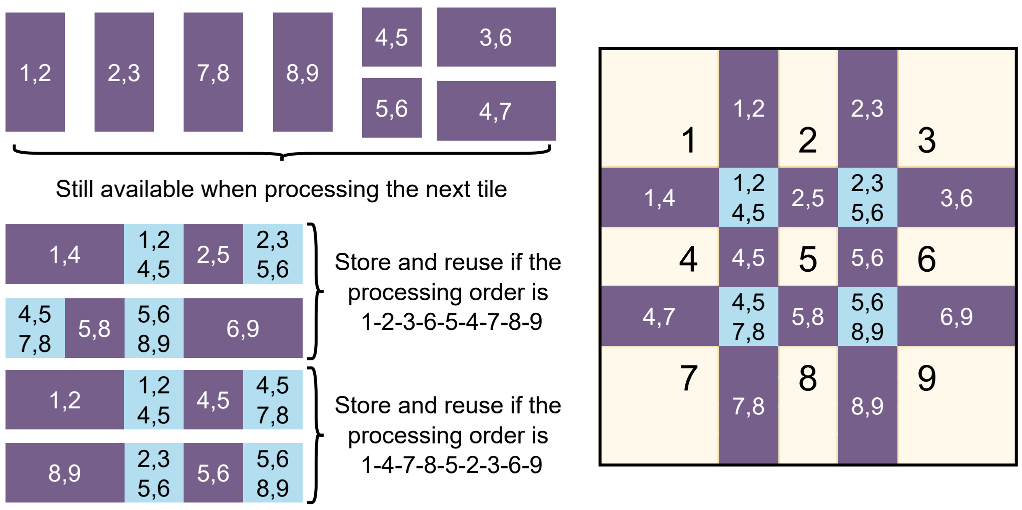
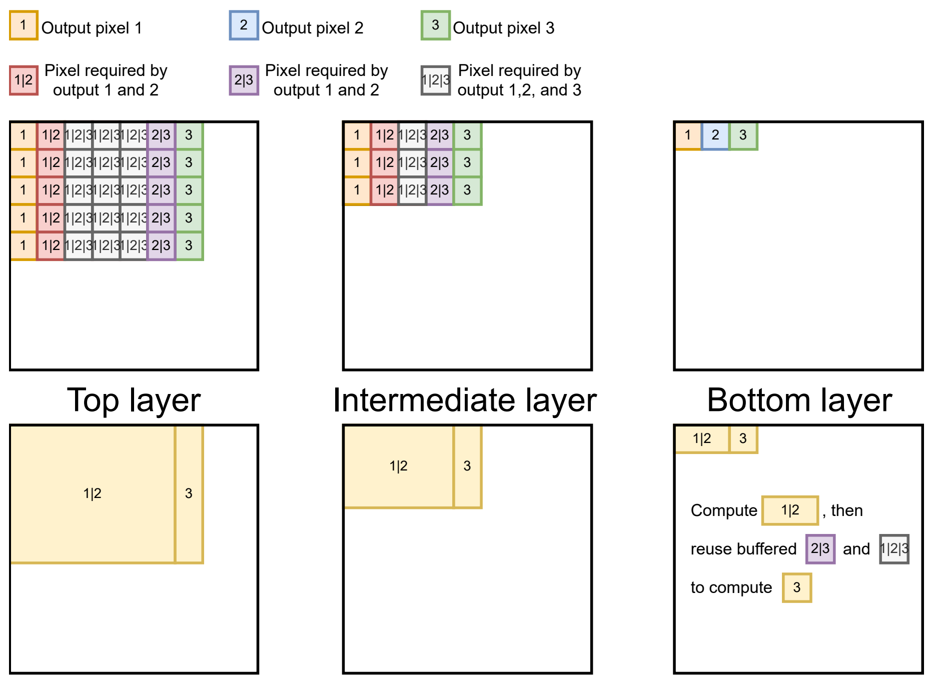

1,2
2,3
7,8
8,9
4,5
5,6
3,6
4,7
Still available when processing the next tile
1,4
1,2 4,5
2,5
2,3 5,6
4,5 7,8
5,8
5,6 8,9
6,9
Store and reuse if the processing order is 1-2-3-6-5-4-7-8-9
1,2
1,2 4,5
4,5
4,5 7,8
8,9
2,3 5,6
5,6
5,6 8,9
Store and reuse if the processing order is 1-4-7-8-5-2-3-6-9
1
2
3
4
5
6
7
8
9

Fig. 3.11 Overlapping tiles that require buffering. Only one block out of the four marked for reuse has to be stored to achieve zero recomputation scheduling.

Multi-Tiling for Minimum Reuse Memory with No Recomputation

Multi-tiling is used to schedule the computation of unique pixels without generating redundant results and, paired with reuse buffers, enables inter-layer scheduling with zero recomputation. Multi-tiling starts as a normal tiling process by generating a set of output tiles, *Tox Toy*, and then evaluating the corresponding input tiles, *Tix* and *Tiy*. The procedure is depicted in Figure 3.12: the output pixels of the top layer are colored based on the data dependencies of the intermediate layer, which requires them to produce other intermediate results used to compute the output pixels 1, 2, and 3 of the bottom layer. The computation is traced back using the input/output relations of Equation (3.4), assuming $Nkx = Nky = 3$, $Sx = Sy = 1$. In this example, the first tile to be evaluated is the one marked as 1|2 in the bottom right square. New pixels marked with 1, 2, 1|2, 2|3, and 1|2|3 are computed. From Equation (3.7) it is possible to evaluate the overlapping in the top and intermediate layer and store all the pixels that will be reused during the computation for output pixel 3, which are all the pixels marked with 2|3 and 1|2|3. Therefore, to compute the pixel number 3 in the bottom layer, all the intermediate pixels marked with 3 in the preceding layers must be computed. To do so, pixels 2|3 and 1|2|3 are read from the top-layer reuse buffer and used to generate the intermediate pixels, which are then processed with the pixels 2|3 and 1|2|3 read from the intermediate layer reuse buffer, to generate the final output value.



1 Output pixel 1
2 Output pixel 2
3 Output pixel 3
1|2 Pixel required by output 1 and 2
2|3 Pixel required by output 1 and 2
1|2|3 Pixel required by output 1,2, and 3
Top layer
Intermediate layer
Bottom layer
Compute 1|2, then reuse buffered 2|3 and 1|2|3 to compute 3

Fig. 3.12 Example of multi-tiling applied to generate two output tiles of two and one output pixels.

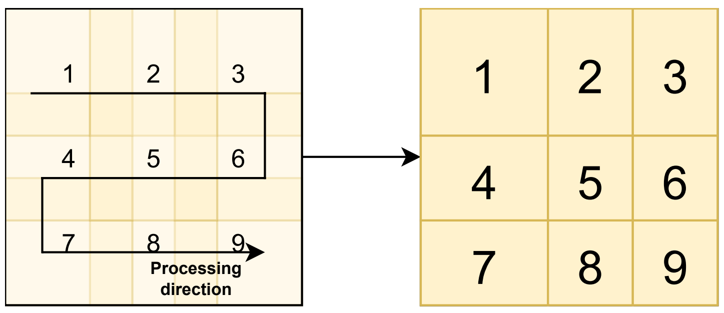


Fig. 3.13 Multi-tiling covers the computation of new unique pixels. Notice how the tiles on the right cover the pixels covered for the first time by the tiles on the left.

To reduce the amount of data saved in reuse buffers, only unique pixels are scheduled for computation to reduce the overlapping. To produce new blocks of unique pixels in intermediate layers, the tile set must be shaped accordingly to exclude the recomputation of pixels stored in reuse buffers. If the example of Figure 3.12 is continued over the entire feature map, the resulting tiles will look like what is depicted in Figure 3.13. These tiles are used to schedule the computation of new pixels every time they are executed. By comparing the left and right squares of Figure 3.12, it can be noticed that the tiles only cover pixels generated for the first time, minus the overlapping with the tiling factors related to previous pixels. To define the spatial dimensions of these tiling factors, it is necessary to evaluate the three horizontal and vertical sizes from which the entire set. can be defined Naming the three horizontal sizes X_1, X_2, and X_3, and the three vertical sizes Y_1, Y_2, and Y_3, it is possible to define with Equation (3.10) the tile dimensions and occurrence. The latter, with the terms X_{reps} and Y_{reps}, represents the number of times a tile is repeated along the X or Y dimension. The padding correction is necessary to avoid scheduling nonexistent pixels covering the borders of the feature map. Finally, it is possible to define the tile sizes in Equation (3.11) using the notation and position of Figure 3.13. The repetitions depend on the size of *Tox* and *Toy* from which this set is generated. A possible set of tiles comprehends 1, 2, 3, 4, 6, or 9 valid tiles. In the case of nine valid tiles, the tile matrix Equation (3.12) represents the repetitions across the feature map.

$$\begin{aligned} X_{\text{reps}} &= \left(\left\lceil \frac{Nox}{Tox} \right\rceil - 2 \right) & Y_{\text{reps}} &= \left(\left\lceil \frac{Noy}{Toy} \right\rceil - 2 \right) \\ X_1 &= Tix - Px & Y_1 &= Tiy - Py \\ X_2 &= Tix - (Nkx - Sx) & Y_2 &= Tiy - (Nky - Sy) \\ X_3 &= Tix - 2 \cdot Px - X_1 - X_2 \cdot X_{\text{reps}} & Y_3 &= Tiy - 2 \cdot Py - Y_1 - Y_2 \cdot Y_{\text{reps}} \end{aligned} \tag{3.10}$$

$$\begin{aligned} Tile\,1 &= X_1 \cdot Y_1 & Tile\,2 &= X_2 \cdot Y_1 & Tile\,3 &= X_3 \cdot Y_1 \\ Tile\,4 &= X_1 \cdot Y_2 & Tile\,5 &= X_2 \cdot Y_2 & Tile\,6 &= X_3 \cdot Y_2 \\ Tile\,7 &= X_1 \cdot Y_3 & Tile\,8 &= X_2 \cdot Y_3 & Tile\,9 &= X_3 \cdot Y_3 \end{aligned} \tag{3.11}$$

$$\begin{bmatrix} Tile1 & Tile2 & Tile3 \\ Tile4 & Tile5 & Tile6 \\ Tile7 & Tile8 & Tile9 \end{bmatrix} = \begin{bmatrix} 1 & X_{\text{reps}} & 1 \\ Y_{\text{reps}} & (Y_{\text{reps}} \cdot X_{\text{reps}}) & Y_{\text{reps}} \\ 1 & X_{\text{reps}} & 1 \end{bmatrix} \tag{3.12}$$

Finally, for a convolutional layer, the total MAC computation with standard- and multi-tiling can be evaluated with Equation (3.13). The total MAC operations for any layer, using the reuse model presented in this section, is bound to be the same with both standard and multi-tiling.

$$\begin{aligned} compute_{\text{standard}} &= Nof \cdot Nif \cdot Nkx \cdot Nky \cdot Nox \cdot Noy \\ compute_{\text{multi}} &= Nof \cdot Nif \cdot Nkx \cdot Nky \cdot \left(\sum_{n=1}^{9} Tile_{\text{n}} \cdot Repetitions_{\text{n}} \right) \end{aligned} \tag{3.13}$$

Hardware Resource Partitioning

Mapping frameworks such as [2, 3, 106, 107, 109] search for efficient spatial and temporal tiling and reordering. Resource partitioning is conducted at the layer level to map the execution of a single workload, whereas with inter-layer scheduling it occurs at the DNN level, to map the execution of multiple layers on a fraction of the available resources. This work explores array and register file (RF) partitioning, namely spatial and temporal fusion, depicted in Figure 3.14. The former is similar to spatial tiling at the array level (unrolling, PE set), and the latter is similar to temporal tiling at the RF level (loop tiling). Since the main focus is the analysis of inter-layer scheduling opportunities, an analysis of the required high-level and PE-level control is omitted, as they would be architecture-dependent.

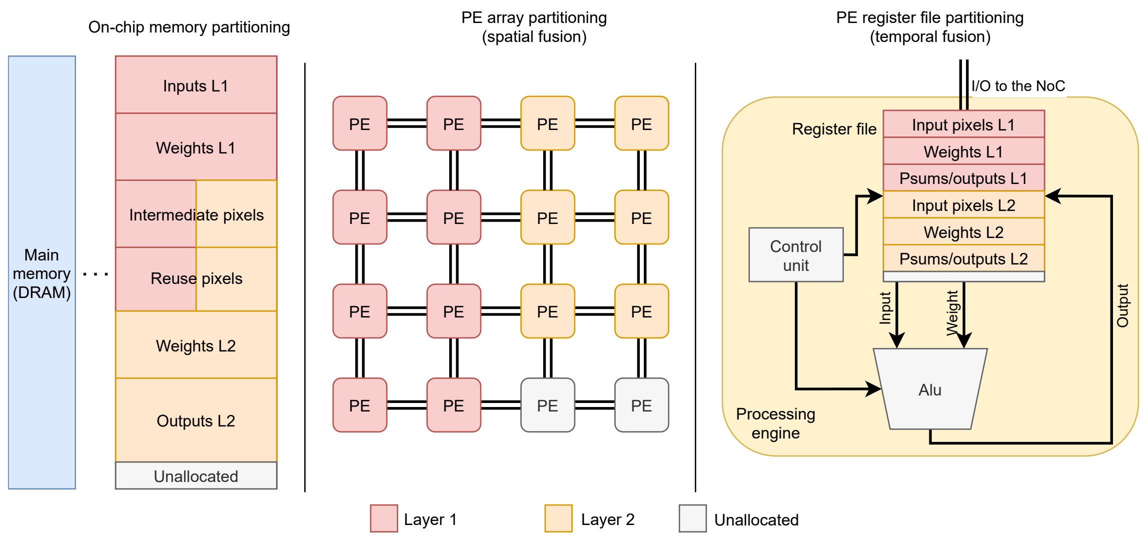


Fig. 3.14 Resource partitioning is always applied on the on-chip buffer. Partitioning is also applied on the PE array, for spatial fusion, and on the RFs, for temporal fusion. A hypothetical 2-layer fusion mapping is depicted, showing allocated memory and array regions. With both single- and inter-layer scheduling, it is possible to have reduced resource utilization due to spatial/temporal tiling factors that are not divisors of the dimensions of the array or the memory. PEs partitioned with either spatial or temporal policies use the on-chip memory to communicate with PEs assigned to other layers.

Spatial Fusion: It is the partitioning of the PE array to compute multiple layers sharing the same on-chip memory and reuse buffers. This mapping models an accelerator with a PE array that can be split into smaller regions, or PE sets. Each PE set computes a different layer and communicates with the on-chip buffer to read and store the data. Each PE set computes the output of a specific layer and then transmits it to the on-chip memory, which is then read by another PE set that computes the output pixels of another layer and repeats the same process. The objective of spatial fusion is to find the size of sub-arrays like those depicted in Figure 3.14. It is impossible to know the best partitioning beforehand, as any combination of sub-arrays with any widths and heights could be the best one for the chosen performance metrics. Each sub-array will execute only one layer within the fusion schedule, so the partitioning must optimize the resource allocation of each workload. To find the best partitioning for N number of layers, the Hw-Flow-Fusion evaluates all the N sub-arrays that can fit within the original

PE array without overlapping, using the area fitter methodology of [149] to remove illegal solutions. Only sub-arrays of height and width that are even divisors of the PE array dimensions are searched to reduce the solution space, which is further reduced by removing solutions known to be illegal for certain dataflows. For instance, with a row-stationary dataflow, it is possible to remove arrays with a height that is smaller than the horizontal dimension of the kernel, as they would not have any legal mapping, as detailed in [52]. Finally, each tile set of each fused layer is mapped for each combination of sub-arrays. Then hardware metrics are collected and compared to find the most energy/throughput-efficient partitioning. A drawback of spatial fusion is that the unrolling factors of the fusion schedule are smaller than those of a single-layer schedule, being constrained by the sub-array sizes, which might result in increased computation latency.

Temporal Fusion: involves dividing the RF of each PE in the array for each layer of the fusion schedule. This mapping models an accelerator where the RF within the PEs can store input, weight, and output pixels for different layers, interleaving their computation. This concept resembles to CPU multi-threading, enabling a single core (PE) to handle multiple threads (layers) that share the same cache (RF) and datapath (MAC). When there are insufficient input pixels to compute a partial sum or output pixel, the execution of a layer is temporarily halted, and the PE switches to processing another layer with available data in the RF. In theory, this approach can diminish PE idle time by ensuring a constant flow of tasks to process and is similar to interleaving, leveraged by dataflows such as row-stationary in [52] and output-stationary in [126] to compute multiple input or output channels within each PE. Reducing the RF size available to each layer due to resource sharing, and therefore interleaving capacity, should not significantly impact the performance [128]. At the same time, unrolling the execution of each layer over the original array size (compared to spatial fusion) should result in lower computation latency and higher energy efficiency, as both strictly depend on the number of PEs and the possibility of having larger spatial tiling factors (not constrained in temporal fusion). In the hardware model adopted by Hw-Flow-Fusion and [2, 3, 107], the RF of a certain PE is divided into three sections: one for the weights, one for the inputs, and one for the partial sums. In inter-layer scheduling, each of these sections has to be partitioned into N sub-sections, one for each layer. The search space is limited to one combination for each fusion schedule, which is obtained by dividing the input, weight, and partial sums RF into N sub-sections, respectively. For instance, the RF of a standard row-stationary model accelerator in [3, 107] is divided into 384 bytes for the weights, 24 for the input pixels, and 36 for the partial sums. In the case of a fusion schedule with two layers, with N = 2, each layer would have 192, 24, and 36 bytes for the weights, input pixels, and partial sums, respectively.

Proposed Inter-Layer Scheduling Framework

The inter-layer scheduling framework HW-Flow-Fusion, depicted in Figure 3.15, is built on Hw-Flow, presented in Section 3.1, which is used for single-layer scheduling. Valid hardware resource partitions for spatial and temporal fusion are evaluated as explained in the previous paragraphs; tile sets are evaluated using Equation (3.10) and Equation (3.12), whereas buffering requirements are evaluated with Equation (3.5) and Equation (3.8), as explained above.

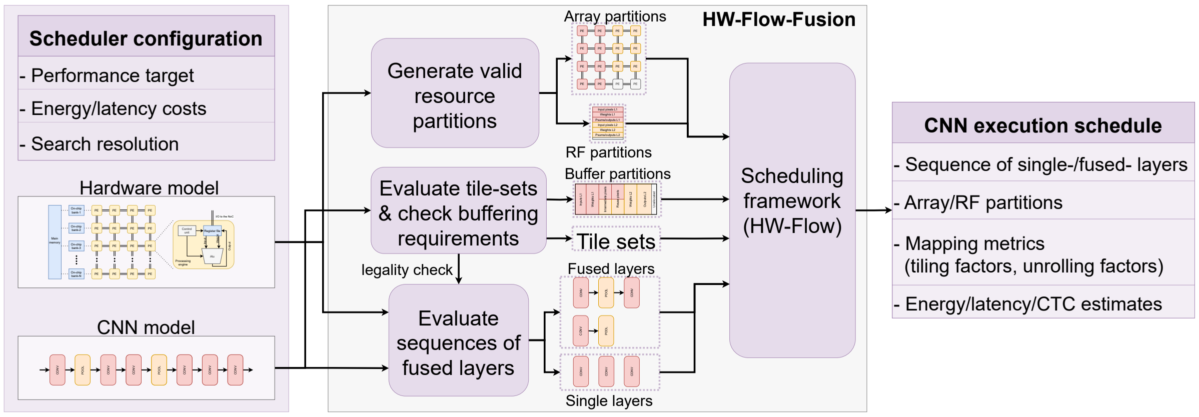

Scheduler configuration
- Performance target
- Energy/latency costs
- Search resolution
Hardware model
CNN model
HW-Flow-Fusion
Array partitions
Generate valid resource partitions
RF partitions
Buffer partitions
Evaluate tile-sets & check buffering requirements
Tile sets
legality check
Fused layers
Evaluate sequences of fused layers
Single layers
Scheduling framework (HW-Flow)
CNN execution schedule
- Sequence of single-/fused- layers
- Array/RF partitions
- Mapping metrics (tiling factors, unrolling factors)
- Energy/latency/CTC estimates

Fig. 3.15 Overview of the inter-layer scheduling framework. The framework takes as input a DNN model with layer types and dimensions, an abstract hardware model with memory and PE array specifications, and scheduler settings such as the energy/delay model and search resolution. Valid partitions, tile sets, and sequences of layers are evaluated and marked for single- or inter-layer scheduling. The output schedule is composed of the mapping parameters and performance metrics.

The process for generating and mapping the sequence of fused layers initiates by assessing sequences of layers that can be fused within the constraints of the available hardware resources. Starting with a sequence comprising a minimum of two layers, every combination of tile sets is examined to verify the validity of the fusion schedule. This involves comparing the total allocated memory with the buffer size available. A solution is deemed legal and preserved for subsequent evaluation if the total memory occupation is either less than or equal to the available memory size. If at least one valid solution exists for the current sequence of layers, a new adjacent layer is added, and the validity check process starts anew. This iterative process persists until no valid sets exist for a given sequence. When this happens, the cycle starts again with a new sequence, beginning with the first layer that could not be fused and progressing to the subsequent one. Multiple layers can be included in different fusion schedules. For instance, if the first four layers of a DNN can be fused, this procedure outputs three legal fusion schedules: one for the first two layers, one for the first three layers, and one with all four layers. This process continues until the entire DNN has been analyzed and outputs a set of sequences and single layers, which compose the solution space searched by the scheduling framework. The scheduler receives the corresponding layer dimensions, tile sets, and buffer/RF/array partitions for each sequence of fused layers or single layers. The fused sequences are compared against each other and against the corresponding singularly scheduled layers. The final mapping includes only the fusion schedules that improve the target hardware metrics with respect to singularly scheduled layers. The hardware metrics are evaluated using the same architecture and energy/delay models of Section 3.1. The energy and delay of each operation (MAC operation, RF read, NoC transfer, etc.) are multiplied by the total access count specific to that resource. For instance, the DRAM energy for each operation is multiplied for the write access count evaluated as $Output_{write}$ in Equation (2.20).

3.2.3 Results

Six hardware configurations of a row-stationary spatial-array accelerator are used in the experiments, with the model architecture and scheduling constraints based on Eyeriss [52] and Timeloop [107], reported in Tables 3.1 and 3.2. Each configuration is simulated with a batch

Config.	PEx	PEy	on-Chip Buffer (kB)	RF (B)	Precision
1	32	16	512	512	8
2	32	16	1536	512	8
3	32	32	1536	512	8
4	48	32	1536	512	8
5	32	16	1536	1024	8
6	32	16	1536	1536	8

Table 3.1 Hardware configurations of the Eyeriss models used in the experiments.

size of one and 8-bit data precision. The energy for MAC operations and data movement is evaluated as in [2, 3]. The MAC energy cost comprehends the contribution of one multiplication, one addition, two memory read operations to fetch the operands, and one memory write operation to write the results, all from/to the RF. Any memory transfer comprehends one read and one write operation between different levels. The on-chip buffer bandwidth is 2 bytes per cycle per bank, and the DRAM burst length is 8 bytes. Bandwidth efficiency is evaluated with the refined CTC proposed in Caffeine [131].

Reuse Buffer Comparison and Impact of On-chip Memory on Layer-Fusion

The memory required to fuse multiple layers depends on the constraints $Nif = Tif$ and $Nof = Tof$, which force the entire weight volume to stay on-chip. As depicted in Figure 3.17, large on-chip

Config.	MAC 8 bit [pJ]	on-Chip Buffer Access [pJ]	DRAM Access [pJ]
1	1.75	26.70	200.0
2, 3, 4	1.75	78.16	200.0
5	1.79	78.16	200.0
6	1.83	78.16	200.0

Table 3.2 Energy costs for each hardware configuration.

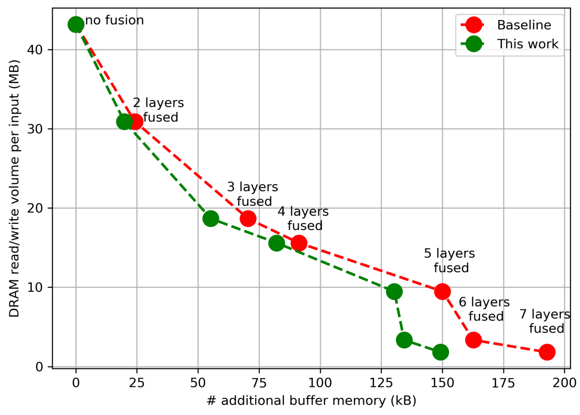


Fig. 3.16 Reuse buffer comparison.

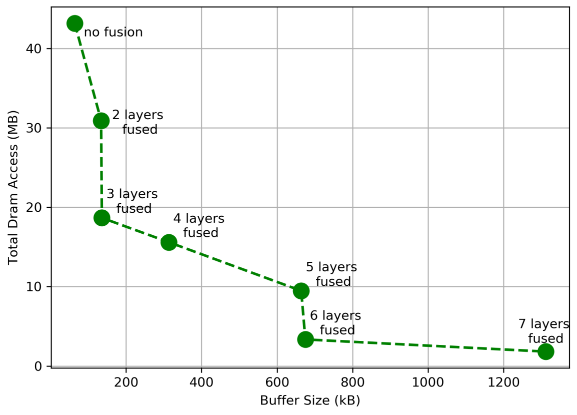


Fig. 3.17 Minimum storage to fuse VGG16 layers.

buffers allow to schedule more layers, but increasing its size is also antagonistic to energy efficiency [128]. The reuse buffer strategy proposed in this work is compared to [146] using the same DNN model, a VGG16-E [11], with 16-bit quantization. The first seven layers, CONV1-CONV2-POOL1-CONV3-CONV4-POOL-CONV5, are scheduled. The proposed reuse model improves over [146], representing the baseline in Figure 3.16. The additional buffer memory is evaluated as kB of reuse buffer allocated to achieve zero recomputation. The reuse model presented in this work outperforms [146], requiring from ~15% to ~22% less additional memory to fuse the same number of layers and achieve the same communication volume reduction. The gap widens when the number of fused layers increases, as deeper layers have more feature maps and the size of each sequential overlap that must be stored with the reuse model [146] increases. Regarding the reduction of the communication volume of Figure 3.17, DRAM accesses are reduced by ~60% when fusing the first three layers. There is almost no difference in memory requirements when fusing two or three and five or six layers because the third and sixth max pooling layers require no weight memory.

Comparison of Spatial Fusion, Temporal Fusion, and Single Layer Hardware Metrics

The hardware metrics estimated with standard scheduling, here named single-layer, are compared against those estimated with inter-layer scheduling using ResNet-18 and VGG16-E, with the hardware configuration 1 of Table 3.1, setting the maximum number of fused layers to two. Only the valid fusion sequences are compared against single-layer scheduling. The energy and latency estimates for ResNet-18 and VGG16-E are reported in Figure 3.18, the overall energy and latency ratios are reported in Table 3.3, adding the contributions for each layer in the sequence. Only six couples of layers could be fused with this hardware configuration because the on-chip buffer size limits fusion possibilities once the number of channels starts to increase in deep layers. For VGG16-E, the buffer size prevents the fusion of any layer after the sixth. However, there are two max pooling layers that precede two convolutions, allowing the scheduler to find two additional fusion sequences. For ResNet-18, fusing couples of consecutive layers up to the twelfth convolutional layer is possible.

	$\frac{Temporal\,fusion}{Single-layer}$		$\frac{Spatial\,fusion}{Single-layer}$		$\frac{Spatial\,fusion}{Temporal\,fusion}$	
NN	**VGG16-E**	**ResNet-18**	**VGG16-E**	**ResNet-18**	**VGG16-E**	**ResNet-18**
Energy	94%	91%	94%	90%	100%	98%
Latency	61%	66%	114%	118%	188%	180%
Com. volume	49%	47%	49%	48%	100%	101%

Table 3.3 Energy, latency, and off-chip to on-chip communication volume comparison with the sum of single- and inter-layer scheduling of the sequences of layers reported in Figure 3.18.

Temporal fusion can reduce the energy, latency, and communication volume for almost every sequence. In contrast, spatial fusion can only improve the energy and communication volume with a significant increment of the latency. Temporal fusion reduces the overall latency to 66% for ResNet-18 and to 61% for VGG16. Spatial fusion can match or slightly improve the energy with respect to temporal fusion. The major contribution to the latency is due to the computation rather than communication, as spatial fusion achieves approximately the same communication volume as temporal fusion. A rise in the computation latency is normal as it depends on the unrolling rate and available PE, as demonstrated in [128]. By analyzing the results of Figure 3.18, it is possible to conduct the same observations in [128] regarding architectural design choices for high throughput, which is that the number of PEs is more important than the size of the RF. It is worth recalling that temporal fusion uses the entire PE array and reduced RF for each layer, whereas spatial fusion uses reduced PE arrays and the entire RF for each layer. In this experiment, the scheduler found the array partitioning reported in Table 3.4. The array partitioning search space is limited to sub-arrays with spatial dimensions that are divisors of the original PE array size, whereas the RF partitions are fixed to $\frac{1}{N}$ of the original size, with N being the number of layers in the fusion sequence. Layers scheduled with spatial fusion are executed using partitions that are half of the original PE array size, with one exception for the first two layers of VGG16-E, where one quarter of the original array is assigned to each layer.

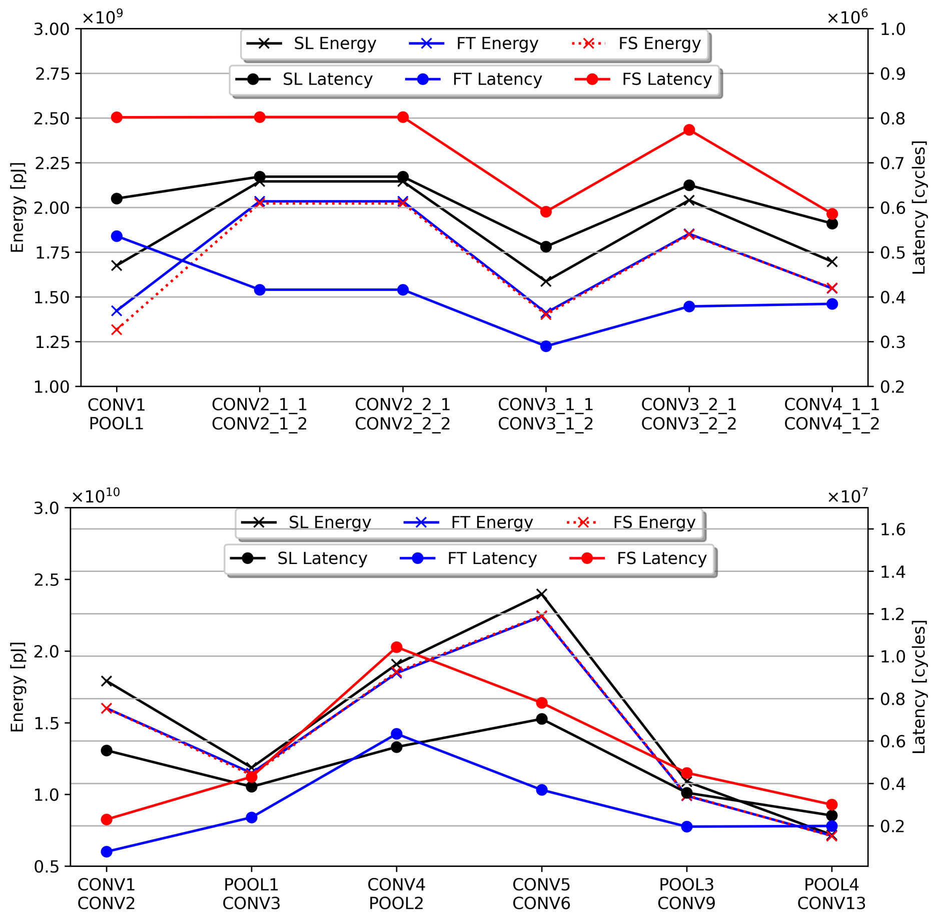


Fig. 3.18 ResNet-18 (top) and VGG16-E (bottom) single layer (SL) vs. temporal fusion (FT) vs. spatial fusion (FS) on hw config. 1.

Therefore, the communication latency improvement achieved with inter-layer scheduling is lost for spatial fusion due to the higher computation latency resulting from executing each workload on fewer PEs concerning the single-layer scheduling approach. The energy consumption of spatial and temporal fusion is marginally improved with respect to singularly scheduled layers because only the communication energy is improved with the inter-layer scheduling method presented in this work. Further investigation of inter-layer scheduling should focus on reducing data movement between the RF and on-chip buffer by directly forwarding computed output pixels from one processing engine to another, adopting a low-level memory manager as the one proposed in [128] or in [148]. The major contributions to the overall energy consumption come from the data movement within the PE array and the on-chip memory accesses, as demonstrated in [52, 107]. Moreover, the overall communication volume is halved for both DNNs, a result that could have been anticipated for VGG16-E by observing Figure 3.17. With CONV1 and

CONV2 fused, the communication volume is reduced by ~25%, with POOL1 and CONV3 by ~10%, and with layers CONV4 and POOL2 by ~17%. While there are three more couples of fused layers in Figure 3.18, these six layers dominate the pixel communication volume.

Table 3.4 Resource partitioning found by the scheduler for ResNet-18 and VGG16-E layers. Only PE array partitions are actually searched, as the RF sizes are fixed during the search.

ResNet-18			**VGG16-E**		
Layer	**Spatial PE Array [PEx, PEy]**	**Temporal RF [Wgt, In, Out]**	**Layer**	**Spatial PE Array [PEx, PEy]**	**Temporal RF [Wgt, In, Out]**
CONV1	[16, 16]	[192, 12, 16]	CONV1	[8, 8]	[192, 12, 16]
POOL1	[16, 16]	[192, 12, 16]	CONV2	[8, 8]	[192, 12, 16]
CONV2_1_1	[32, 8]	[192, 12, 16]	POOL1	[32, 8]	[192, 12, 16]
CONV2_1_2	[32, 8]	[192, 12, 16]	CONV3	[32, 8]	[192, 12, 16]
CONV2_2_1	[32, 8]	[192, 12, 16]	CONV4	[32, 8]	[192, 12, 16]
CONV2_2_2	[32, 8]	[192, 12, 16]	POOL2	[32, 8]	[192, 12, 16]
CONV3_1_1	[32, 8]	[192, 12, 16]	CONV5	[32, 8]	[192, 12, 16]
CONV3_1_2	[32, 8]	[192, 12, 16]	CONV6	[32, 8]	[192, 12, 16]
CONV3_2_1	[32, 8]	[192, 12, 16]	POOL3	[16, 8]	[192, 12, 16]
CONV3_2_2	[32, 8]	[192, 12, 16]	CONV9	[32, 8]	[192, 12, 16]
CONV4_1_1	[32, 8]	[192, 12, 16]	POOL4	[16, 16]	[192, 12, 16]
CONV4_1_2	[32, 8]	[192, 12, 16]	CONV13	[16, 16]	[192, 12, 16]

Finally, the improvements in the CTC ratio are analyzed, recalling that it defines the maximum computation performance achievable with a certain bandwidth. The CTC ratio, defined in Equation (2.24), can be increased by reusing more data or changing the memory hierarchy, which might not be possible due to energy/area constraints. By removing the off-chip communication between intermediate layers and reusing intermediate pixels, inter-layer scheduling can outperform single-layer scheduling. The CTC ratios are reported in Table 3.5, with a more noticeable improvement in layers dominated by pixel volumes, such as the first couple of layers of VGG16-E, and generally in fusion sequences containing a POOL layer at the end. Equation (2.24) can be helpful to understand the low CTC ratio in deeper layers, recalling that it also depends on the burst length and, therefore, how much continuous data are read from the off-chip memory. Moving multiple small sequences of pixels is more inefficient than moving a few large sequences. As the proposed strategy requires the entire weight volume saved on-chip, the available intermediate pixel storage shrinks when the weight storage increases in deeper layers, resulting in smaller tile sizes that can fit on-chip. This effect generates a communication pattern of small memory read and write operations, increasing the communication latency and reducing the memory bandwidth efficiency. Additionally, as can be observed in Figure 3.13, the tile sets used in the processing of pixels around the center of the feature map spatial dimensions are inherently smaller than the others and might not use all the available on-chip buffer. Currently, the scheduler selects the biggest tile set with the largest memory footprint to perform the legality check of the fusion schedule. A possible solution to increase the bandwidth efficiency could be to regroup the tile sets with a repetition count higher than one, in particular, referring to Equation (3.11), tiles 2, 4, 5, 6, and 8.

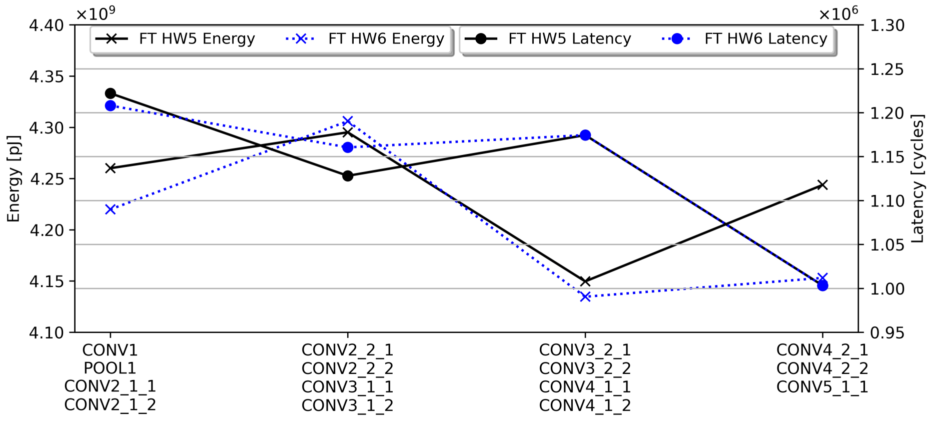
Table 3.5 CTC ratios for ResNet-18 and VGG16-E for the single- and inter-layer schedules of Figure 3.18. For single-layer the CTC ratios are reported for each layer, whereas for spatial and temporal fusion the CTC ratios of the fused sequences of layers are reported.

ResNet-18				**VGG16-E**			
Layers	**Single**	**Temporal**	**Spatial**	**Layers**	**Single**	**Temporal**	**Spatial**
CONV1 POOL1	3.52 0.03	9.62	9.48	CONV1 CONV2	0.80 6.97	58.06	58.06
CONV2_1_1 CONV2_1_2	5.03 5.03	14.65	14.65	POOL1 CONV3	0.05 9.50	15.09	15.09
CONV2_2_1 CONV2_2_2	5.03 5.03	14.65	14.65	CONV4 POOL2	12.91 0.053	23.02	23.93
CONV3_1_1 CONV3_1_2	4.73 9.95	10.90	10.80	CONV5 CONV6	8.53 9.68	42.47	42.47
CONV3_2_1 CONV3_2_2	9.95 9.95	9.90	9.90	POOL3 CONV9	0.05 15.95	11.76	11.76
CONV4_1_1 CONV4_1_2	3.99 5.12	5.10	5.05	POOL4 CONV13	0.05 5.58	4.83	4.83

Hardware Constraints and Scaling on Inter-Layer Scheduling

The simulations in this section aim to understand how layer fusion performance scales with different hardware resources. In particular, the hardware configurations 3 and 4 of Table 3.1 are used to measure if spatial fusion can leverage the additional PEs by increasing the array size when fusing, at most, four layers. Similarly, hardware configurations 5 and 6 are used with temporal fusion for the same purpose, with additional RF memory for each PE. In both experiments, the on-chip buffer size was set to 1536 kB to increase the solution space explored by the scheduling framework, relaxing the legality checks done while evaluating the sequences of layers that can be fused. The results with temporal fusion are reported in Figure 3.19. The scheduler can fuse at least four layers for configurations 5 and 6, with a minimum RF memory-per-layer equal to or higher than the one allocated for two layers with hardware configuration 1. No performance scaling is observed when the RF size is increased since the energy and latency metrics with HW5 and HW6 of Table 3.1 are improved less than 1% for energy, latency, and CTC ratios, confirming that the RF size does not significantly impact on the performance, coherently with [128].

Figure 3.20 reports the performance scaling with spatial fusion for ResNet-18 and VGG16-E. The overall energy difference with single-layer scheduling for both DNNs with the two hardware configurations is in the range of $\pm 1\%$. Similarly to the results for spatial fusion



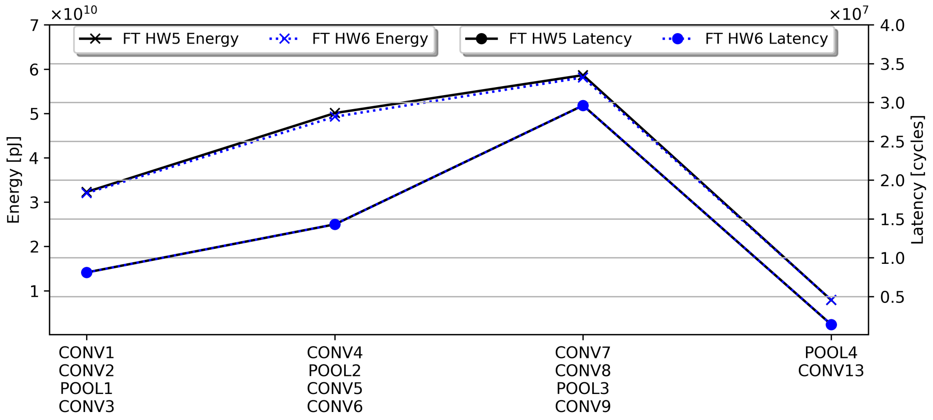
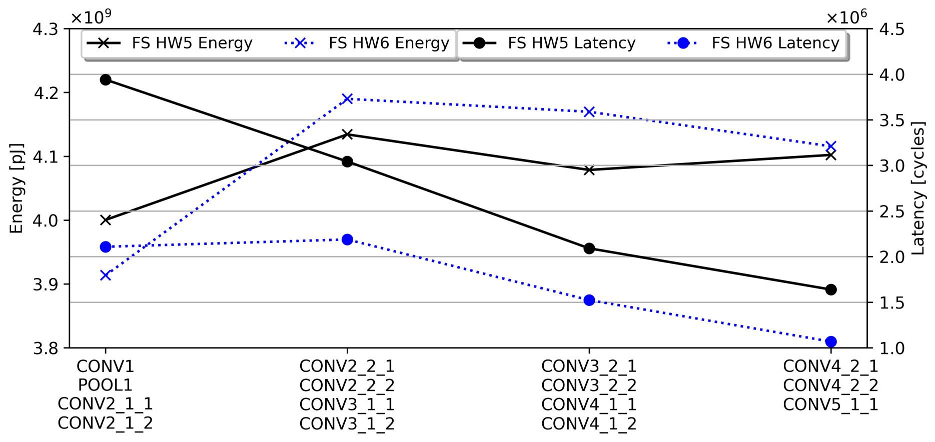


Fig. 3.19 ResNet-18 (top) and VGG16-E (bottom) mapped with temporal fusion on hw config. 5 and 6.

of Figure 3.18, no noticeable improvement in energy efficiency was achieved by scaling the PE array size. On the other hand, the latency was significantly improved, with a 45% latency reduction between hardware config. 5 and 6 for ResNet-18 and 75% for VGG16-E. These results again correlate with the observations made in [128] and prove that spatial fusion can benefit from additional PEs. A larger PE array means that the scheduler can unroll the execution of each layer over larger partitions, reducing the constraints on the unrolling factors and resulting in much lower computation latency.

Discussion

The last section of this chapter demonstrates that the proposed inter-layer scheduling framework can be used to evaluate the performance of different fusion strategies, providing a clear understanding of the trade-offs between energy, latency, and throughput. Temporal fusion proved to be the best choice for small accelerators such as the one modeled in the hardware



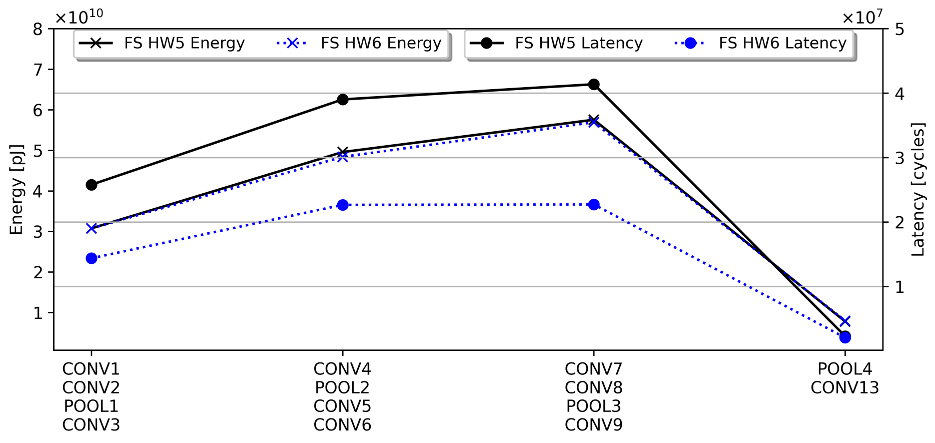


Fig. 3.20 ResNet-18 (top) and VGG16-E (bottom) mapped with spatial fusion on hw config. 3 and 4.

config. 1, which is similar to [52, 150] but does not scale with additional resources. In contrast, spatial fusion is severely limited on small PE arrays; however, it might be the best option to improve the computing performance on large accelerators, such as those modeled with hardware config. 3 and 4, which are similar to [50, 51]. The scheduler has yet to be implemented and tested with real hardware, and the results presented in this chapter are based on simulations. Therefore, future work should focus on developing hardware accelerators with proper support for inter-layer scheduling.

Chapter 4

Compressed and Error Resilient Deep Neural Networks at the Edge

The deployment of DNNs on edge devices has become pivotal for various applications, enabling advanced functionalities, but also emphasizing the need for efficient model compression techniques. Two key approaches, pruning and quantization, are used to deploy DNNs on resource-constrained edge devices [49, 62]. As edge devices play an increasingly pivotal role in real-time sensory processing and decision-making, it is imperative to address the challenge of achieving an optimal model compression without compromising robustness to hardware errors and adversarial attacks [7]. Achieving this balance is crucial to ensure the reliability and security of edge devices, especially in safety-critical applications, such as automotive [134, 135]. This chapter addresses the delicate trade-off between model optimization and preserving the DNN's robustness to numerical error. Parts of the works presented here have been previously published in [6, 8, 9].

Section 4.1 presents a low-complexity and architecture-agnostic approach to mitigate unwanted numerical errors in the computation of CNNs for object detection, such as bit-flips due to logic transients caused by several natural and artificial sources, leveraging motion estimation techniques to predict the future position of known objects, restoring the correct detections and eliminating the wrong ones.

An automated codesign methodology for the deployment of layer-wise approximate DNNs on IoT devices is introduced in Section 4.2. Shallow and deep DNNs are optimized with a genetic algorithm that searches for low-energy and high-accuracy configurations, assigning a different approximation level to each layer. A runtime reconfigurable AxM enables this design methodology without using redundant arithmetic units.

4.1 Error Resilient Object Detection with Automated Output Correction

This section introduces a low-complexity method to detect and correct computation errors occurring during the inference of CNNs for object detection. This approach utilizes motion estimation techniques to forecast the future positions of recognized objects, mitigating temporary incorrect detections. The errors are identified and removed from the output by leveraging the spatio-temporal correlation between successive input images and output predictions, restoring the accurate output that might have been compromised without intervention. Called **ERODE** (Error Resilient Object DEtection), this technique is agnostic to the specific CNN model or hardware architecture executing it. The ERODE framework is designed as a plug-in to seamlessly integrate with CNN-based object detection systems, safeguarding task accuracy in the presence of errors.

4.1.1 Detection and Correction of Inference Errors in Previous Works

Draghetti et al. [136] propose using inter-frame spatio-temporal correlation to identify errors in CNN inference. The fundamental assumption is that a minimal absolute pixel difference between consecutive images implies nearly identical inputs, leading to very similar CNN predictions. Deviations from this condition suggest a potential inference error. However, reliance on a user-defined threshold for similarity measurement introduces several weaknesses. Variations in brightness, noise, or camera movement may trigger inaccurate error detection, requiring an adjustment of the threshold for every different environmental condition. Overcoming this limitation, ERODE leverages relative pixel differences and established motion estimation techniques [151, 152], assessing input-output spatio-temporal correlation across multiple consecutive images without using fixed thresholds. Some parameters, which do not affect the error-detection capabilities but the system filtering action and selectivity, still require input from the user. Furthermore, a comprehensive examination of faults during quantized CNN model inference and their impact on task accuracy is conducted in [7, 133]. These studies underscore that activation errors significantly influence CNN accuracy more than weight errors. Additionally, [7, 132] observes that recent CNNs demonstrate inherent resilience to errors, especially when compressed through quantization for deployment. This study specifically targets bit-flips in activations, utilizing EfficientDet D0, a modern CNN for object detection [15], quantized to 16-bit.

4.1.2 Proposed Methodology

ERODE leverages the spatio-temporal correlation between consecutive images in a video sequence to update the position of detected objects using motion estimation techniques, adjusting the bounding box size and coordinates without processing the image with the CNN. Additionally, velocity and direction are evaluated to check whether the trajectory of objects is constant or changes. This set of predictions and additional properties is compared against the inference results and is used to assess the presence of errors by searching for abrupt changes in the identified objects' properties. For example, sudden alterations in the label assigned to a bounding box that previously held a different label or an instantaneous acceleration of a bounding box that was previously stationary might indicate computation errors during the inference process. This approach enhances the robustness of object detection by validating predictions against motion-derived expectations, contributing to error identification and correction.

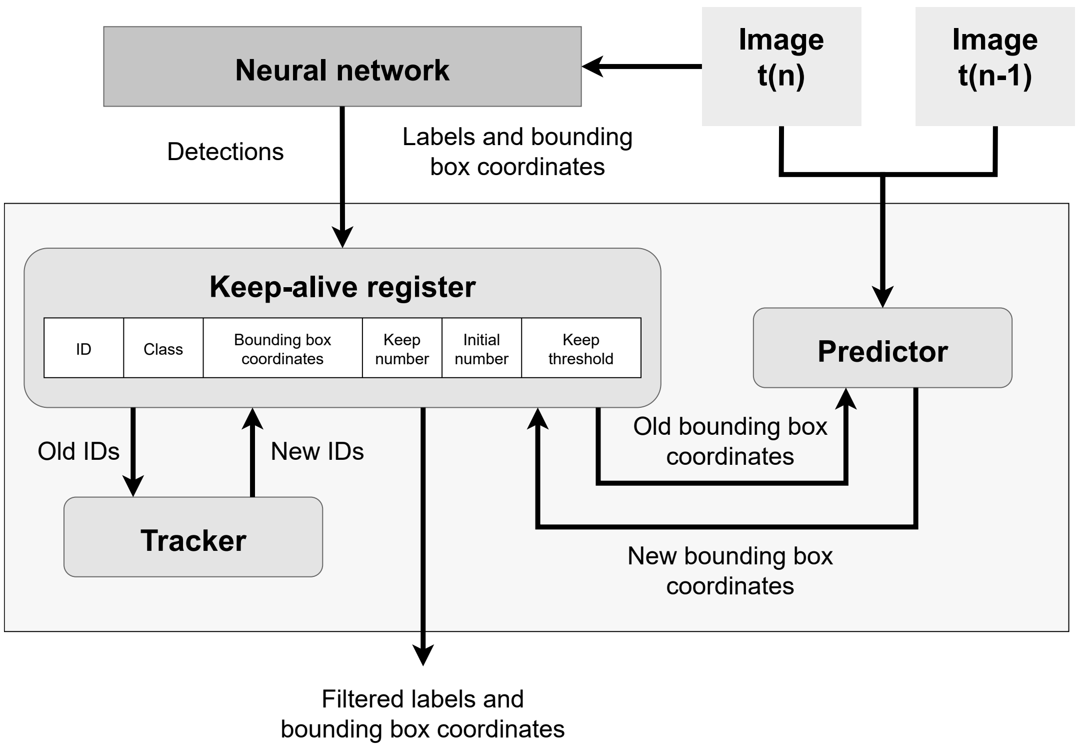
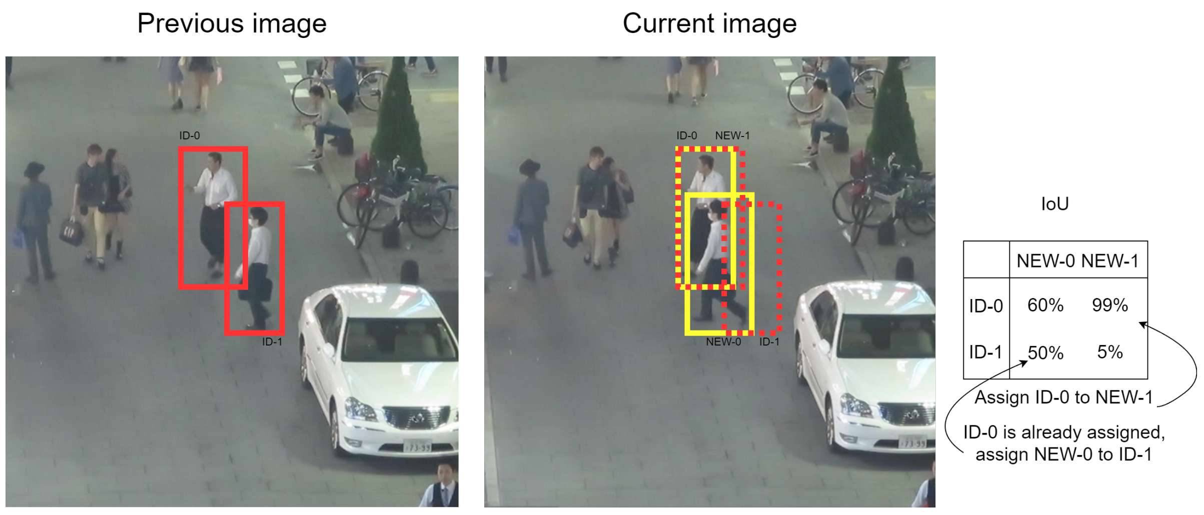


Fig. 4.1 The ERODE framework building blocks.

The ERODE framework, depicted in Figure 4.1, comprises three parts: the tracker, the keep-alive register, and the predictor. The CNN generates the predictions, i.e., the bounding boxes and the corresponding label, that are passed to the tracker, which assigns a unique ID to each detected object. ERODE saves the IDs in a data structure called keep-alive register to identify the presence of objects through multiple images and selects which objects are relevant, i.e., persistent. The predictor updates relevant object features (bounding box coordinates, dimensions, and velocity) using motion estimation techniques [151, 152]. The system's output comprises the keep-alive register entries, which substitutes the CNN's output predictions.

Tracker

The tracking algorithm assigns unique IDs to all the objects in the image, which are used to observe their behavior in different time instants. In this work, the sorting algorithm [153] is used to create a correlation between two consecutive CNN outputs, applying the Hungarian algorithm [154] to solve the assignment problem, using the IoU metric to compute the cost function. A cost matrix for each detection is generated from previously identified objects with labels and bounding boxes. The entries of the keep-alive register at $n-1$ (previous time-instant/image) are inserted in the rows, and the new detections made by the CNN at n (current image) are inserted in the columns. After that, the IoU between rows and column entries are evaluated. A high IoU proves that the position in the current time instant is similar to the position in the past; therefore, objects of consecutive images can be assigned with a unique ID encoded as a positive integer. The ID assignment process is depicted in Figure 4.2, in the case of two objects present in consecutive frames.



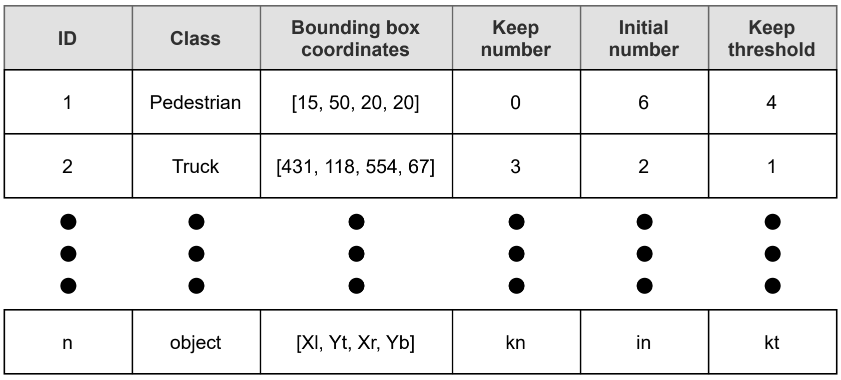
	NEW-0	NEW-1
ID-0	60%	99%
ID-1	50%	5%

Fig. 4.2 The IDs of old detections (red) are saved in the cost matrix and assigned to the new detections (yellow) according to the IoU value, prioritizing detections with higher overlapping.

On the other hand, objects with a low or zero IoU are considered new entries and are assigned with the first positive number available in the keep-alive register. New entries can be either new objects identified by the CNN or glitches due to computation errors. Objects of the previous time instant with no association with those detected in the current one could no longer be present in the image or not detected due to computation errors.

Keep-alive Register

The decision to assign, add or remove IDs is made considering the features related to every single object stored in the keep-alive register. When an object enters the image and is successfully detected by the CNN, it is immediately stored in the keep-alive register, depicted in Figure 4.3.



ID	Class	Bounding box coordinates	Keep number	Initial number	Keep threshold
1	Pedestrian	[15, 50, 20, 20]	0	6	4
2	Truck	[431, 118, 554, 67]	3	2	1
⋮	⋮	⋮	⋮	⋮	⋮
n	object	[Xl, Yt, Xr, Yb]	kn	in	kt

Fig. 4.3 The keep-alive register is a data structure that contains features produced by the CNN and by ERODE. The two objects in this figure are included to provide a hypothetical utilization.

The keep-alive register is scanned for each detection to see if any objects are already stored. If the object is found, that entry is updated with the newly detected features; if not, a new entry is created with the first available ID. The main function of the keep-alive register is to filter out detections that could result from correct or faulty inference. In order to discern correct detections from errors, only the objects present in multiple frames are shown in the system's output. To do so, three user-defined parameters, the *initial number threshold*, the *keep number threshold*, and the *keep-alive threshold*, can be set to tune the selectivity of the filtering action and affect the precision and recall metrics. The *initial number threshold* sets the minimum consecutive detections for each object before they are considered persistent or correct and not glitches due to computation errors, and is the same for all the classes and all the entries in the keep-alive register. The *initial number threshold* concept is borrowed from [155] and indicates the degree of selectivity of the filter: a low threshold means that few consecutive detections are required to accept the object as an output of the system. The *initial number* counts the number of consecutive detections from the first one of the same object, and if it reaches the *initial number threshold*, the object can be considered as correctly detected. On the contrary, the object is deleted from the keep-alive register because it is identified as an error. Any object in the keep-alive register can have two different fates when the subsequent detection is performed: if detected, the algorithm assesses that it is present in the image, and so that the detection is correct, whereas if it is not detected, it could be due to an error or to the object leaving the image. Similarly, objects that are not detected for a few frames could no longer be present in the image or be the result of computation errors. The *keep number* counts the number of consecutive frames in which the CNN does not detect an object and is reset to zero each time it is detected. Therefore, the *keep number* indicates when a prediction is needed: if it is equal to zero, the object's position is given by the detector, on the contrary, it must be evaluated using the predictor if the value exceeds the *keep number threshold*. The latter parameter can be used to reduce the arithmetic operations required by ERODE to process a frame, limiting the usage of the predictor, trading-off computational complexity with worsened error detection and recovery capabilities. Since the predicted position is a guess and the object can leave the image anytime, a limitation on the number of consecutive times a prediction can be used is given [156]. This limit is set by the *keep-alive threshold*, a user-defined parameter that sets the maximum consecutive non-detections before the object is removed from the keep-alive register. It is dynamically tuned according to the CNN's output: it is increased for each consecutive detection and, conversely, decreased for each non-detection.

Predictor

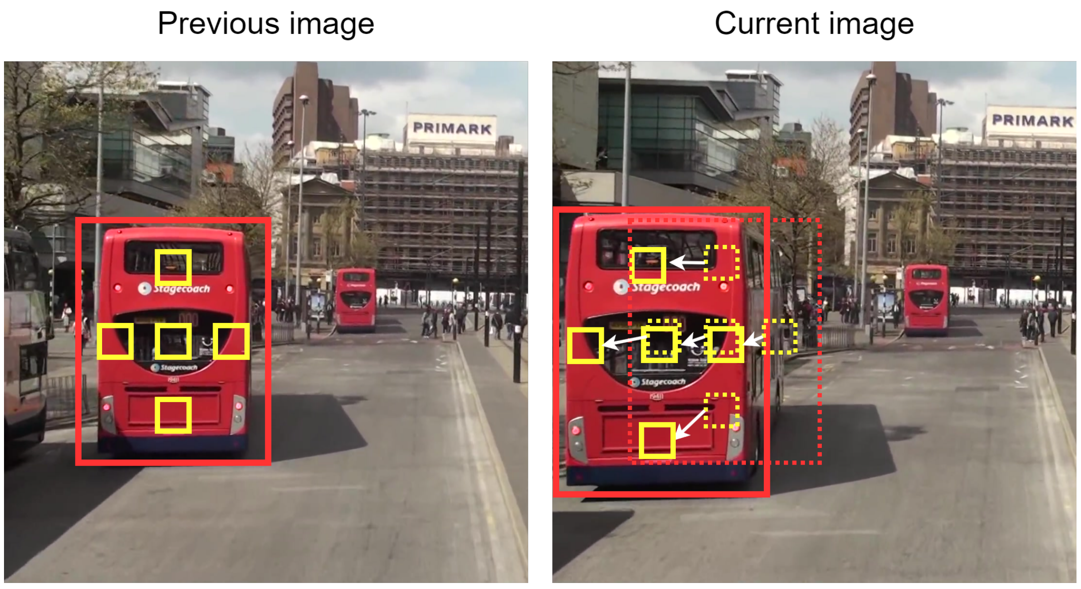


Fig. 4.4 Motion tracking is done using patches of pixels centered around five points of interest with the same distance from the bounding box's center. The new position of each patch is evaluated using diamond search. The relative movement of the patches is used to move and resize the bounding box.

The predictor is used when an object is undetected for one or more consecutive frames and the *keep number* is greater than zero. The predictor updates the position and dimension of known objects by searching for similarities between the current and previous image, using points of interest derived as the center of the bounding boxes, ordered in a cross-like shape as depicted in Figure 4.4. The predictor uses the diamond search algorithm proposed in [152], with a search window of 15 pixels, to estimate the movement and new coordinates of the bounding boxes for each entry in the keep-alive register. This motion estimation algorithm was selected due to its low computational complexity and adaptability to different ranges of motions.

4.1.3 Results

Figure 4.5 details the computing setup and results, which comprises the NN, the error injection algorithm, the ERODE framework, and the benchmarks.

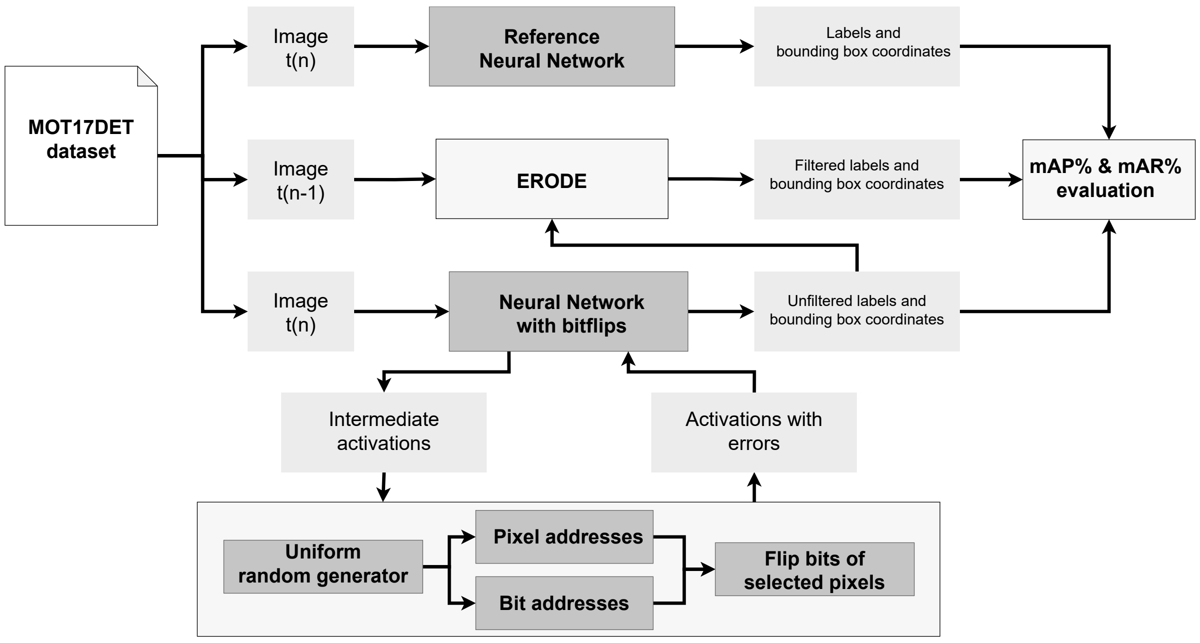


Fig. 4.5 The computing setup used in the experiments. The parameters Δ and ε, not present in the figure, are used to control the random number generator.

EfficientDet D0 [15] is the detector used to generate the predictions. It is a modern CNN architecture for object detection, with activations and weights quantized to 16 bits using scale quantization [74]. The model is retrained for 20 epochs using the hyperparameters detailed in [15] to recover from the accuracy loss induced by the quantization. The quantized model has the same accuracy metrics as the original one. The fault-free CNN is then used to generate the baseline detections set for each image of each sequence of MOT17DET [34], which is a dataset of street scenes with images captured with stationary and moving cameras mounted on the car's roof or hood. The baseline detection set is used to evaluate the accuracy loss of the CNN executed with computation errors with and without the support of ERODE, noted as faulty CNN and ERODE, respectively, in the top and bottom of Figure 4.6. To evaluate the error recovery capabilities of ERODE, the simulation setup includes a scenario where errors occur not on each image but with a frequency defined as image error rate Δ. The image error rate used in the experiment is set to $\Delta \in [2,3,4,5]$, which allows testing the effectiveness of the keep-alive register in filtering wrong detections occurring at different rates. For instance, a Δ equal to two means that one every two images will be processed with faults injected during the inference. Moreover, the activations error rate ε is the percentage of the total computation volume subjected to bit flips in each frame. The experiments use $\varepsilon \in [0.001\%, 0.003\%, 0.005\%]$. The selection of which activations are subjected to bit-flips and which bits are flipped is made by sampling random integer numbers from a uniform distribution, meaning that each activation within the tensor and each bit within the activation have an equal probability of being selected. Only one bit can be flipped for each activation. As demonstrated in [7], bit-flips in the activations induce a higher accuracy degradation than bit-flips in the weights. Therefore, only faults targeting the activations are analyzed in the experiments. The number of errors generated for each combination of Δ and ε is reported in Table 4.1.

Table 4.1 Errors injected during the inference with different rates.

	$\Delta = 2$	$\Delta = 3$	$\Delta = 4$	$\Delta = 5$
$\varepsilon = \mathbf{0.001\%}$	11538852	7679515	5765075	4599007
$\varepsilon = \mathbf{0.003\%}$	35107176	23365070	17540350	13992566
$\varepsilon = \mathbf{0.005\%}$	58712628	39075335	29334175	23400923

Faults are uniformly distributed over each convolutional layer of EfficientDet D0, assuming that computation errors occur during the entire inference. The metrics used to evaluate ERODE's effectiveness are: true positive detections with an IoU that is at least 50% with the baseline, false positive detections that are not present in the baseline, and false negative detections that are only present in the baseline. Precision and recall are evaluated as in Equation (4.1).

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN} \tag{4.1}$$

The keep-alive register has three user-defined parameters that must be set. For these experiments, the following values are used: *keep-alive threshold*=10, *keep number threshold*=5, *initial number threshold*=1. The precision variation with different Δ and ε is depicted in the top of Figure 4.6. First, it is possible to notice how EfficientDet D0 is very susceptible to computation errors, as a ε=0.003% is high enough to induce task accuracy loss of 40% compared to the baseline for Δ=2 and ε=0.005% further drops the accuracy to 70%. When Δ is increased, inference errors occur less frequently, and the overall precision loss is reduced. Similarly, the same behavior can be seen for the recall results in the bottom of Figure 4.6. The discrepancy between the precision and accuracy loss could mean that the CNN tends to see non-existing objects rather than not detecting existing ones. An inspection of a subset of output predictions highlighted the presence of multiple wrong objects, with a small variation of the bounding box coordinates of correct objects. When the CNN is supported with ERODE, the accuracy and recall loss is significantly reduced. By leveraging the keep-alive register, it is possible

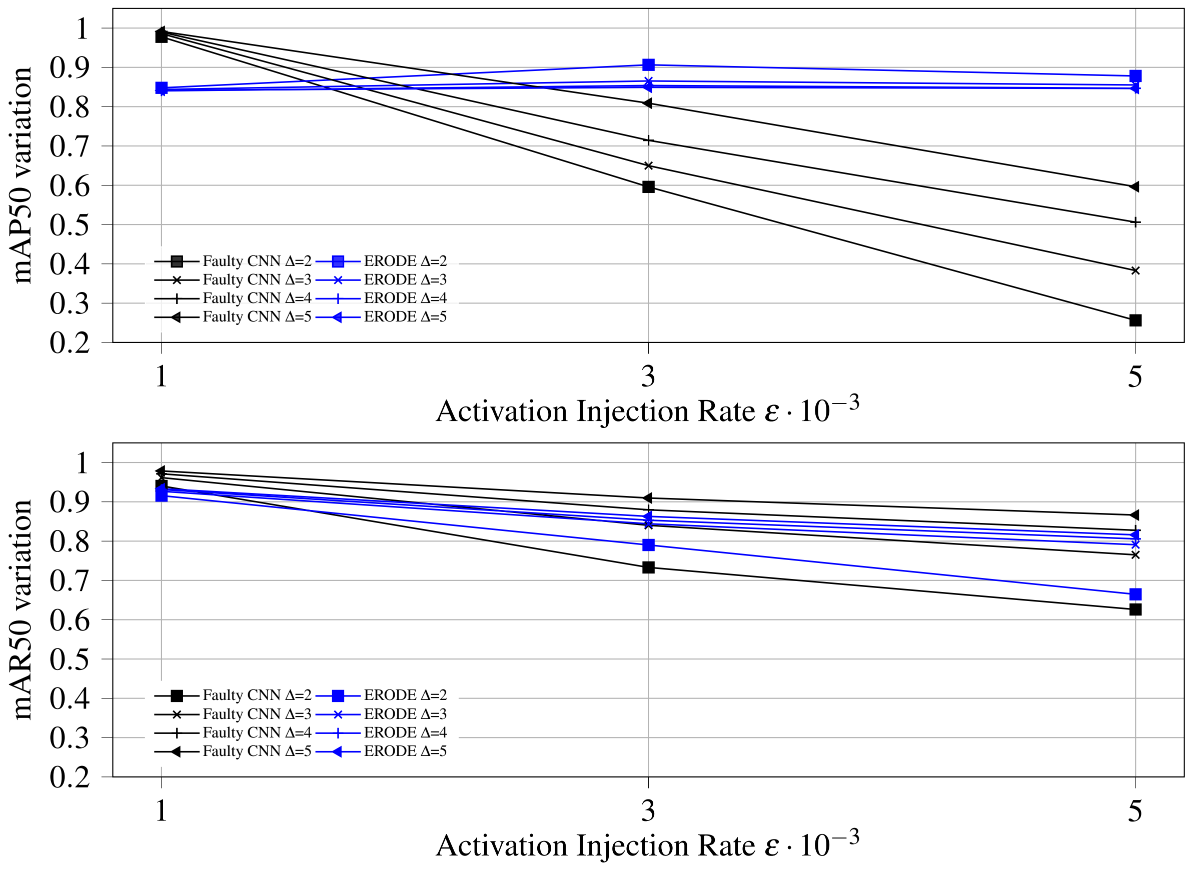


Fig. 4.6 Precision (top) and recall variation (bottom) with different Δ and ε.

to filter false positives, using the *initial number threshold* to eliminate detections that occur only in the Δ images due to computation errors. Moreover, missed detections are restored with the predictor after the object is not detected for a number of frames higher than the *keep number threshold*. ERODE's precision degradation could be due to the predictor's bounding box estimation, which might not overlap precisely with the ones generated with the baseline CNN, and to objects no longer present in the frame that persists for a few time instants in the keep-alive register. Moreover, the *initial number threshold* could also induce the removal from the register of correct objects that are not detected continuously in all images, reducing the recall by increasing false negatives. This effect can be mitigated by improving the accuracy of the CNN so that it detects objects more consistently or by decreasing the *initial number threshold*. Using an ensemble of differently trained CNNs with the same model architecture can also increase the baseline accuracy. Given the above, ERODE can be effectively used for error-resilient object detection, as it can filter out wrong detections and recover information lost due to computation faults. Finally, the computational complexity is reported in Table 4.2, estimated as the average number of multiplications and sums required to execute ERODE for each image. The complexity increases with the error rate as more glitches are generated in the CNN's output, resulting in more bounding boxes evaluated by ERODE. The glitches increase the number of entries in the keep-alive register that must be checked and compared against pre-existing objects with the tracker, also increasing the cost matrices built for each entry, while missed detections must be restored by using the predictor to move and resize the bounding boxes. The computation overhead of Table 4.2 is negligible compared to the number of MACs required to compute EfficientDet D0, which is $\approx 2.5 \cdot 10^9$ [15].

Table 4.2 Computational complexity averaged over the entire MOT17DET dataset.

		$\Delta = 2$	$\Delta = 3$	$\Delta = 4$	$\Delta = 5$
$\varepsilon = 0.001\%$	**Products**	48	49	49	49
	Sums	2565	2599	2611	2619
$\varepsilon = 0.003\%$	**Products**	34	43	47	48
	Sums	3394	3064	3519	3267
$\varepsilon = 0.005\%$	**Products**	97	82	79	64
	Sums	14963	9166	8984	5915

Discussion

In conclusion, ERODE is an effective and easy-to-integrate CNN plug-in that can mitigate accuracy and precision degradation in the presence of computation errors. Future work should include weight faults and explore computation errors induced by adversarial attacks. Unfortunately, adversarial samples are highly effective against CNNs for object detections and there are multiple types of attack that can invalidate entirely the output in multiple frames [157]. Therefore, it is advisable to search for a joint CNN/software solution in which the robustness of the CNN is increased while ERODE is adapted to analyze other data apart from the input images and output prediction to identify unexpected changes within the intermediate activations generated during the inference.

4.2 Hardware/NN Co-design for Reconfigurable Approximate Edge Inference

The superior task accuracy of DNNs comes at the cost of high computational complexity and memory requirements, thereby presenting challenges in deploying energy-constrained devices, such as MCU [62, 63]. To tackle the problem of efficient inference at the edge, introduced in Section 2.2, dedicated DNN hardware accelerators have been developed to optimize energy efficiency and throughput by reducing the data movement and the cost of arithmetic operations [1, 50–53, 158]. As detailed in Section 2.3, a considerable research effort has been put into reducing the size and precision of DNN models, developing pruning and quantization strategies. In particular, layer-wise quantization leverages the different degrees of robustness and tolerance to error introduction of DNN layers [1, 3, 105, 159] Similarly to pruning and quantization, approximation is another optimization technique mainly aimed at reducing the arithmetic energy [81]. The basic idea is to reduce the computational complexity and cost using operators that produce inexact results and have a lower switching activity or gate count. However, as the sensitivity of single layers inside the same model and among different models is highly variable, designing a multiplier with different approximation levels is fundamental to ensuring flexibility and adaptability. Several strategies have been explored in the literature to design hardware supporting layer-wise approximate DNNs [85, 116, 119], leveraging retraining or parameters fine-tuning to reduce the accuracy degradation. However, previous works rely on dedicated accelerators and support very few approximation levels [119, 160], limiting the possibility of finding the optimal error-accuracy trade-off, due to a smaller design space; other works use several non-reconfigurable multipliers instances in the accelerator's systolic array [85, 115, 118]. The latter approach increases the area overhead and prevents the same hardware from executing DNNs with different parameters or model architectures with the same energy efficiency or accuracy. Another problem is that such hardware accelerators are seldom implemented in reality and often are just high-level models simulated with tools such as [106–109], adapted to include the cost of approximate operations. Since IoT devices have limited area and power budgets [62, 64] and must be able to adapt to different workloads and performance targets, it is necessary to adopt a layer-wise approximation strategy that relies on a single multiplier offering several accuracy levels. Moreover, the hardware platform using this multiplier should be programmable and reconfigurable to support the deployment of different DNNs, reusing the same hardware as efficiently as possible without requiring a redesign or recall of the IoT device. Such flexibility allows choosing which DNN is suited for a particular scenario, prioritizing the battery life of the edge device or the accuracy of the predictions.

To achieve low-cost reconfigurable AxC, three components are necessary: a reconfigurable hardware platform with a compiler that can find an optimal loop scheduling, an AxM that can be reconfigured runtime, and finally, a hardware-aware design space exploration strategy that selects the best trade-off between optimal energy consumption and DNN accuracy. These three components are wrapped up in a single framework named **MARLIN** and are presented in Section 4.2.2. The PULP MCU [161] is selected as the target IoT platform with RI5CY [162], a RISC-V core. The RISC-V core is modified to host a runtime reconfigurable mixed-precision AxM with 256 accuracy levels that supports different quantization levels. Thanks to a custom RISC-V toolchain that is approximation-aware, the multiplier can be controlled via instructions embedded seamlessly in the code at compile time. Regarding the design exploration, a layer-wise approximation strategy based on NSGA-II is developed. It is applicable to any DNN topology with convolutional and fully connected layers, with no modification to the model architecture. The search objective is to reduce the energy of arithmetic operations while retaining the original task accuracy. During the optimization process, approximate DNNs are evaluated by their error resilience and energy, assigning a different multiplier configuration to each layer. The hardware platform composed of the RISC-V core and the AxM, executing approximate DNNs found with the above strategy, has been synthesized and simulated to prove the effectiveness and feasibility of this methodology.

4.2.1 Hardware Accelerators and Mapping of Approximate DNNs

DNN accelerators are typically based on a hardware architecture that comprises a memory hierarchy and an array of interconnected processing engines (PEs) [1, 50–52, 107], similar to the one depicted in Figure 2.14. The memory hierarchy usually comprises a main memory (off-chip DRAM), global buffers (on-chip SRAM), and the registers within the processing engines. The off-chip and on-chip memories are connected through the system's bus, whereas the PEs communicate through a network-on-chip. The execution of a DNN is scheduled with a dedicated mapper that generates the instructions and partitions the resources, minimizing the energy and latency associated with the data movement, as detailed in Section 2.4. The complete cost of a multiplication considered by a mapper includes the energy and latency required to read all the operands, move them through the memory hierarchy, compute and store the result [52, 107, 163]. The mappers used in [50–52] can be compared to the compilers used to generate machine code for processors such as RI5CY[162], whose purpose is to optimize the performance and resource usage, leveraging loop blocking techniques such as temporal and spatial tiling (also known as folding, unrolling, interleaving, etc.). The approximation methodology described in Section 4.2.2 is orthogonal to the mapping process as it only modifies the energy associated with the arithmetic operations and not the data movement. In [163], a tool that predicts the energy of approximate DNNs with a mapper based on [107] is presented, validating the assumption that AxC only modifies the arithmetic energy without affecting the data movement. Therefore, MARLIN does not influence the mapping and could be easily integrated into [50–52], similarly to what has been done for DORY in Section 4.2.2. Alternative hardware architectures for energy-efficient DNN inference at the edge are MCUs such as [53, 158], with Instruction Set Architecture (ISA) extensions and dedicated DPUs inserted in the RISC-V pipeline. In this work, PULP [158] is selected as the target platform, a MCU-based low-power computing platform with a host CPU relying on the RI5CY or zero-riscy cores and equipped with a multi-core programmable cluster. The motivation for using PULP is twofold: the RTL and the toolchain are open-source and well-documented [161], and, being a MCU-based platform, PULP truly represents a low-power IoT device with strict resource constraints [62]. Additionally, PULP is selected to produce a prototype that can be shared and adapted without limiting the compatibility of this methodology to the PULP platform, as highlighted by the fact that the layer-wise approximation strategy and the deployment process detailed in Section 4.2.2 do not have any hardware dependencies except for any reconfigurable AxM. The PULP project includes libraries and tools to easily export a DNN model written in PyTorch to C code compatible with the MCU. The MARLIN framework includes modified versions of the RI5CY core and software tools, adapted to include the approximation level selection through Control Status Register (CSR) instructions, whose generation and compilation are added to the original PULP toolchain. The porting of approximate hardware on a RISC-V-based platform has also been done in [160]. A limitation of [160] is that the configuration signals, handled by a control unit external to the core, are generated by the user, thus relying on his expertise rather than on an automated mechanism. Another problem with this approach is that the external action is code-dependent, as only some instructions can be executed with inexact operators. As explained in Section 4.2.2, these limitations are overcome by embedding the runtime approximation control in the instructions processed by the RISC-V core, leveraging the flexibility of the PULP platform. Another argument in favor of using a single multiplier architecture, according to [164] and [165], is that the resilience of a DNN model to adversarial attacks depends on the AxM adopted. Consequently, using different configurations, including an exact one, is suggested to achieve higher error tolerance in various scenarios. A single reconfigurable multiplier architecture such as TEMET [8], applied to [165], would remove the constraint of using a single fixed AxM, enabling error compensation, used in [85, 118, 119] to improve the accuracy. Moreover, it can eliminate the limitations of [164], in which 13 different multipliers are used within each PE to support 13 approximation levels, enabling 21.3 times more approximation levels with a thirteenth of the multipliers instances.

4.2.2 Proposed Methodology

Overview

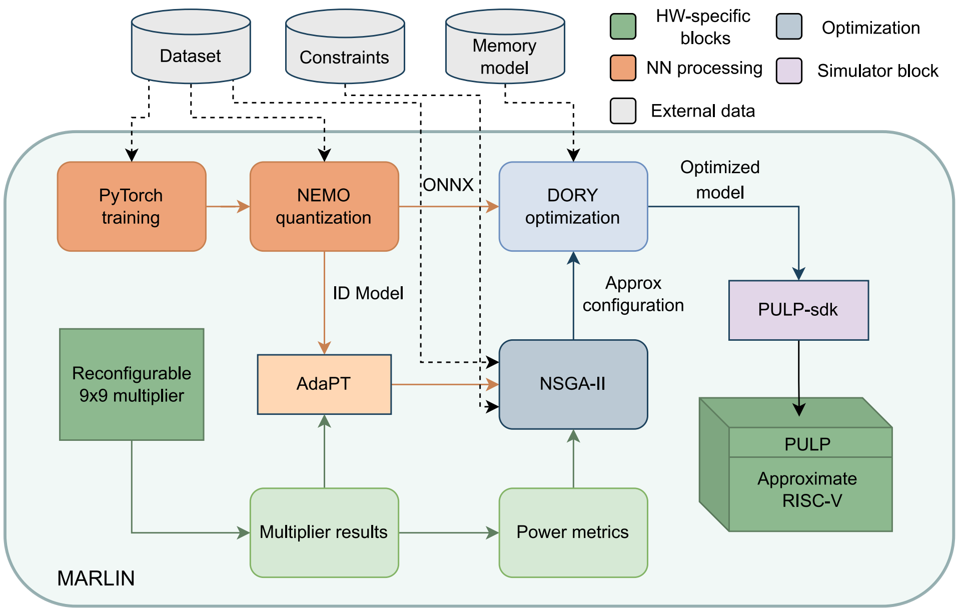


Fig. 4.7 MARLIN framework with hardware support specific for PULP SoC.

The MARLIN framework depicted in Figure 4.7 comprises different building blocks interacting with one another to determine a flow covering from the model definition up to its hardware deployment. The first external input required is a valid dataset, such as MNIST or CIFAR-10, which defines the target task of the DNN model. Given a specific application, there are often constraints on the minimum acceptable accuracy or the maximum tolerable energy consumption. Once the training dataset is provided with the specifications, a suitable DNN model is identified and described with PyTorch [29]. During the definition of the DNN model architecture and the training process, hyperparameter tuning is crucial to obtain consistent results and a model that is already robust to numerical errors, leveraging quantization-aware training or post-training quantization. This phase implies choosing the number of training epochs, the type of optimizer, the batch size, and other learning parameters. A standardized representation of the DNN in the form of a data flow graph is required to port the model to PULP. For this reason, the trained DNN is passed to NEMO [166], which transforms a floating-point model to an integer one in ONNX format. The precision of the model is fixed at this point in the procedure, with the added constraint that the bit-width of weights and activations cannot be above eight bits, either signed or unsigned. Up to this point of the procedure, the model has no knowledge of the approximation. On the hardware side, a reconfigurable AxM is designed and instantiated in MARLIN with a 9x9-bit parallelism. The constraint on 9x9 bits is required by the PULP toolchain with NEMO and DORY [167], which is detailed later on. As MARLIN's optimization software only requires, for each configurable approximation level, a LUT storing all the possible results for each couple of input operands and the average power to execute a single multiplication, the multiplier block is interchangeable. A single runtime reconfigurable approximate unit or several multipliers with fixed approximation levels could be integrated into the framework with little or no modification. Once this high-level description of the computational unit is available, the approximate model can be implemented and tested through the AdaPT library [111]. Any DNN topology built with PyTorch's convolutional and fully connected layers can be easily included in MARLIN by changing the layer instances with the AdaPT ones without modifying the model architecture or retraining. MARLIN solves the complex multi-objective problem of assigning an optimal approximation level to each layer through NSGA-II, performing multiple simulations of the model with the selected configurations to evaluate the accuracy and power consumption. The Pareto front evaluated in this way will show different optimal trade-offs between accuracy and power, corresponding to the two fitness functions NSGA-II tries to optimize. The last step that MARLIN performs on the software side is the generation of the C code to execute the model on the target hardware, using a modified version of DORY [167]. This is the first part of MARLIN which requires knowledge of the actual hardware architecture, including a detailed high-level description of every memory level size and latency to perform memory tiling effectively. For this work, PULP was selected as the target platform among those supported. DORY receives as an input the ONNX model generated by NEMO and an additional node-by-node dictionary of the DNN containing information of the approximation of each layer retrieved by NSGA-II. The modified DORY tool generates the C code for the provided approximate architecture with the received configuration. For this purpose, DORY is made aware of the modifications the PULP platform undergoes. An approximate unit is added to the execution stage of the cluster cores to approximate all the relevant instructions in the computation of convolutional layers. It is based on the same reconfigurable multiplier of [8], whose LUTs are used by NSGA-II and AdaPT. This unit is managed by a dedicated CSR who is in charge of activating, deactivating, and configuring its approximation level. The final code runs on the PULP platform through PULP-SDK and can be executed by providing input data read from the external L3 memory while the weights are stored in L2 or L3 memory. The support of a real hardware RTL on which the model can run is crucial for validating the proposed co-design methodology. This allows accurate estimations of the metrics of interest on a complex system.

Genetic Search of Optimal Inter-Layer Approximation

Algorithm 2 Approximation level selection with NSGA-II

```
1:  ▷ $M$ is the quantized exact model
2:  ▷ $axx\_mult$ is the AxM
3:  ▷ $Ng$ is the number of generations
4:  ▷ $Np$ is the population size
5:  ▷ $Pc$ is the crossover probability
6:  ▷ $Pm$ is the mutation probability
7:  ▷ $\vartheta$ is the chromosome of $L$ elements, in range $[0,A]$, storing the mult. configuration
8:  ▷ $f_1(\vartheta)$,$f_2(\vartheta)$ are the fitness functions to optimize
9:  ▷ $P_n$ is the population at iteration $n$, with size $Np$
10: ▷ **Initialization**
11: $L \leftarrow count(M.Conv)$                         ▷ Number of Conv. layers in the model
12: $A \leftarrow A_0$                                   ▷ Number of approximation levels for multiplier
13: $P \leftarrow P_0$                                   ▷ Initial population vector randomly set
14: $f(P_0) \leftarrow (f_1(P_0), f_2(P_0))$             ▷ Initial fitness evaluation
15: ▷ **Execution**
16: **for** $(n = 0; n < Ng; n++)$ **do**
17:     $Q_n \leftarrow \text{Tournament}(P_n, Pc, Pm)$
18:     $retrain\_models(M, axx\_mult, Q_n)$
19:     $f_1(Q_n) \leftarrow 1/accuracy(M, axx\_mult, Q_n)$
20:     $f_2(Q_n) \leftarrow energy(M, axx\_mult, Q_n)$
21:     $f \leftarrow (f_1, f_2)$
22:     $R_n \leftarrow P_n + Q_n$                       ▷ Total population, size $2Np$
23:     **for each** $\vartheta$ in $R_n$ **do**
24:         $Rank(\vartheta)$
25:         $F_i \leftarrow F_i \cup \vartheta$          ▷ $F_i$ are the fronts
26:     **for each** $\vartheta$ in $R_n$ **do**
27:         **for each** $\phi_k$ in $f$ **do**
28:             $dis_\vartheta \leftarrow dis_\vartheta + Crowding\_distance(\vartheta, \phi_k)$
29:     Order $R_n$ based on fronts and crowding distance
30:     $P_{n+1} \leftarrow$ best $Np$ solutions in $R_n$    ▷ Update iteration counter
31: **return** $\theta_{best}$                           ▷ Optimum Pareto front is returned
```

The NSGA-II algorithm is used to solve the multi-objective problem of finding DNN configurations with different trade-offs between energy and accuracy. NSGA-II is a multi-objective genetic algorithm that evolves a population of solutions using non-dominated sorting and crowding distance assignment to classify and rank individuals based on their dominance and diversity. Crossover, i.e., recombination of different chromosomes, and mutation, i.e., variation of a gene, are applied to create offspring solutions, which are then integrated with the parent population. The selection process favors solutions from less crowded Pareto fronts and those with higher crowding distances, promoting the front exploration and providing a diverse set of non-dominated solutions [143]. The motivations for choosing NSGA-II are its proven effectiveness in multi-objective optimization and relative ease of implementation and tuning compared to other alternatives such as reinforcement learning or Bayesian optimization. NSGA-II generates a set of optimized approximate DNN configurations and selects for each one which approximation level is more suitable for each convolutional layer. Algorithm 2 details the NSGA-II search flow. Each chromosome has a dimension L, which is the number of layers composing the DNN. The alleles of each gene are encoded as an integer number between 0 (exact level) and A, which is the number of approximation levels supported by the multiplier. Single-point crossover is used to combine chromosomes while maintaining inter-layer dependencies between approximate configurations, a strategy used in [118] to reduce the effect of computation errors on the DNN accuracy without retraining. At the beginning of each iteration, Np approximate DNNs are retrained with 10% of the training split (0.1 epoch). Then, the accuracy is evaluated with the validation dataset. Contrary to previous works [85, 119], the accuracy of candidate inexact DNNs is not evaluated immediately but after a quick retraining with a fraction of an epoch. By leveraging partial retraining and validation, each DNN configuration is evaluated by its resilience to computation errors and the retraining effort required to recover from such errors. Solutions with faster recovery will have higher validation accuracy than others that are less fit, using the same number of training samples, and therefore have an evolutionary advantage. Therefore, retraining (or fine-tuning) is used to compensate for the error and enhance the design space exploration. For what concerns the fitness evaluation in Algorithm 2, the inference energy is estimated as in [85, 119] by multiplying the number of multiplications of each layer of the model M with the average energy of the AxM when set to the approximation level defined by the corresponding gene of chromosome ϑ. After evaluating the two fitness functions, the algorithm continues with mutation, crossover, tournament selection, ranking, and finally, the evaluation of the crowding distance and the generation of the new front. The cycle starts anew until all the Ng generations have been evaluated.

Reconfigurable Approximate Multiplier

MARLIN uses a single-cycle AxM architecture to introduce minimum modifications in the control flow of the RI5CY core. The complete architecture is not discussed in detail in this section, as it can be found at [8, 9]. The parallel multiplier is based on a Dadda reduction tree [82], with a modified Baugh-Wooley algorithm [168]. A variation of the truncation mechanism proposed in [169] is used to handle the dynamic setting of approximation and precision, allowing easy support of approximate and exact configurations for full and reduced bit-width of the operands. Truncation relies on a masking signal to select specific columns of the partial product matrix to fix at zero to reduce the switching activity, hence dynamic power, of the logic gates in that section of the matrix at the expense of an incorrect output. An externally configurable masking signal, noted as a, manages the selection of the approximation level, as shown in Figure 4.8 for the case of 9-bit inputs and a on eight bits. To enable power saving with mixed-precision and sub-8bit operands, a precision masking signal p is introduced to perform data-gating on the partial product matrix using a mechanism similar to the approximation and can be configured according to the precision of the expected result. The minimum supported bit-width for the input operands is fixed to two. The precision mask has the length of the output minus four. Figure 4.8 shows the precision signal on fourteen bits p_j covering the most significant part of the partial products. When a precision mask bit is set to zero, the corresponding column of the matrix is entirely zeroed.

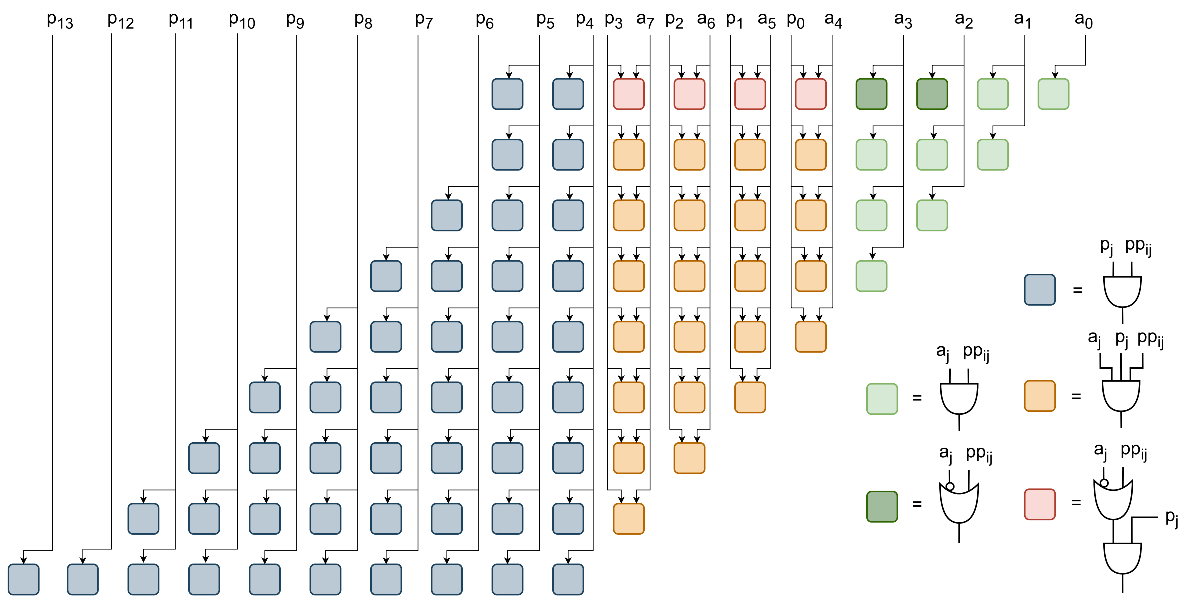


Fig. 4.8 Precision and approximation configuration management for the proposed multiplier. The approximation level is selected with the mask a, while the precision is selected with the mask p. a_j indicates the j^{th} bit of the `approx_mask` signal a, p_j the j^{th} bit of the `precision_mask` signal p, while pp_{ij} is the j^{th} bit of the i^{th} partial product evaluated according to modified Baugh-Wooley algorithm.

Reconfigurable Approximate RISC-V Core for PULP SoC

The RI5CY core architecture instantiated in the PULP cluster must be adapted to enable runtime reconfigurable approximation and precision. The first customization to the original pipeline, necessary to introduce inexact operators in the core, is exposing them in the ISA so that a programmer can effectively use them. A possible solution is proposed in [160]: adding custom instructions that, when decoded, configure the execution stage to use approximate operators. However, while trivial and easy to implement at low-level, this approach requires an additional instruction for each inexact arithmetic operation supported by the hardware, defined with a new custom format capable of encoding the approximation level. Another drawback is related to the fact that, in this specific case, the C executable code is generated automatically by the DORY tool. The latter is a specific problem due to the platform selected for the implementation in this work, but a similar issue can arise with other architectures such as NVDLA [55]. If custom instructions were used, an expert user would have to replace the standard instructions with the custom ones, whenever necessary, analyzing the generated C code line-by-line. The same could be achieved by making the compiler aware of the approximated instructions and where they are needed, but that would be time-consuming to implement and maintain. The methodology suggested in this work focuses on flexibility and simplicity by defining a new custom CSR handling all the control and configuration of approximate operators. This approach is scalable: a single 32-bit register can manage all the new operations and does not occupy additional instruction encoding space. It also enables reconfiguration, as part of the register bits control the approximation and precision level. Finally, it is much more programmer-friendly as it does not require significant changes in the C code of the microcontroller, except for the addition of CSR instructions. To achieve these results, the proposed methodology to enable online configuration of the AxM relies on the following steps:

1. A CSR write instruction sets the `precision_mask` and `approx_mask` fields according to the specification of the layer.

2. Before the computation starts, a CSR set instruction enables the AxM unit with the control signal `approx_mac` and the cluster of multipliers of the approximate DPU with the control signal `approx_dot8`, which are disabled when the computation is over.

Each CSR instruction takes one clock cycle to execute. In the general case, the last couple of CSR instructions are executed a number of times which depends on the tiler split performed by DORY. Being mapping-dependent, estimating the correct number of instructions or providing a meaningful value is difficult. Their position in the code is optimized to produce minimum overhead in the control flow, considering the presence of MAC instructions that must produce the correct result. The usage of three instructions, rather than two, is forced by the specific organization of the template C files provided by DORY and PULP-NN C library [162]. The CSR instruction executed more frequently is the one activating the approximate unit. It is located before the matmul function, whose pseudocode and number of assembly instructions

Algorithm 3 PULP-NN matmul function pseudocode

```
1:  ▷ $ch\_in/ch\_out$ input/output channels of the Conv layer
2:  ▷ $k\_x/k\_y$ filter dimension along x/y
3:  ▷ $im2col$ is $ch\_in \cdot k\_x \cdot k\_y$
4:  ▷ $col\_cnt\_im2col$ is $im2col$ & 0x3
5:  ▷ $chan\_left$ is $ch\_out$ & 0x3
6:  ▷ **Initialization**
7:   Load params. from stack and define variables                      ▷ 26 instr
8:  ▷ **Execution**
9:  **for** ($i = 0; i < ch\_out >> 2; i++$) **do**
10:     Setup                                          ▷ if first iteration 41 instr, else 6 instr
11:     **for** ($j = 0; j < im2col >> 2; j++$) **do**
12:         Setup                                                       ▷ 15 instr
13:         Multiply and accumulate + save             ▷ 15 instr $\cdot (im2col >> 2)$ + 9 instr
14:     **if** ($im2col >> 2 == 0$) **then**; Setup **end if**          ▷ 12 instr
15:     **while** ($col\_cnt\_im2col\,! = 0$) **do**
16:         Setup                                                       ▷ 4 instr
17:         Multiply and accumulate + save             ▷ 15 instr $\cdot (im2col >> 2)$ + 2 instr
18:     Quantize and save results                                       ▷ 54 instr
19: Setup                                                               ▷ 10 instr
20: **while** ($chan\_left$) **do**
21:     Setup                                          ▷ if first iteration 23 instr, else 1 instr
22:     **for** ($j = 0; j < im2col >> 2; j++$) **do**
23:         Setup                                                       ▷ 6 instr
24:         Multiply and accumulate + save             ▷ 6 instr $\cdot (im2col >> 2)$ +4 instr
25:     **if** ($im2col >> 2 == 0$) **then**; Setup **end if**          ▷ 6 instr
26:     **while** ($col\_cnt\_im2col\,! = 0$) **do**
27:         Setup                                                       ▷ 4 instr
28:         Multiply and accumulate + save             ▷ 6 instr $\cdot (im2col >> 2)$ + 1 instr
29:     Quantize and save results                                       ▷ 13 instr
30: Save parameters and return                                          ▷ 24 instr
```

are reported in Algorithm 3. In a worst-case estimation, a CSR set is performed once for each matmult function call, providing a quantitative measure of the reconfiguration overhead on the execution time. The GCC compiler is extended to account for the new `approx` CSR, whose fields are given in Figure 4.9. The Xpulp ISA extension [162] provides additional multiply-

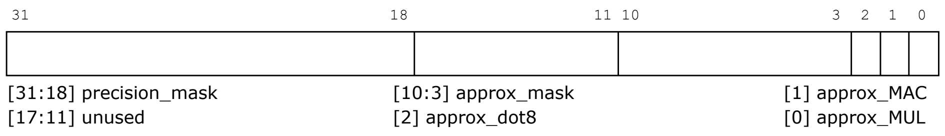


Fig. 4.9 The `approx` CSR configuration.

related instructions compared to the basic ones of RV32M. Some of these, such as MAC and SIMD dot products, are useful for data-intensive applications such as DNN inference. This work aims to provide an approximate implementation only for instructions used in convolutional and linear layers. To select them, an in-depth analysis of the disassembly code produced by custom convolutions and a complete DNN is performed. The assembly instructions included in these benchmarks are approximated, together with others for which it is straightforward to extend support, and are listed in Table 4.3. In this prototype, the approximate pipeline only computes 8-bit multiplications, even when the issued instruction expects a 32-bit or 16-bit operation. This restriction cannot cause any error assuming that DNN layers quantization is always on 8 bits or below, which is the ordinary case for DORY. The instructions for which approximate support is provided are split into three subgroups, according to Table 4.3. For each category, the `approx` CSR has a configuration bit; when this bit is set, all the instructions belonging to that group are executed in approximate mode. Besides the custom CSR, a unit responsible for inexact computation is inserted in the execution stage of the pipeline alongside the exact multiplier unit, as shown in the high-level block diagram of the approximate core in Figure 4.10. This design choice requires some trivial modifications in the decoding phase of

Table 4.3 Approximate instructions mnemonics.

MAC	MUL		DOT8	
p.mac	mul	p.mulsN	pv.dotup.b	pv.sdotup.b
p.macsN	p.muls	p.muluN	pv.dotusp.b	pv.sdotusp.b
p.macuN	p.mulu		pv.dotsp.b	pv.sdotsp.b

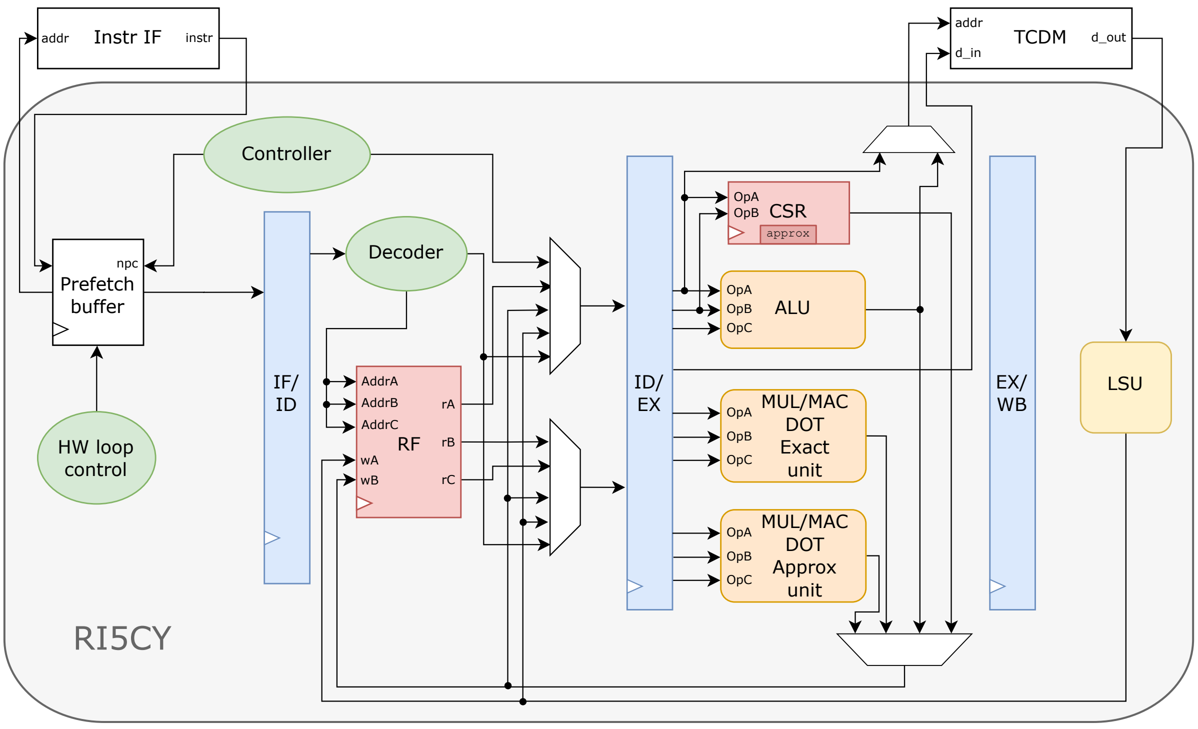


Fig. 4.10 RI5CY pipeline with approximate operations support.

the instruction. Based on some control signals, the decoder has to activate either the correct or the inexact unit. The arithmetic block that is not selected for the instruction currently in the decode stage does not perform any operation in the next clock cycle as its inputs are not updated, and therefore, it is data-gated. Four instances of the designed multiplier are allocated in the reconfigurable approximate unit, as in Figure 4.11. They are all used in parallel to perform SIMD dot products on 8 bits, while only one is activated for MUL and MAC-related operations.

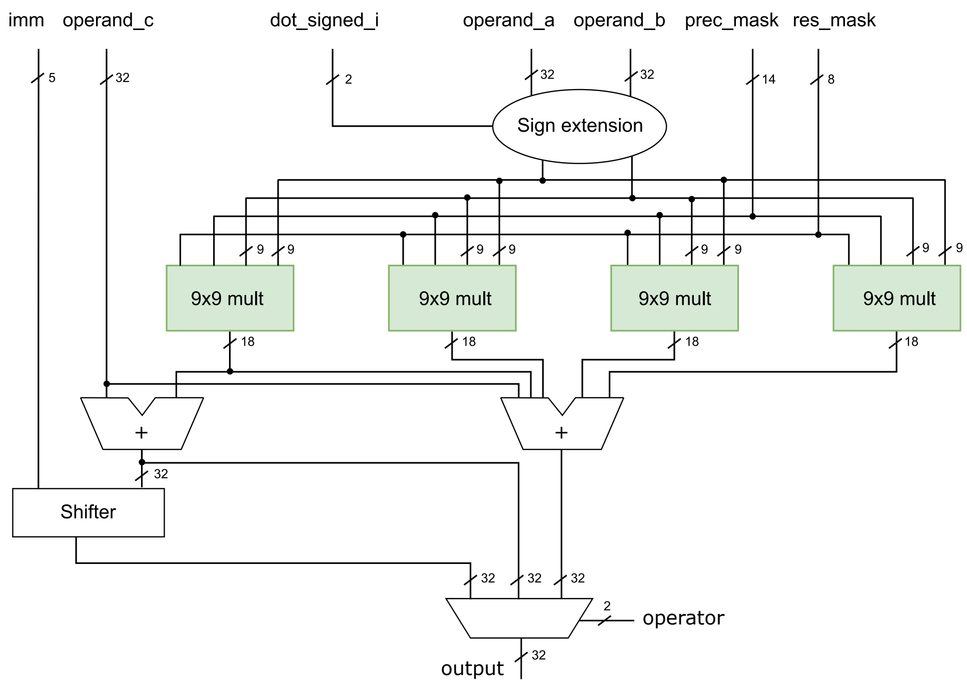


Fig. 4.11 RI5CY AxM unit. The leftmost multiplier is in charge of MUL and MAC-related operations. The shifter is for immediate instructions with the N suffix in Table 4.3. All multipliers work in parallel to execute dot8 operations and their result is fed to a 32-bit five-operands adder. The sign extension block handles the split of the 32-bit operands into four 8-bit chunks and their sign extension according to the decoded instruction.

Approximation in PULP Toolchain

Through NEMO and DORY libraries, the PULP platform offers software support to generate executable C code tailored to its architecture, starting from a PyTorch DNN model. These two parts of the toolchain must be adapted to enable the proposed runtime approximation methodology. NEMO is a PyTorch add-on framework developed as a support tool to transform an already trained, full-precision DNN into an integer one, performing the quantization and calibration of the model. The DNN can be quantized and calibrated before passing it to NEMO by using fake quantization floating point values during the training. DORY is an open-source tool for optimizing DNN mapping on PULP and other MCUs. Two building blocks of DORY, the configurable templates and PULP-NN back-end functions, are modified to automatically add the approximate CSR instructions in proper points of the C code, while the mapping optimization is unaffected. Besides an ONNX graph, the modified DORY uses as input a JSON file containing a layer-by-layer description of the quantization and approximation of the DNN. Once these parameters become part of the DORY intermediate representation, they are used to fill hardware-specific template files with the correct CSR setting and generate the C code for the different layers. Using a JSON dictionary guarantees flexibility, as new items can be defined for each node in the network. Furthermore, it is general; at this level, every type of AxM, whether reconfigurable or not, could be available. Moreover, as the precision information on the layer is kept separate from the approximation level, it is possible to configure the layer as exact but with reduced precision. This choice allows to save power by leveraging operations with reduced bit-width rather than with inexact computation.

4.2.3 Results

This section presents the computing setup used to conduct experiments to validate the proposed methodology, summarized in Figure 4.12. The modified RISCY is synthesized and tested with the reconfigurable multiplier and relevant hardware metrics of the core and the arithmetic operator alone are extracted and analyzed. Additionally, the layer-wise approximation strategy is compared against other state-of-the-art techniques.

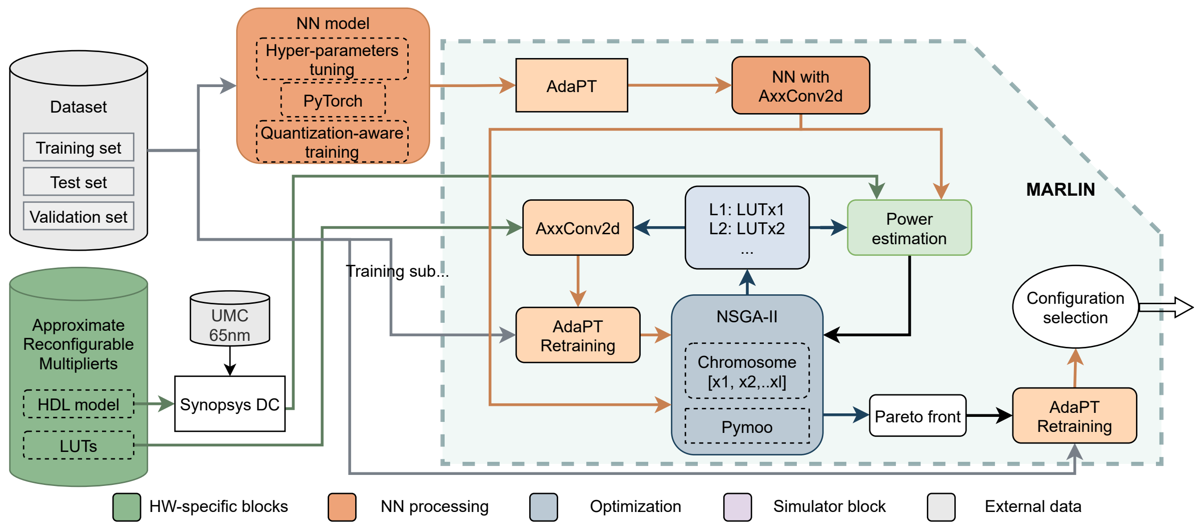


Fig. 4.12 MARLIN framework and computing setup for software simulation.

Approximate RISC-V Core Characterization

The RI5CY core featuring the approximate extension is synthesized to extract area, delay, and power estimations using Synopsys Design Compiler and the UMC 65 nm library [170]. The two cores with exact and reconfigurable approximate operators are both synthesized with clock-gating and without boundary optimizations, using the command:
`compile_ultra -no_autoungroup - no_boundary_optimization -timing -gate_clock.`.
Table 4.4 compares area and timing values obtained. The delay constraints are satisfied by

Table 4.4 Performance and area comparison for exact and approximate RI5CY and their relative multiplier units.

		Exact RI5CY	Approx RI5CY
Area [μm^2] (GE)	Exact mult	14842.8 (10k)	
	Approx mult	-	4737.2 (3k)
	Total	60621.1 (42k)	67006.8 (47k)
Timing [ns]	Exact mult	4.45	
	Approx mult	-	4.41

both designs, and from the fourth column of Table 4.4, it can be observed that the multiplier of [8] does not increase the microprocessor critical path, as it is still determined by the exact multiplier unit. The area overhead of the approximate unit alone is 7.8%, while the total overhead, considering the additional control logic part and the `approx` CSR, is 10.5%. The extra area cost is mainly due to the allocation of four reconfigurable multipliers in order to manage 8-bit dot products. The area overhead is definitely acceptable as this is the first prototype of this architecture and it could be further decreased if the four reconfigurable multipliers replaced the exact ones, which is possible since they also feature a non-approximate mode, that can be used to process also non NN-specific instructions. The latter would require no special modification or additional instructions inserted in the compiled code.
The RTL model of the RI5CY is replaced by the gate-level netlist for all cores in the PULP cluster to enable post-synthesis simulation and power estimation on a demonstrative use-case. A simple DNN model is designed with PyTorch and used as a benchmark on the MNIST dataset to verify the correct behavior of the entire framework and to collect power metrics. It comprises five convolutional layers (with bias), each followed by a ReLU activation function, and a final linear layer. The entire structure is depicted in Table 4.5. The model is trained for 30 epochs, with a batch size of 32, an initial learning rate of $3 \cdot 10^{-3}$, with a step factor of 0.3 every 5 epochs. Quantization-aware training, with scale quantization [74], is used to achieve the same accuracy as the full precision DNN, using 9-bit for both activations and weights. The 9-bit constraint is given by the DORY and NEMO frameworks; the other results presented

Table 4.5 Custom NN architecture description.

Layer name	Output size	Kernel size	Output channels	# Mult
conv1	28x28	7x7	3	115224
conv2	28x28	5x5	8	470400
max pool	14x14	3x3	8	0
conv3	14x14	3x3	10	141120
conv4	14x14	3x3	16	282240
max pool	7x7	3x3	16	0
conv5	7x7	3x3	24	169344
max pool	3x3	3x3	24	0
linear	1x10	9x24	1	2160

in this paper are for 8-bit quantization. Stochastic gradient descent with momentum 0.9 is used with a weight decay of 10^{-3}. This particular value for the weight decay was chosen to desensitize the DNN from the additional numerical error introduced by inexact multipliers, which adds up to the quantization error. The DNN is run on the PULP platform for a single input image, i.e., batch size of 1, ten times for each layer-wise approximate configuration. For this model, the overhead of the CSR instructions execution can be quantified, according to Algorithm 3, in the worst case, which is the first convolutional layer, as one CSR set instruction every 403 instruction (0.25%). In the best case, which is the last convolution, the extra cost is 0.026%. The performance overhead due to the CSR switching, and thus of reconfiguration, is negligible compared to the overall processing. Through Siemens QuestaSim, the VCD dumps of the entire core and the approximate and exact multiplying units are collected and used as inputs for Synopsys Power Shell to extract power metrics based on the actual switching activity. Simulations are performed using the multipliers configurations obtained from the NSGA-II run that achieve accuracy over 90% with no retraining. The retraining in this experiment is not important, as these measurements serve only to provide a quantitative analysis of the impact of the new multiplier architecture and DPU on the original performance and to validate part of the proposed methodology. For every configuration, a random input for each of the ten possible categories is fed to the network, meaning numbers from zero to nine for MNIST. The post-synthesis of the exact RI5CY core is simulated with the same inputs, and the power of the accurate multiplier unit is collected; all results are averaged. Table 4.6 reports, for the configurations listed in the first column, the DNN test accuracy and, in the third column, the power of the multiplier unit (the exact one for the exact configuration in the first row, the approximate one for all other cases) averaged over the ten simulations. The fourth column contains the relative energy saving of the multiplier unit measured post-synthesis, while the last column shows the relative energy saving estimated at the end of the NSGA-II search, which is carried out as described in Section 4.2.2, but with no retraining. The first row of

Table 4.6 contains the power estimation of the exact RI5CY multiplier unit, where all operators, including the multipliers computing dot8 instructions, are described behaviorally, thus leaving the architecture selection to Synopsys, with the DesignWare library. Consequently, for a meaningful comparison, the reference model for the NSGA-II relative estimation is the average energy consumed by a behavioral 9-bit multiplier synthesized by Synopsys Design Compiler, with the same settings as the AxM. Although the exact multiplier always executes housekeeping operations, these are a negligible fraction of its overall workload when all DNN multiplications are mapped on it; thus, their contribution to the average power is minimal. Therefore, since the number of multiplications of the DNN model is fixed, and so is the time the multiplier stays active, the relative energy is considered equivalent to the ratio between the average power of the approximate unit in the modified RI5CY and that of the exact one in the unmodified core.

Table 4.6 Power consumption and energy saving of the multiplying unit with different layer-wise configurations for the target DNN.

Configuration	Test Acc [%]	Average RI5CY mult power [μW]	Relative RI5CY mult energy saving	Relative NSGA-II energy saving
Exact	98.7	10.46	0%	0%
[0, 0, 0, 0, 0]	98.7	2.606	75%	42%
[59, 31, 15, 12, 3]	98.7	2.509	76%	46%
[59, 31, 15, 31, 31]	98.5	2.474	76%	48%
[255, 63, 15, 31, 11]	97.8	2.412	77%	50%
[255, 126, 3, 63, 63]	91.3	2.403	77%	53%

The fourth column of Table 4.6 shows that the obtained average energy saving on the multiplier unit is at least 75% when comparing the exact core and the approximate one with all multipliers configured as accurate (first and second row in Table 4.6). The maximum saving is 77%, which is more than 20% higher than the high-level estimation performed in NSGA-II. However, the advantage of adopting different approximate configurations is heavily reduced compared to the initial evaluation. This can be observed by rescaling the energy results in the fourth and fifth columns of Table 4.6 with respect to the exact configuration of the multiplier used in this work [8]. The highest estimated energy reduction, with respect to the model using the exact configuration of the reconfigurable multiplier for all layers, is 7.7%, with an accuracy loss of 7.4%, while the predicted saving was 20.1%. The cause of the gain drop, obtained by configuring the multiplier with a higher approximation, has to be addressed to the chosen task for the network. When the average power of the multiplier is estimated, 100000 random values with uniform distribution are used as inputs to the multiplier. However, the MNIST dataset is composed of black numbers on a vast white background, which means that the network inputs and, consequently, those of the multipliers are not uniformly distributed. Both input images and intermediate activations show a high concentration of zeros, contrary to the initial assumption. The main consequence of most values being zero is that data-gating, which is the primary source of power saving when approximation is applied, becomes ineffective as the operands are already zeros and have a low switching probability themselves. A higher sensitivity of the selected multiplier [8] to zero input operands, compared to the one from the Synopsys DesignWare library, could also explain the increased relative energy saving in the example with respect to the estimates used during the NSGA-II search, which based the power estimations on LUTs. Even acknowledging the difference in the statistics of the operands, the optimization algorithm is run using the estimated average energy computed with uniformly distributed inputs, keeping it data-agnostic, as commonly done in state-of-the-art optimization frameworks, where performance metrics are estimated independently from the statistics of the dataset [2, 3, 105, 107, 159]. The latter enables the proposed search strategy to generalize to new data and thus demonstrate the efficacy of the methodology, an approach that was also used in [85, 116, 119, 163] to estimate the energy reduction with AxMs. Moreover, these results also demonstrate an important but also foreseeable conclusion: AxC, while a feasible optimization technique, cannot significantly improve the energy and latency metrics of the whole system. Consequently, it should be used paired with other techniques, such as pruning or quantization, to effectively reduce the energy, or as a low-power defense technique against adversarial attacks [164, 165], or to desensitize the DNN model from hardware errors [6, 136].

Benchmark with CIFAR-10

Six variations of the ResNet model architecture [13] are used to experiment with shallow and deep DNNs for image classification, testing the effectiveness of MARLIN with CIFAR-10 [10], a more challenging dataset than MNIST. The implementation of the ResNet models is based on the original paper, using the same model architecture and hyper-parameters, but with 44k iterations instead of 64k, substituting the original multi-step scheduling of the learning rate with a cyclical scheduler [39], ranging between 10^{-1} and 10^{-4}. The reason for opting for a cyclical learning rate instead of a stationary one is to achieve faster convergence with fewer training iterations during the genetic search. Separate quantization-aware and full precision training are carried out for the INT8 and FP32 models. All the experiments with AxMs use the INT8 quantized models; the FP32 results are presented only to provide a comparison with INT8. Scale quantization with the straight-through-estimator is applied to the weights [74], while the activations are quantized using PACT [70]. Both techniques are applied to generate fake quantized data used to achieve quantization-aware training, to include the numerical error introduced with computation with reduced precision in the forward and backward pass of each iteration. The loss of an DNN executed in this way will comprehend the normal mis-prediction error and the quantization error, which are both compensated after one optimizer step.

Table 4.7 Neural Networks Used in the Experiments.

Neural network	# Conv. layers	# Mult.	FP32 accuracy	INT8 accuracy	Design space size
ResNet-8	7	12.2M	85.33%	85.43%	$72 \cdot 10^{15}$
ResNet-14	13	26.4M	90.17%	90.32%	$20 \cdot 10^{30}$
ResNet-20	19	40.6M	91.77%	91.50%	$57 \cdot 10^{44}$
ResNet-32	31	68.9M	92.65%	92.58%	$45 \cdot 10^{74}$
ResNet-50	49	111.3M	92.88%	92.60%	$10 \cdot 10^{117}$
Resnet-56	55	125.5M	93.14%	93.11%	$28 \cdot 10^{131}$

Table 4.7 reports the DNNs used in the experiments. The INT8 accuracy results are evaluated using the approximation level 0 of the proposed multiplier, which provides exact results, for all the layers of each network. The multiplications reported in Table 4.7 are evaluated for the inference of a single 32x32x3 input image from the CIFAR-10 dataset. The design space size reported in the rightmost column is evaluated as the number of unique approximate layer-wise configurations, evaluated as A_x^L, with A_x as the number of accuracy levels of the reconfigurable multiplier, in this case 256, and L the number of layers that can be approximated.

For ResNet-8, ResNet-14, and ResNet-20, the genetic search is run for 80 generations with a population of 70 individuals, whereas for ResNet-32, ResNet-50, and ResNet-56, the number of generations is increased to 120. Before the definitive experiments, a tuning of the agent is performed to select the population and generation values. No statistical analysis or quantitative evaluations are performed at this point. Then, a single seed is used for all the experiments presented in this work, to enable data-reproducibility. The results presented in this section are taken from a single run of the search algorithm, using the same seed for each random number generator in each experiment (every run for each experiment will generate the same results for the same generation). The mutation probability P_m and the crossover probability P_c are set to 0.8. During the search phase, every approximate DNN is retrained with 10% of the training set and the accuracy is evaluated with the validation set (5000 unseen images from the training set). The test set is not used during the search phase as it would have biased the results and negatively affected the genetic algorithm. In this phase, the DNN accuracy influences the evolution of each individual's configuration, i.e., the layer-wise approximation; therefore, to obtain DNN models that can generalize on new unseen data and prove the effectiveness of the proposed methodology, any correlation with the final test results had to be removed. Finally, once the Pareto front has been computed, the approximate DNNs are retrained for one full epoch and evaluated using the test set.The accuracy and energy results of only the dominant solution after the full retraining of the Pareto front are reported in Figure 4.13, providing a visual representation of the energy-accuracy trade-offs this methodology offers. For every DNN under test, the absolute accuracy difference between *Pareto valid.* and *Pareto test* marks of each configuration is almost always smaller than 1.5%, with even smaller values for deeper models. Therefore, the validation accuracy could represent a good approximation of the final results, justifying the choice of using it in the proposed search strategy.

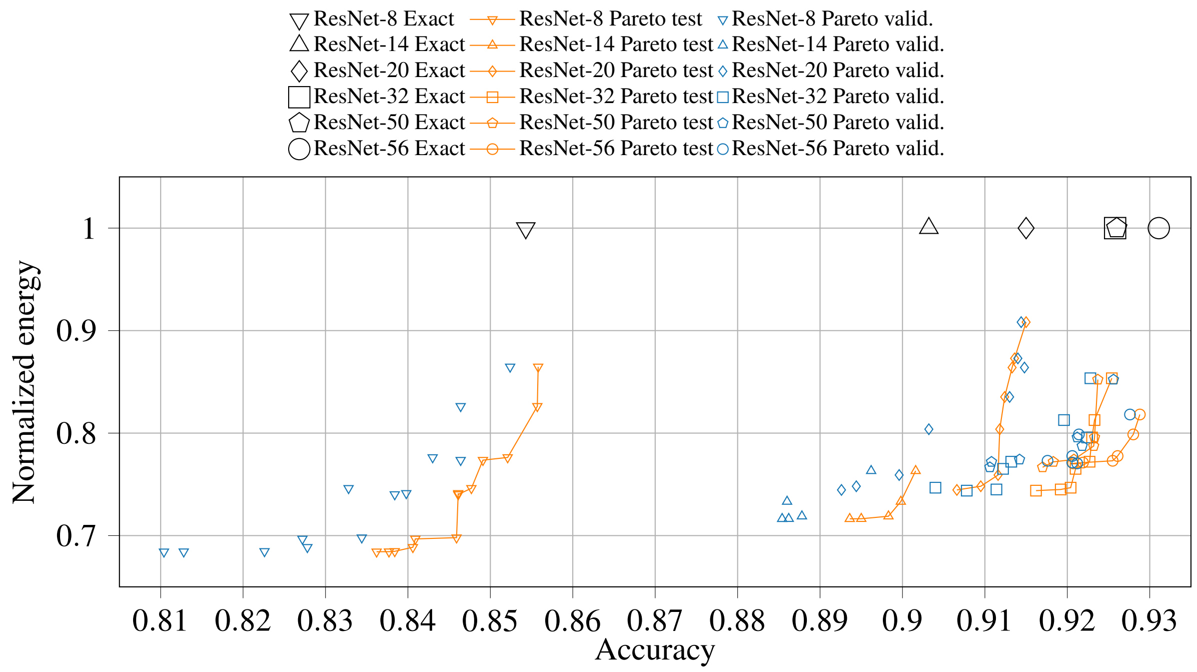


Fig. 4.13 Top 1% accuracy and normalized energy variation with different ResNet configurations. The *Pareto valid.* blue marks represent the validation accuracy evaluated during the genetic search, whereas the *Pareto test* orange marks represent the corresponding approximate configuration tested after the final retraining.

Figure 4.14 depicts the utilization of approximation levels for each DNN layer. It is possible to notice the presence of peaks around levels with an index equal to or smaller than $2^j - 1, j \in [0,1,2,3,4,5,6,7,8]$, corresponding to the Pareto optimal points of the energy/mean relative error distance (MRED) metric of the multiplier used in this work, which is taken from [8]. In Figure 4.14, it is shown how the majority of approximation levels are used in the Pareto front, justifying the choice to maintain power and MRED-dominated configurations as the most efficient levels might not be optimal ones to achieve a high task accuracy. Moreover, when concatenated appropriately, some approximation levels, even the Pareto-dominated ones, might mitigate the effect of computation errors on the final results, a strategy used in [118] to reduce the accuracy degradation. However, Figure 4.14 also highlights that some approximation levels are never used in this use case. Future development should add the possibility of pruning the search space, removing unused solutions or those with the lowest utilization. In these experiments, to prove that this methodology is effective with an ample search space, low- and zero-usage solutions are deliberately kept to test the search algorithm in a worst-case scenario with the highest complexity.

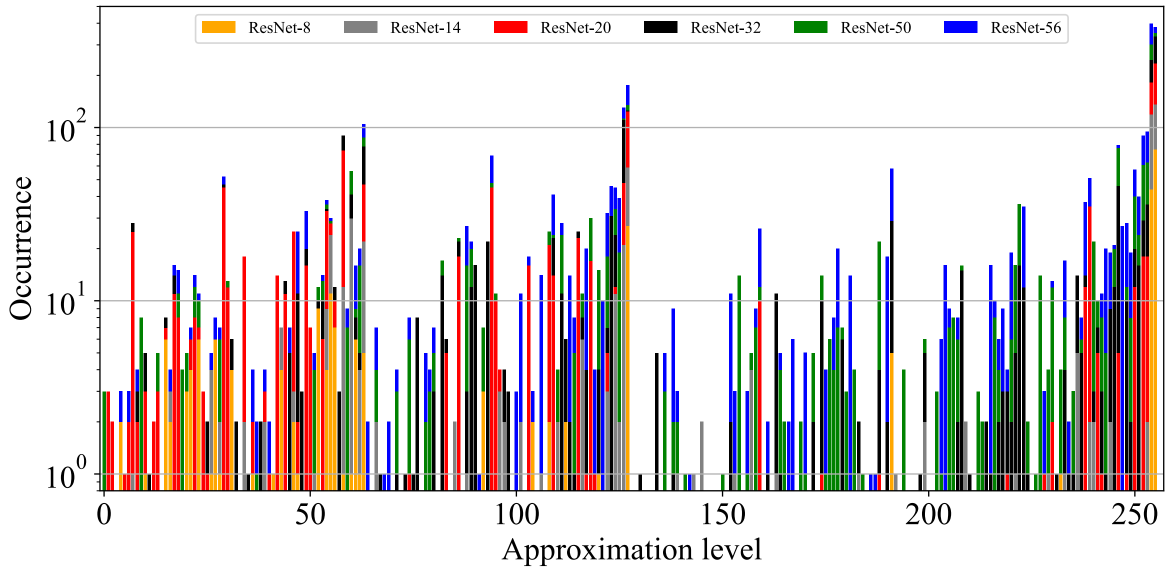


Fig. 4.14 Approximation levels utilization for all the configurations found for each ResNet model.

Table 4.8 Comparison with ALWANN with 0.5% and 1% relative accuracy degradation.

Neural network	This work			ALWANN		
	Absolute accuracy	Relative accuracy	Energy	Absolute accuracy	Relative accuracy	Energy
ResNet-8 0.5%	85.21%	99.74%	77.62%	83.16%	99.88%	84.31%
ResNet-8 1%	84.59%	99.02%	69.80%	Same solutions as ResNet-8 0.5%		
ResNet-14 0.5%	89.98%	99.63%	73.32%	85.42%	99.85%	74.34%
ResNet-14 1%	89.50%	99.09%	71.64%	84.77%	99.09%	70.85%
ResNet-50 0.5%	92.14%	99.50%	80.67%	89.08%	99.92%	78.47 %
ResNet-50 1%	91.70%	99.03%	76.67%	88.58%	99.36%	70.02 %

Table 4.8 compares the proposed approach with ALWANN [85] to understand how MARLIN stands against the state-of-the-art. To make a fair comparison, approximate DNNs with weights updated after a single epoch retraining for MARLIN and weight fine-tuning for ALWANN are chosen. For each approximate DNN configuration, in this work and ALWANN, the total energy is evaluated as the sum of the energy consumed by the AxM unit, as in Table 4.6, multiplied by the number of multiplications of each layer of the DNN model. The same methodology is applied for the comparison with [119]. This enables a fair comparison

of the arithmetic energy between methodologies with different architecture choices (MCUs vs. ASIC accelerators). The proposed method can achieve better results for shallower DNNs such as ResNet-8 and maintain the same relative gains for ResNet-14, whereas it was not able to achieve higher energy efficiency than ALWANN for ResNet-50. Comparing the absolute accuracy of the DNN models, the approximate ResNet-14 within 1% relative accuracy degradation outperforms all the ResNet-50 models presented by ALWANN in top-1 accuracy and, by extension, in energy efficiency, as ResNet-14 has 76.3% fewer multiplications than ResNet-50. The proposed methodology is competitive, considering that the multipliers used in ALWANN have better area and power-MRED metrics but are not reconfigurable [84, 85]. The main advantage of a reconfigurable multiplier against several arrays of fixed multipliers is that it is possible to improve the energy efficiency of arithmetic operations with a lower area. This approach, extended to a systolic array, would require a single array with the same multiplier architecture, whereas ALWANN requires N separate sub-arrays to support N approximation levels.

Table 4.9 compares MARLIN with the results presented in [119], without including absolute accuracy metrics, as they are not reported. Similarly to the previous comparison, it is decided to consider approximate DNNs with one-epoch retraining for MARLIN and approximate DNNs with weight fine-tuning with and without additional bias for [119]. Compared to the DNNs with no additional bias, the approximate DNNs configurations found with MARLIN require up to 13.1% less energy for ResNet-20, up to 13% less energy for ResNet-32, and up to 15.1% for ResNet-56. When an additional error correction bias is added to the convolutional layer in [119], MARLIN can still achieve up to 9,8% less energy for ResNet-20, 8.6% for ResNet-32, and 7,3% for ResNet-56, without increasing the number of parameters and operations. Using more approximation levels proved to be an effective way to further reduce the inference energy, as MARLIN had 256 configurations against the 3 used in [119]. It is possible to justify these results with the traditional weight-update strategy used in this work and the presence of more approximate configurations. Retraining each configuration is slower and more computationally expensive than the multiplier-specific fine-tuning of [119] but allows a better adjustment of the DNN parameters to compensate for the computation errors, resulting in higher accuracy.

Compatibility with Other Accelerator Architectures

Two important modifications are necessary to port MARLIN to other platforms: adapting the hardware architecture and the mapper to include the multiplier and the configuration instructions. The former would require an additional 8-bit control signal from the PE control unit to set the approximation level. The elongated critical path delay due to the approximation

Table 4.9 Normalized energy comparison with 0.5%, 1%, and 2% accuracy degradation.

Neural network	Normalized energy		
	Ours	[119] w/o bias	[119] with bias
ResNet-20 0.5%	75.91%	86.1%	83.1%
ResNet-20 1%	74.46%	85.6%	83.0%
ResNet-20 2%	74.46%	85.1%	82.5%
ResNet-32 0.5%	77.21%	85.7%	81.7%
ResNet-32 1%	74.50%	85.7%	81.7%
ResNet-32 2%	74.39%	85.5%	81.4%
ResNet-56 0.5%	79.87%	94.0%	83.0%
ResNet-56 1%	77.12%	86.1%	83.0%
ResNet-56 2%	77.04%	86.1%	83.0%

logic could be a problem for some accelerators, but it is not for [50, 52], which have a critical path compatible with the proposed multiplier. For what concerns the mapper, recalling the discussion of Section 4.2.1, a modification similar to what has been done in this work in the PULP toolchain can be implemented, inserting custom instructions to configure the multiplier, with negligible impact on the execution time. Since the scheduling does not change, the number of computation cycles would also be unaffected. Table 4.10 reports the area and the energy overheads of including and controlling the AxMs in three accelerators, mapping the 19 convolutional layers of the ResNet-20 with 1% accuracy degradation and 74.46% energy of Table 4.9. The power model of Timeloop [107] is adopted to evaluate the energy used to communicate to the PEs the approximation level of each layer, assuming one off-chip to on-chip memory transfer, and then *#PEs* transfers from the on-chip memory to the PEs' registers. The configuration energy of the entire DNN is always below 0.002% of the total energy evaluated with Timeloop. The area overhead of 35% against exact multipliers can be negligible, considering that they account for less than 10% of the PE area in [50, 52].

Table 4.10 MARLIN's area and communication overhead applied to other HW accelerators.

	Eyeriss [52]	DianNao [51]	Simba [50]
# PEs	256	256	1024
PE conf. comm. energy [pJ] (relative)	2138 0.001%	1997 0.001%	2369 0.002%
Mult. area exact [μm^2] (GE)	155468 (108k)	155468 (108k)	621875 (432k)
Mult. area approx. [μm^2] (GE) (relative)	210585 (146k) (+35%)	210585 (146k) (+35%)	842342 (586k) (+35%)

Discussion

The optimization approach of Section 4.2.2 allowed MARLIN to outperform previous works that relied on parameter fine-tuning [85, 119], leveraging partial retraining. MARLIN was run on a 32-thread Ryzen 5950X CPU with 64GB DDR4 DRAM and an Nvidia Quadro RTX A5000 GPU. The GPU was used only during the initial training of the FP32 and exact INT8 DNNs presented in Table 4.7, while the CPU was used to simulate the approximate convolutional layers during the training, validation, and test done during the search, as AdaPT only supports CPU computation [111]. The number of threads used during the computation was set to 16 for every experiment to compare how MARLIN execution time scales with different DNN depths. Table 4.11 reports the execution time for the search phase and the training of the last Pareto front of Figure 4.13. On average, partial retraining is ≈6x faster than full retraining. Compared to [85], the iteration time during the search phase is reduced by 72.8% for ResNet-8, 92.3% for ResNet-14, and 85.4% for ResNet-50. This speed-up is due to the increased utilization of CPU threads, as the training loop used in this work processes more images during each iteration compared to [85].

Table 4.11 MARLIN's average execution time with 16 threads.

Neural network	Search phase		Final training	
	One iter.	Total	One iter.	Total
ResNet-8	6.8 sec.	10.6 hours	40.9 sec.	24.6 min.
ResNet-14	7.7 sec.	12 hours	79.4 sec.	35.7 min.
ResNet-20	19.6 sec.	30.5 hours	115.7 sec.	90.6 min.
ResNet-32	30.3 sec.	70.7 hours	189.0 sec.	81.9 min.
ResNet-50	47.1 sec.	109.9 hours	296.2 sec.	69.1 min.
ResNet-56	56.9 sec.	132.8 hours	329.2 sec.	93.3 min.

A limitation in finding the optimal trade-off between energy and accuracy is the dimension of the search space, which determines the search time and requires a carefully tuned search strategy. This problem is also found in mixed precision layer-wise quantization, in which the search space is q^{2L}, with q quantization levels for weights and activations, for L layers [105, 3]. Future work should focus on pruning the search space after a number of experiments (i.e., NSGA-II generations) to reduce its size, possibly reducing the time to converge. A further improvement over layer-wise approximation can be proposed by looking at past works on quantization. In AutoQ [159], channel-wise quantization is used to reduce the inference energy with less accuracy degradation than [105, 3], proving that DNN resilience to quantization errors has an intra-layer dependency besides the inter-layer one. Therefore, achieving an optimal energy-accuracy trade-off is possible by extending AxC in the channel dimension. In

AutoQ [159], a bit-serial accelerator is required to support channel-wise quantization, whereas, with MARLIN, the only necessary modification to enable it with the proposed RISC-V core would be to adapt the CSR instructions inserted by the presented modified version of DORY. Nonetheless, the main challenge would be efficiently exploring a wider search space.

Chapter 5

Conclusion

Deploying DNNs on edge devices is still challenging due to the high computational complexity and memory requirements, but several joint optimization techniques presented in this doctoral thesis can improve performance, increase the system's battery life, and enable the execution of complex AI applications. In particular, a methodology for optimizing DNN models with compression and scheduling techniques to improve the performance of hardware accelerators was presented in Chapter 3. In Chapter 4, and in particular in Section 4.1, discussed the design and implementation of an algorithm for correcting errors in object detection, while Section 4.2 presents a framework for the deployment of approximate DNNs with runtime reconfiguration on MCUs. As the main contributions and discussion of future development for each of the presented works are already discussed in the respective chapters, this chapter discusses the general outlook on future development on robust, hardware-aware compression.

Hardware-aware Compression

At the time of writing, the reseach effort of the scientific community is shifting from "traditional" DNNs to transformers and large language models (LLMs), which are becoming increasingly popular in the field of natural language processing and computer vision. While there are successful quantization [171, 172] and pruning [173] strategies developed for vision transformers and LLMs, reducing the computing requirements from many to few TPUs or HP GPUs, there is still a performance barrier preventing a "democratization" of large models in research. Nonetheless, open-source pre-compressed models that can be executed on a single GPU with 8GB or VRAM are available and can be used as a starting point for further optimization [174]. Given the limited resources of edge devices, aggressive sub-8bit quantization and online pruning could be leveraged to deploy LLMs without incurring in heavy task accuracy loss, while preserving the adaptability of the model to different inputs. While the techniques presented in Section 2.3 can be extended to these models, it is also necessary to optimize the search strategy. As highlighted in Section 3.1 and in Section 4.2, the search space for optimal pruning, quantization, and approximation policies is very large and depends on the model's size. The number of parameters for the smallest Llama3 model is 4 billion, which is 4705x the number of parameters of ResNet56, the largest model used in the experiments of Section 4.2.3, which had a design space size of $28 \cdot 10^{131}$, without factoring in mixed-precision quantization, pruning, and optimal hardware mapping. In this context, the general direction for future development could be to remove time-consuming operations such as the fine-tuning of the model, simulation or testing of the task accuracy. Accuracy predictors such as the one used in [78] to estimate the compressed model's performance and improved hardware mapping and estimation tools could be used to reduce the iteration time. Efficient search strategies that iteratively reduce the search space, such as the ones proposed in [2, 3], could be adopted to further reduce the complexity.

Combining Compression and Error Resilience

Adversarial robustness and error resilience are two different aspects of the same problem: secure deployment of DNNs. Possible mitigation strategies have been proposed in Section 2.5, in Section 4.1, and in Section 4.2. A promising research direction worth exploring is including approximate computing as a defense mechanism against adversarial attacks. As the attacker is unaware of the approximate nature of the model during the inference, the adversarial samples generated for the original model may not be effective against the approximate one. This is because the loss of the exact and approximate models are not the same, and the adversarial generator needs it to build the perturbation. In [165], the authors show that by increasing the perturbation budget, it is possible to train an attacker capable of breaking a DNN that is transferable to different approximate models, potentially making approximate computing useless. However, the experiments are carried out with a single multiplier used to compute the entire network, which is not the case for the approximate DNNs presented in Section 4.2. It is possible that the attacker is not able to generate transferable adversarial samples with the same efficacy when there are $28 \cdot 10^{131}$ unique approximate DNNs compared to the dozens considered in [165]. In case the perturbation budget is increased, as proposed in [165] to break the proposed defensive approximation, resulting in a more distorted adversarial sample, filtering could be used to remove the pixel noise, as proposed in [175]. Therefore, even if approximate computing alone is not enough to protect the model from adversarial attacks, it could be used in conjunction with other techniques to increase the robustness of the model.

Bibliography

[1] N. Fasfous, M. R. Vemparala, A. Frickenstein, E. Valpreda, D. Salihu, J. Höfer, A. Singh, N.-S. Nagaraja, H.-J. Voegel, N. A. Vu Doan, M. Martina, J. Becker, and W. Stechele, "AnaCoNGA: Analytical HW-CNN co-design using nested genetic algorithms," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 238–243, IEEE, 2022. https://doi.org/10.23919/DATE54114.2022.9774574.

[2] M. R. Vemparala, N. Fasfous, A. Frickenstein, E. Valpreda, M. Camalleri, Q. Zhao, C. Unger, N. S. Nagaraja, M. Martina, and W. Stechele, "HW-Flow: A multi-abstraction level HW-CNN codesign pruning methodology," *Leibniz Transactions on Embedded Systems (LITES)*, vol. 1, 2022. https://doi.org/10.4230/LITES.8.1.3.

[3] N. Fasfous, M. R. Vemparala, A. Frickenstein, E. Valpreda, D. Salihu, N. A. V. Doan, C. Unger, N. S. Nagaraja, M. Martina, and W. Stechele, "HW-FlowQ: A multi-abstraction level HW-CNN co-design quantization methodology," *ACM Transactions on Embedded Computing Systems*, vol. 20, no. 5s, 2021. https://doi.org/10.1145/3476997.

[4] E. Valpreda, P. Morì, N. Fasfous, M. R. Vemparala, A. Frickenstein, L. Frickenstein, W. Stechele, C. Passerone, G. Masera, and M. Martina, "HW-Flow-Fusion: Inter-layer scheduling for convolutional neural network accelerators with dataflow architectures," *Electronics*, vol. 11, no. 18, p. 2933, 2022. https://doi.org/10.3390/electronics11182933.

[5] G. Aiello, B. Bussolino, E. Valpreda, M. Ruo Roch, G. Masera, M. Martina, and S. Marsi, "Nlcmap: A framework for the efficient mapping of non-linear convolutional neural networks on fpga accelerators," in *2022 IEEE International Conference on Image Processing (ICIP)*, pp. 926–930, 2022. https://doi.org/10.1109/ICIP46576.2022.9897288.

[6] E. Valpreda, G. Palumbo, M. Caon, G. Masera, and M. Martina, "Erode: Error resilient object detection by recovering bounding box and class information," in *2023 18th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*, pp. 277–280, 2023. https://doi.org/10.1109/PRIME58259.2023.10161894.

[7] N. Fasfous, L. Frickenstein, M. Neumeier, M. R. Vemparala, A. Frickenstein, E. Valpreda, M. Martina, and W. Stechele, "Mind the scaling factors: Resilience analysis of quantized adversarially robust cnns," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 706–711, 2022. https://doi.org/10.23919/DATE54114.2022.9774686.

[8] F. Guella, E. Valpreda, M. Caon, G. Masera, and M. Martina, "TEMET: Truncated REconfigurable Multiplier with Error Tuning," in *Applications in Electronics Pervading Industry, Environment and Society*, vol. 1110, pp. 370–377, 2024. https://doi.org/10.1007/978-3-031-48121-5_53.

[9] F. Guella, E. Valpreda, M. Caon, G. Masera, and M. Martina, "MARLIN: a co-design methodology for approximate reconfigurable inference of neural networks at the edge," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 71, no. 2105-2118, pp. 1–14, 2024. https://doi.org/10.1109/TCSI.2024.3365952.

[10] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," 2009. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014. https://doi.org/10.48550/arXiv.1409.1556.

[12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015. https://doi.org/10.1109/CVPR.2015.7298594.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on computer vision and pattern recognition*, pp. 770–778, 2016. https://doi.org/10.48550/arXiv.1409.4842.

[14] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4510–4520, June 2018. https://doi.org/10.1109/CVPR.2018.00474.

[15] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 10778–10787, 2020. https://doi.org/10.1109/CVPR42600.2020.01079.

[16] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2980–2988, 2017. https://doi.org/10.1109/ICCV.2017.322.

[17] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," 2017. https://doi.org/10.48550/arXiv.1706.05587.

[18] X. Zhou, D. Wang, and P. Krähenbühl, "Objects as points," in *ArXiv*, 2019. https://doi.org/10.48550/arXiv.1904.07850.

[19] S. Gidaris and N. Komodakis, "Object detection via a multi-region and semantic segmentation-aware cnn model," in *Proceedings of the IEEE international conference on computer vision*, pp. 1134–1142, 2015. https://doi.org/10.1109/ICCV.2015.135.

[20] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015. https://doi.org/10.48550/arXiv.1411.4038.

[21] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, pp. 801–818, Springer, 2018. https://doi.org/10.48550/arXiv.1409.4842.

[22] S. Pereira, A. Pinto, V. Alves, and C. A. Silva, "Brain tumor segmentation using convolutional neural networks in mri images," *IEEE Transactions on Medical Imaging*, vol. 35, pp. 1240–1251, May 2016. https://doi.org/10.1109/TMI.2016.2538465.

[23] D. B, R. K. K, R. S, and R. R, "Improved object detection in video surveillance using deep convolutional neural network learning," in *2021 Fifth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pp. 1–8, 2021. https://doi.org/10.1109/I-SMAC52330.2021.9640894.

[24] F. Zhang, F. Yang, C. Li, and G. Yuan, "Cmnet: A connect-and-merge convolutional neural network for fast vehicle detection in urban traffic surveillance," *IEEE Access*, vol. 7, pp. 72660–72671, 2019. https://doi.org/10.1109/ACCESS.2019.2919103.

[25] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2722–2730, Dec 2015. https://doi.org/10.1109/ICCV.2015.312.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015. https://doi.org/10.1038/nature14236.

[27] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. M. O. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *ArXiv*, 2015. https://doi.org/10.48550/arXiv.1509.02971.

[28] Tensorflow. Accessed: Jan 31, 2024 [Online], https://www.tensorflow.org/.

[29] PyTorch. Accessed: Jan. 31, 2024 [Online], https://pytorch.org/.

[30] GPU NVIDIA A100. Accessed: March 1, 2024 [Online], https://www.nvidia.com/it-it/data-center/a100/.

[31] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, 2017. https://doi.org/10.1145/3140659.3080246.

[32] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, 2015. https://doi.org/10.48550/arXiv.1409.0575.

[33] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The Cityscapes Dataset for Semantic Urban Scene Understanding," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. https://dblp.org/rec/journals/corr/CordtsORREBFRS16.bib.

[34] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, and K. Schindler, “Mot16: A benchmark for multi-object tracking,” *arXiv preprint arXiv:1603.00831*, 2016. https://doi.org/10.48550/arXiv.1603.00831.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” in *Advances in Neural Information Processing Systems (NeurIPS)* (F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds.), Curran Associates Inc., 2012. https://doi.org/10.1145/3065386.

[36] R. F., “The perceptron: A probabilistic model for information storage and organization in the brain,” *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958. https://psycnet.apa.org/doi/10.1037/h0042519.

[37] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015. https://doi.org/10.1109/ICCV.2015.123.

[38] A. Mao, M. Mohri, and Y. Zhong, “Cross-entropy loss functions: theoretical analysis and applications,” in *Proceedings of the 40th International Conference on Machine Learning*, 2023. https://doi.org/10.48550/arXiv.2304.07288.

[39] L. N. Smith, “Cyclical learning rates for training neural networks,” in *IEEE winter Conference on applications of computer vision (WACV)*, pp. 464–472, IEEE, 2017. https://doi.org/10.48550/arXiv.1506.01186.

[40] AMD Instinct MI300X Accelerators. Accessed: March 1, 2024 [Online], https://www.amd.com/en/products/accelerators/instinct/mi300/mi300x.html.

[41] P. Ramachandran, B. Zoph, and Q. Le, “Searching for activation functions,” *arXiv preprint arXiv:1710.05941*, 2018. https://arxiv.org/pdf/1710.05941.pdf.

[42] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML’15, p. 448–456, 2015. https://doi.org/10.48550/arXiv.1502.03167.

[43] J. Frankle, D. J. Schwab, and A. S. Morcos, “Training batchnorm and only batchnorm: On the expressive power of random features in cnns,” in *International Conference on Learning Representations*, 2020. https://doi.org/10.48550/arXiv.2003.00152.

[44] Intel Corp., *Lower Numerical Precision Deep Learning Inference and Training White Paper*, 1 2018. https://www.intel.com/content/dam/develop/external/us/en/documents/lower-numerical-precision-deep-learning-jan2018-754765.pdf.

[45] Scalable End-to-End Enterprise AI on 4th Gen Intel Xeon. Accessed Mar 21, 2024 [Online], https://cdrdv2.intel.com/v1/dl/getContent/781659?fileName=E2E_xeon-ai-final.pdf.

[46] 4th Gen AMD EPYC Processor Architecture. Accessed Mar 21, 2024 [Online], https://www.amd.com/system/files/documents/4th-gen-epyc-processor-architecture-white-paper.pdf.

[47] NVIDIA Pascal Architecture. Accessed: Mar 21, 2024 [online], https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[48] AMD RDNA Architecture. Accessed: Mar 21, 2024 [Online], https://www.amd.com/system/files/documents/rdna-whitepaper.pdf.

[49] M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, and M. Shafique, "Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead," *IEEE Access*, vol. 8, pp. 225134–225180, 2020. https://doi.org/10.1109/ACCESS.2020.3039858.

[50] Y. S. Shao *et al.*, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, p. 14–27, 2019. https://doi.org/10.1145/3352460.3358302.

[51] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 269–284, 2014. https://doi.org/10.1145/2654822.2541967.

[52] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017. https://doi.org/10.1109/JSSC.2016.2616357.

[53] A. Garofalo, G. Tagliavini, F. Conti, D. Rossi, and L. Benini, "XpulpNN: Accelerating quantized neural networks on RISC-V processors through ISA extensions," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 186–191, 2020. https://doi.org/10.23919/DATE48585.2020.9116529.

[54] H. Genc *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, p. 769–774, 2021. https://doi.org/10.1109/DAC18074.2021.9586216.

[55] Nvidia, "Nvdla open source project." https://nvdla.org/, 2017. Accessed: 2022-06-28.

[56] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic," in *2016 International Conference on Field-Programmable Technology (FPT)*, pp. 77–84, 2016. https://doi.org/10.1109/FPT.2016.7929192.

[57] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 3, 2018. https://doi.org/10.1145/3242897.

[58] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "High-performance low-memory lowering: Gemm-based algorithms for dnn convolution," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 99–106, 2020. https://doi.org/10.1109/SBAC-PAD49847.2020.00024.

[59] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, “Optimizing the convolution operation to accelerate deep neural networks on fpga,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1354–1367, 2018. https://doi.org/10.1109/TVLSI.2018.2815603.

[60] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, “Deep convolutional neural network architecture with reconfigurable computation patterns,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 8, pp. 2220–2233, 2017. https://doi.org/10.1109/TVLSI.2017.2688340.

[61] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, “On-device training under 256kb memory,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 22941–22954, 2022. https://doi.org/10.48550/arXiv.2206.15472.

[62] M. Merenda, C. Porcaro, and D. Iero, “Edge machine learning for AI-enabled IoT devices: A review,” *Sensors*, vol. 20, no. 9, p. 2533, 2020. https://doi.org/10.3390/s20092533.

[63] M. Shafique, T. Theocharides, V. J. Reddy, and B. Murmann, “TinyML: Current progress, research challenges, and future roadmap,” in *58th ACM/IEEE Design Automation Conference (DAC)*, pp. 1303–1306, 2021. https://doi.org/10.1109/DAC18074.2021.9586232.

[64] H. Ren, D. Anicic, and T. Runkler, “TinyOL: TinyML with online-learning on microcontrollers,” 2021. https://arxiv.org/abs/2103.08295.

[65] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, “On-device training under 256kb memory,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 22941–22954, 2022. https://doi.org/10.48550/arXiv.2206.15472.

[66] J. Launay, I. Poli, F. Boniface, and F. Krzakala, “Direct feedback alignment scales to modern deep learning tasks and architectures,” *Advances in neural information processing systems*, vol. 33, pp. 9346–9360, 2020. https://doi.org/10.48550/arXiv.2006.12878.

[67] Y. Umuroglu, L. Rasnayake, and M. Sjalander, “Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing,” in *FPL*, 2018. https://doi.org/10.48550/arXiv.1806.08862.

[68] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, “Xilinx adaptive compute acceleration platform: Versaltm architecture,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, p. 84–93, 2019. https://doi.org/10.1145/3289602.3293906.

[69] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, “DoReFa-net: Training low bit-width convolutional neural networks with low bitwidth gradients,” *ArXiv*, vol. abs/1606.06160, 2016. https://doi.org/10.48550/arXiv.1606.06160.

[70] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, “PACT: Parameterized clipping activation for quantized neural networks,” *ArXiv*, vol. abs/1805.06085, 2018. https://doi.org/10.48550/ARXIV.1805.06085.

[71] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” in *Proceedings of the European conference on computer vision (ECCV)*, pp. 525–542, Springer, 2016. https://doi.org/10.1007/978-3-319-46493-0_32.

[72] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks: Training neural networks with weights and activations constrained to +1 or -1,” in *Advances in Neural Information Processing Systems 29*, pp. 4107–4115, 2016. https://doi.org/10.48550/arXiv.1602.02830.

[73] M. Courbariaux, Y. Bengio, and J. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, 2015. https://doi.org/10.48550/arXiv.1511.00363.

[74] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, “Integer quantization for deep learning inference: Principles and empirical evaluation,” *CoRR*, vol. abs/2004.09602, 2020. https://doi.org/10.48550/arXiv.2004.09602.

[75] B. Barabasz and D. Gregg, “Winograd convolution for dnns: Beyond linear polynomials,” in *Advances in Artificial Intelligence (AI*IA)*, pp. 307–320, 2019. https://doi.org/10.1007/978-3-030-35166-3_22.

[76] P. Mori, S. B. Sampath, L. Frickenstein, M.-R. Vemparala, N. Fasfous, A. Frickenstein, W. Stechele, and C. Passerone, “Winotrain: Winograd-aware training for accurate full 8-bit convolution acceleration,” in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2023. https://doi.org/10.1109/DAC56929.2023.10247805.

[77] P. Mori, L. Frickenstein, S. B. Sampath, M. Thoma, N. Fasfous, M. R. Vemparala, A. Frickenstein, C. Unger, W. Stechele, D. Mueller-Gritschneder, and C. Passerone, “Wino vidi vici: Conquering numerical instability of 8-bit winograd convolution for accurate inference acceleration on edge,” in *2024 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pp. 53–62, 2024. https://doi.org/10.1109/WACV57701.2024.00013.

[78] T. Wang, K. Wang, H. Cai, J. Lin, Z. Liu, H. Wang, Y. Lin, and S. Han, “Apq: Joint search for network architecture, pruning and quantization policy,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2075–2084, 2020. https://doi.org/10.1109/CVPR42600.2020.00215.

[79] W. Jiang, L. Yang, S. Dasgupta, J. Hu, and Y. Shi, “Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4154–4165, 2020. https://doi.org/10.1109/TCAD.2020.3012863.

[80] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, “AMC: AutoML for Model Compression and Acceleration on Mobile Devices,” in *European Conference on Computer Vision (ECCV)*, 2018. https://doi.org/10.1007/978-3-030-01234-2_48.

[81] H. Jiang, F. J. H. Santiago, H. Mo, L. Liu, and J. Han, “Approximate arithmetic circuits: A survey, characterization, and recent applications,” *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2108–2135, 2020. https://doi.org/10.1109/JPROC.2020.3006451.

[82] L. Dadda, “Some schemes for parallel multipliers,” *Alta Frequenza*, vol. 34, pp. 349–356, 1965. https://ieeemilestones.ethw.org/w/images/8/82/Some_schemes_for_parallel_multipliers_%28reprint%29.pdf.

[83] A. G. M. Strollo, E. Napoli, D. De Caro, N. Petra, and G. D. Meo, “Comparison and extension of approximate 4-2 compressors for low-power approximate multipliers,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 9, pp. 3021–3034, 2020. https://doi.org/10.1109/TCSI.2020.2988353.

[84] V. Mrazek, L. Sekanina, and Z. Vasicek, “Libraries of approximate circuits: Automated design and application in CNN accelerators,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 4, pp. 406–418, 2020. https://doi.org/10.1109/JETCAS.2020.3032495.

[85] V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique, “ALWANN: Automatic layer-wise approximation of deep neural network accelerators without retraining,” in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019. https://doi.org/10.1109/iccad45719.2019.8942068.

[86] Y. L. Cun, J. S. Denker, and S. A. Solla, “Optimal Brain Damage,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 1990. DOI: https://dblp.org/rec/conf/nips/CunDS89.bib.

[87] B. Hassibi, D. G. Stork, G. Wolff, and T. Watanabe, “Optimal Brain Surgeon: Extensions and Performance Comparisons,” in *Advances in Neural Information Processing Systems (NeurIPS)*, (San Francisco, CA, USA), 1993. https://doi.org/10.1109/ICNN.1993.298572.

[88] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both Weights and Connections for Efficient Neural Network,” in *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 1135–1143, 2015. https://doi.org/10.48550/arXiv.1506.02626.

[89] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, “Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration,” *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4335–4344, 2018. https://doi.org/10.1109/CVPR.2019.00447.

[90] Y. Guo, A. Yao, and Y. Chen, “Dynamic Network Surgery for Efficient DNNs,” in *Advances in Neural Information Processing Systems (NeurIPS)* (D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, eds.), pp. 1379–1387, 2016. https://doi.org/10.48550/arXiv.1608.04493.

[91] A. Frickenstein, M.-R. Vemparala, N. Fasfous, L. Hauenschild, N.-S. Nagaraja, C. Unger, and W. Stechele, “Alf: Autoencoder-based low-rank filter-sharing for efficient convolutional neural networks,” in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, (DAC), 2020. https://doi.org/10.1109/DAC18072.2020.9218501.

[92] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” 2015. https://doi.org/10.48550/arXiv.1510.00149.

[93] Y. He, X. Zhang, and J. Sun, "Channel Pruning for Accelerating Very Deep Neural Networks," in *IEEE International Conference on Computer Vision (ICCV)*, pp. 1398–1406, 2017. https://doi.org/10.1109/ICCV.2017.155.

[94] Q. Huang, S. K. Zhou, S. You, and U. Neumann, "Learning to Prune Filters in Convolutional Neural Networks," *IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2018. https://doi.org/10.1109/WACV.2018.00083.

[95] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, and H. Sze, Vivienneand Adam, "NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications," in *The European Conference on Computer Vision (ECCV)*, Springer International Publishing, 2018. https://dblp.org/rec/journals/corr/abs-1804-03230.bib.

[96] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, P. Vajda, M. Uyttendaele, and N. K. Jha, "Chamnet: Towards efficient network design through platform-aware model adaptation," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11390–11399, 2019. https://doi.org/10.1109/CVPR.2019.01166.

[97] T. Yang, Y. Chen, and V. Sze, "Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6071–6079, 2017. https://doi.org/10.48550/arXiv.1611.05128.

[98] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," *Journal of Machine Learning Research (JMLR)*, vol. 18, no. 1, p. 6869–6898, 2017. https://doi.org/10.48550/arXiv.1609.07061.

[99] Y. Bengio, N. Léonard, and A. C. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *ArXiv*, 2013. https://doi.org/10.48550/arXiv.1308.3432.

[100] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (G. Gordon, D. Dunson, and M. Dudík, eds.), vol. 15, pp. 315–323, 2011. https://proceedings.mlr.press/v15/glorot11a.html.

[101] Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "HAWQ: Hessian aware quantization of neural networks with mixed-precision," in *IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019. https://doi.org/10.48550/arXiv.1905.03696.

[102] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2704–2713, IEEE Computer Society, 2018. https://doi.org/10.48550/arXiv.1712.05877.

[103] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. Le, “MnasNet: Platform-Aware Neural Architecture Search for Mobile,” in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2815–2823, 06 2019. https://doi.org/10.1109/CVPR.2019.00293.

[104] H. Cai, L. Zhu, and S. Han, “ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware,” in *International Conference on Learning Representations (ICLR)*, 2019. https://arxiv.org/pdf/1812.00332.pdf.

[105] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, “HAQ: Hardware-aware automated quantization with mixed precision,” in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 8604–8612, 2019. https://doi.org/10.1109/CVPR.2019.00881.

[106] L. Mei, P. Houshmand, V. Jain, S. Giraldo, and M. Verhelst, “Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators,” *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1160–1174, 2021. https://doi.org/10.1109/TC.2021.3059962.

[107] A. Parashar, P. Raina, Y. S. Shao, Y. 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 *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019. https://doi.org/10.1109/ISPASS.2019.00042.

[108] Y. N. Wu, J. S. Emer, and V. Sze, “Accelergy: An architecture-level energy estimation methodology for accelerator designs,” in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2019. https://doi.org/10.1109/ICCAD45719.2019.8942149.

[109] Q. Huang, A. Kalaiah, M. Kang, J. Demmel, G. Dinh, J. Wawrzynek, T. Norell, and Y. S. Shao, “Cosa: Scheduling by constrained optimization for spatial accelerators,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 554–566, 2021. https://doi.org/10.1109/ISCA52012.2021.00050.

[110] A. Inci, S. Virupaksha, A. Jain, T.-W. Chin, V. Thallam, R. Ding, and D. Marculescu, “QUIDAM: A framework for quantization-aware dnn accelerator and model co-exploration,” *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 2, 2023. https://doi.org/10.1145/3555807.

[111] D. Danopoulos, G. Zervakis, K. Siozios, D. Soudris, and J. Henkel, “AdaPT: Fast emulation of approximate DNN accelerators in PyTorch,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2022. https://doi.org/10.1109/TCAD.2022.3212645.

[112] D. Danopoulos, G. Zervakis, D. Soudris, and J. Henkel, “Transaxx: Efficient transformers with approximate computing,” in *arXiv eprint 2402.07545*, 2024. https://doi.org/10.48550/arXiv.2402.07545.

[113] F. Vaverka, V. Mrazek, Z. Vasicek, and L. Sekanina, “Tfapprox: Towards a fast emulation of dnn approximate hardware accelerators on gpu,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 294–297, 2020. https://doi.org/10.23919/DATE48585.2020.9116299.

[114] M. Pinos, V. Mrazek, F. Vaverka, Z. Vasicek, and L. Sekanina, "Acceleration techniques for automated design of approximate convolutional neural networks," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 13, no. 1, pp. 212–224, 2023. https://doi.org/10.1109/JETCAS.2023.3235204.

[115] M. S. Ansari, V. Mrazek, B. F. Cockburn, L. Sekanina, Z. Vasicek, and J. Han, "Improving the accuracy and hardware efficiency of neural networks using approximate multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 317–328, 2020. https://doi.org/10.1109/TVLSI.2019.2940943.

[116] X. He, W. Lu, G. Yan, and X. Zhang, "Joint design of training and hardware towards efficient and accuracy-scalable neural network inference," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 4, pp. 810–821, 2018. https://doi.org/10.1109/JETCAS.2018.2845396.

[117] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "AxNN: Energy-efficient neuromorphic systems using approximate computing," in *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 27–32, 2014. https://doi.org/10.1145/2627369.2627613.

[118] P. Jain, S. Huda, M. Maas, J. E. Gonzalez, I. Stoical, and A. Mirhoseini, "Learning to design accurate deep learning accelerators with inaccurate multipliers," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 184–189, 2022. https://doi.org/10.23919/DATE54114.2022.9774607.

[119] Z.-G. Tasoulas, G. Zervakis, I. Anagnostopoulos, H. Amrouch, and J. Henkel, "Weight-oriented approximation for energy-efficient neural network inference accelerators," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 4670–4683, 2020. https://doi.org/10.1109/TCSI.2020.3019460.

[120] E. Trommer, B. Waschneck, and A. Kumar, "Combining gradients and probabilities for heterogeneous approximation of neural networks," in *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–8, 2022. https://doi.org/10.48550/arXiv.2208.07265.

[121] C. De la Parra, A. Guntoro, and A. Kumar, "Efficient accuracy recovery in approximate neural networks by systematic error modelling," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, p. 365–371, 2021. https://doi.org/10.1145/3394885.3431533.

[122] I. Hammad, K. El-Sankary, and J. Gu, "Deep learning training with simulated approximate multipliers," in *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 47–51, 2019. https://doi.org/10.1109/ROBIO49542.2019.8961780.

[123] S. Ullah, S. S. Sahoo, and A. Kumar, "Clapped: A design framework for implementing cross-layer approximation in fpga-based embedded systems," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 475–480, 2021. https://doi.org/10.1109/DAC18074.2021.9586260.

[124] NVIDIA TensorRT - Programmable Inference Accelerator. Accessed: Apr 24, 2024 [Online], https://developer.nvidia.com/tensorrt.

[125] M. Gao, X. Yang, J. Pu, M. Horowitz, and C. Kozyrakis, "TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '19, pp. 807–820, 2019. http://doi.acm.org/10.1145/3297858.3304014.

[126] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer, W. J. Dally, J. Emer, S. W. Keckler, and B. Khailany, "Magnet: A modular accelerator generator for neural networks," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2019. https://doi.org/10.1109/ICCAD45719.2019.8942127.

[127] X. Yang, J. Pu, B. B. Rister, N. Bhagdikar, S. Richardson, S. Kvatinsky, J. Ragan-Kelley, A. Pedram, and M. Horowitz, "A Systematic Approach to Blocking Convolutional Neural Networks," *CoRR*, vol. abs/1606.04209, 2016. http://arxiv.org/abs/1606.04209.

[128] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using halide's scheduling language to analyze dnn accelerators," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, p. 369–383, 2020. https://doi.org/10.1145/3373376.3378514.

[129] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE (Volume: 105, Issue: 12)*, vol. 105, pp. 2295–2329, Nov 2017. https://doi.org/10.1109/JPROC.2017.2761740.

[130] S.-C. Kao and T. Krishna, "Magma: An optimization framework for mapping multiple dnns on multiple accelerator cores," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 814–830, 2022. https://doi.org/10.1109/HPCA53966.2022.00065.

[131] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2019. https://doi.org/10.1109/TCAD.2017.2785257.

[132] J. Lin, C. Gan, and S. Han, "Defensive quantization: When efficiency meets robustness," in *International Conference on Learning Representations*, 2018. https://doi.org/10.48550/arXiv.1904.08444.

[133] Y. He, P. Balaprakash, and Y. Li, "Fidelity: Efficient resilience analysis framework for deep learning accelerators," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 270–281, 2020. https://doi.org/10.1109/MICRO50266.2020.00033.

[134] Waymo. Accessed: Jan 31, 2024 [Online], https://www.waymo.com/.

[135] Uber AV. Accessed: Jan 31, 2024 [Online], https://www.uber.com/us/en/autonomous/.

[136] L. K. Draghetti, F. F. d. Santos, L. Carro, and P. Rech, "Detecting errors in convolutional neural networks using inter frame spatio-temporal correlation," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 310–315, 2019. https://doi.org/10.1109/IOLTS.2019.8854431.

[137] M. Son, H. Park, J. Ahn, and S. Yoo, "Making dram stronger against row hammering," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2017. https://doi.org/10.1145/3061639.3062281.

[138] A. S. Rakin, Z. He, and D. Fan, "Bit-flip attack: Crushing neural network with progressive bit search," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 1211–1220, 2019. https://doi.org/10.1109/ICCV.2019.00130.

[139] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *arXiv 1312.6199*, 2014. https://doi.org/10.48550/arXiv.1312.6199.

[140] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *arXiv 1412.6572*, 2015. https://doi.org/10.48550/arXiv.1412.6572.

[141] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *arXiv 1706.06083*, 2019. https://doi.org/10.48550/arXiv.1706.06083.

[142] E. Wong, L. Rice, and J. Z. Kolter, "Fast is better than free: Revisiting adversarial training," in *International Conference on Learning Representations*, 2019. https://doi.org/10.48550/arXiv.2001.03994.

[143] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multi-objective genetic algorithm: NSGA-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002. https://doi.org/10.1109/4235.996017.

[144] S. Narang, "Deepbench - baidu research." https://github.com/baidu-research/DeepBench, 2016.

[145] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, p. 883–898, 2021. https://doi.org/10.1145/3453483.3454083.

[146] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016. https://doi.org/10.1109/MICRO.2016.7783725.

[147] S. Kao, X. Huang, and T. Krishna, "Dnnfuser: Generative pre-trained transformer as a generalized mapper for layer fusion in DNN accelerators," *ArXiv*, vol. abs/2201.11218, 2022. https://doi.org/10.48550/arXiv.2201.11218.

[148] S. Karl, A. Symons, N. Fasfous, and M. Verhelst, "Genetic algorithm-based framework for layer-fused scheduling of multiple dnns on multi-core systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023. https://doi.org/10.23919/DATE56975.2023.10137070.

[149] E. Huang and R. E. Korf, "Optimal rectangle packing: An absolute placement approach," *Journal of Artificial Intelligence Research*, vol. 46, pp. 47–87, 2013. https://arxiv.org/abs/1402.0557.

[150] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 751–764, 2017. http://doi.acm.org/10.1145/3037697.3037702.

[151] W. Hassen and H. Amiri, "Block matching algorithms for motion estimation," in *2013 7th IEEE International Conference on e-Learning in Industrial Electronics (ICELIE)*, pp. 136–139, 2013. https://doi.org/10.1109/ICELIE.2013.6701287.

[152] S. Zhu and K.-K. Ma, "A new diamond search algorithm for fast block-matching motion estimation," *IEEE Transactions on Image Processing*, vol. 9, no. 2, pp. 287–290, 2000. https://doi.org/10.1109/83.821744.

[153] A. Bewley *et al.*, "Simple online and realtime tracking," in *2016 IEEE international conference on image processing (ICIP)*, pp. 3464–3468, 2016. https://doi.org/10.1109/ICIP.2016.7533003.

[154] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955. https://doi.org/10.1002/nav.3800020109.

[155] E. Bochinski, V. Eiselein, and T. Sikora, "High-speed tracking-by-detection without using image information," in *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pp. 1–6, 2017. https://doi.org/10.1109/AVSS.2017.8078516.

[156] E. Bochinski, T. Senst, and T. Sikora, "Extending iou based multi-object tracking by visual information," in *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pp. 1–6, 2018. https://doi.org/10.1109/AVSS.2018.8639144.

[157] K.-H. Chow, L. Liu, M. Loper, J. Bae, M. Emre Gursoy, S. Truex, W. Wei, and Y. Wu, "Adversarial objectness gradient attacks in real-time object detection systems," in *IEEE International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications*, pp. 263–272, 2020. https://doi.org/10.1109/TPS-ISA50397.2020.00042.

[158] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, "Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2164, p. 20190155, 2020. https://doi.org/10.1098/rsta.2019.0155.

[159] Q. Lou, F. Guo, M. Kim, L. Liu, and L. Jiang, "AutoQ: Automated kernel-wise neural network quantization," in *International Conference on Learning Representations*, 2019. https://doi.org/10.48550/arXiv.1902.05690.

[160] İ. Taştan, M. Karaca, and A. Yurdakul, "Approximate CPU design for iot end-devices with learning capabilities," *Electronics*, vol. 9, no. 1, 2020. https://www.mdpi.com/2079-9292/9/1/125.

[161] PULP platform. Accessed: Jan. 23, 2023 [Online], https://pulp-platform.org.

[162] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, 2017. https://doi.org/10.1109/TVLSI.2017.2654506.

[163] M. Pinos, V. Mrazek, and L. Sekanina, "Prediction of inference energy on cnn accelerators supporting approximate circuits," in *26th Intl. Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, pp. 45–50, 2023. https://doi.org/10.1109/DDECS57882.2023.10139724.

[164] E. Atoofian, "Increasing robustness against adversarial attacks through ensemble of approximate multipliers," in *IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–8, 2022. https://doi.org/10.1109/NAS55553.2022.9925476.

[165] A. Siddique and K. A. Hoque, "Is approximation universally defensive against adversarial attacks in deep neural networks?," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, p. 364–369, 2022. https://dl.acm.org/doi/10.5555/3539845.3539934.

[166] F. Conti, "Technical report: NEMO DNN quantization for deployment model," 2020. https://doi.org/10.48550/ARXIV.2004.05930.

[167] A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi, and F. Conti, "DORY: Automatic end-to-end deployment of real-world DNNs on low-cost IoT MCUs," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1253–1268, 2021. https://doi.org/10.1109/TC.2021.3066883.

[168] C. Baugh and B. Wooley, "A two's complement parallel array multiplication algorithm," *IEEE Transactions on Computers*, vol. C-22, no. 12, pp. 1045–1047, 1973. https://doi.org/10.1109/T-C.1973.223648.

[169] M. de la Guia Solaz, W. Han, and R. Conway, "A flexible low power DSP with a programmable truncated multiplier," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 59, no. 11, pp. 2555–2568, 2012. https://doi.org/10.1109/TCSI.2012.2189059.

[170] Europractice UMC standard cells libraries. Accessed: Mar 1, 2024 [Online], https://europractice-ic.com/technologies/asics/umc/umc-scl/.

[171] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," in *MLSys*, 2024. https://doi.org/10.48550/arXiv.2306.00978.

[172] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: accurate and efficient post-training quantization for large language models," in *Proceedings of the 40th International Conference on Machine Learning*, 2023. https://doi.org/10.48550/arXiv.2211.10438.

[173] H. Yang, H. Yin, M. Shen, P. Molchanov, H. Li, and J. Kautz, "Global vision transformer pruning with hessian-aware saliency," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 18547–18557, 2023. https://doi.org/10.48550/arXiv.2110.04869.

[174] Meta Llama 3. Accessed: Apr 24, 2024 [Online], https://github.com/meta-llama/llama3/tree/main.

[175] C. Xie, Y. Wu, L. v. d. Maaten, A. L. Yuille, and K. He, "Feature denoising for improving adversarial robustness," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 501–509, 2019. https://doi.org/10.1109/CVPR.2019.00059.