Politecnico di Torino

Politecnico di Torino

Master's Degree in Embedded Systems

A.a. 2024/2025

Graduation Session 12/2025

Rapid Prototyping of Edge AI Accelerators

An HLS-based Approach for CNNs on FPGAs for the AIdge ML Deployment Framework

Supervisors:

Prof. Luciano LAVAGNO
Fabian CHERSI (CEA Saclay)
Prof. Mihai T. LAZARESCU
Teodoro URSO
Roberto BOSIO

Candidate:

Jacopo CESARETTI

Abstract

The computational demands of modern Artificial Intelligence (AI), particularly for Convolutional Neural Networks (CNNs) in computer vision, are increasingly challenging to meet with traditional cloud-centric approaches. Relying solely on centralized cloud infrastructures introduces significant latency and bandwidth bottlenecks, while also raising concerns about data privacy and the substantial energy consumption of large-scale data centers. To overcome these limitations, the paradigm of edge computing has gained prominence, processing data locally on dedicated hardware. This work explores the deployment of CNNs on Field-Programmable Gate Arrays (FPGAs), reconfigurable devices that offer a compelling blend of parallel processing capability and energy efficiency for edge applications. This thesis investigates a rapid prototyping methodology for FPGA-based CNN acceleration, leveraging the High-Level Synthesis (HLS) design flow. The research focuses on the implementation and optimization of critical network layers, including quantization and pooling, to create a library of hardware-efficient functions. By abstracting the low-level hardware complexity through HLS, this work enables a streamlined path from a software-defined model to a customized hardware implementation. The final synthesis and deployment phase aims to validate this prototyping approach, fine-tuning the system for optimal performance on the target FPGA.

The primary contribution of this work is a demonstrated methodology that significantly accelerates the development cycle for edge AI accelerators. This approach facilitates rapid exploration of the design space, allowing for quick iteration and evaluation of different model architectures and hardware optimization strategies on FPGAs. The findings contribute to making efficient, low-latency AI inference more accessible by reducing the barrier to entry and development time for hardware deployment.

Desidero esprimere la mia più profonda gratitudine ai professori Lavagno e Lazarescu, la cui guida attenta e competente ha rappresentato un punto fermo durante l'intero percorso di questo lavoro. Un sincero ringraziamento va anche ai dottorandi Teodoro e Roberto, per il supporto costante, la disponibilità e i preziosi suggerimenti che hanno contribuito in modo significativo alla realizzazione di questo progetto.

Un pensiero colmo di riconoscenza è rivolto ai miei genitori, che con amore, pazienza e fiducia incrollabile mi hanno sostenuto in ogni scelta. Senza il loro appoggio silenzioso ma fondamentale, nulla di tutto questo sarebbe stato possibile.

A mia sorella Ludovica va un ringraziamento speciale: grazie per la tua presenza affettuosa, per la leggerezza che sai portare nei momenti più impegnativi e per il modo semplice ma sincero con cui riesci sempre a regalarmi un sorriso.

Un caro ringraziamento va ai miei nonni, che con il loro affetto discreto e la loro costante vicinanza sono stati una fonte preziosa di incoraggiamento e forza.

Infine, un grazie di cuore ai miei amici, che con la loro compagnia, comprensione e spontaneità hanno reso questo percorso più sereno, ricordandomi l'importanza di circondarsi di persone autentiche.

Table of Contents

List of Figures

Acronyms

AI Artificial Intelligence

ASIC Application-Specific Integrated Circuit

BN Batch Normalization

BRAM Block RAM

CEA Commissariat à l'Énergie Atomique et aux Énergies Alternatives

CNN Convolutional Neural Network

CPU Central Processing Unit

DMA Direct Memory Access

DSP Digital Signal Processor

DPU Deep Learning Processing Unit

FF Flip-Flop

FEM Finite Element Method

FFT Fast Fourier Transform

FPGA Field-Programmable Gate Array

FPS Frames Per Second

GAP Global Average Pooling

GEMM General Matrix Multiplication

GPU Graphics Processing Unit

HDL Hardware Description Language

HLS High-Level Synthesis

II Initiation Interval

INT Integer

LTI Linear Time-Invariant

LUT Look-Up Table

LUTRAM Look-Up Table RAM

MAC Multiply-Accumulate

MCMK Multiple Channel Multiple Kernel

MPSoC Multiprocessor System-on-Chip

N2D2 Neural Network Design & Deployment

ONNX Open Neural Network Exchange

PL Programmable Logic

PS Processing System

PYNQ Python Productivity for Zynq

QAT Quantization-Aware Training

ReLU Rectified Linear Unit

ResNet Residual Network

RTL Register-Transfer Level

SENet Squeeze-and-Excitation Network

SGD Stochastic Gradient Descent

SoC System-on-Chip

TCL Tool Command Language

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

XDC Xilinx Design Constraints

ZCU Zynq UltraScale+ Evaluation Board

Chapter 1

Introduction

The rapid advancement of Artificial Intelligence (AI) is fundamentally transforming industries and societal infrastructures. At the heart of this transformation lie Deep Learning models, particularly Convolutional Neural Networks (CNNs), which have become the de facto standard for a wide range of computer vision tasks, from autonomous driving to medical image analysis. However, the remarkable performance of these models comes at a significant computational cost, creating a critical challenge for their deployment in real-world, latency-sensitive applications. Traditional cloud-centric approaches, where data is sent to a remote server for processing, are often inadequate for these scenarios. The inherent latency, bandwidth constraints, and privacy concerns associated with constant data transmission to the cloud pose significant bottlenecks. This has led to the emergence of edge computing, a paradigm that shifts computation from the centralized cloud to devices at the network's edge, closer to where data is generated. By processing data locally, edge computing offers reduced latency, improved bandwidth efficiency, and enhanced data privacy.

Field-Programmable Gate Arrays (FPGAs) have emerged as a particularly compelling hardware platform for edge AI acceleration. Their reconfigurable architecture allows for the creation of custom, application-specific compute engines that can exploit the fine-grained parallelism inherent in CNN workloads. Furthermore, FPGAs often provide a superior performance-per-watt ratio compared to general-purpose processors, a critical consideration for power-constrained edge devices. The design process for FPGAs has been significantly streamlined by High-Level Synthesis (HLS) tools, such as Xilinx Vitis HLS, which enable developers to describe hardware accelerators using high-level languages like C++ instead of traditional Hardware Description Languages (HDLs). This raises the level of abstraction and is key to enabling rapid prototyping.

Despite these advancements, a significant gap remains between a trained CNN model and its efficient implementation on FPGA hardware. The process of model

quantization, layer-specific optimization, and final system synthesis often requires substantial hardware expertise and remains a time-consuming, iterative process. This thesis addresses this challenge by proposing and validating a rapid prototyping **methodology** for deploying CNNs onto FPGA platforms. Our work, conducted in collaboration with CEA Saclay, demonstrates an end-to-end workflow. The core contributions of this work are:

- The development and integration of optimized, hardware-aware layers for quantization and pooling, crucial for reducing model complexity and leveraging parallel hardware.
- A **methodology** for the system-level synthesis and deployment of a complete quantized CNN model onto an FPGA target.
- An evaluation of this rapid prototyping **approach**, assessing its effectiveness in generating efficient hardware accelerators for edge inference.

1.1 Goal

The primary goal of this Master's thesis is to design, implement, and evaluate a rapid prototyping **methodology** for the deployment of Convolutional Neural Networks (CNNs) onto FPGA-based edge devices. This work aims to bridge the gap between software-defined neural network models and their efficient hardware realization, reducing the traditional development time and required expertise for creating custom AI accelerators.
To achieve this overarching goal, the work is structured around the following specific objectives:

1. **Library Development:** To design and implement a set of parameterized, hardware-optimized template functions for critical CNN layers using High-Level Synthesis (HLS). These templates form the core of a reusable **library of hardware components**.
2. **Model Integration and Optimization:** To apply this **library** to a target CNN model, focusing on hardware-centric optimizations to ensure the model is suitable for resource-constrained FPGA environments.
3. **System Implementation:** To execute the complete hardware synthesis flow, translating the optimized model into a deployable bitstream for the target FPGA platform, and to fine-tune the hardware configuration for optimal performance.

4. **Methodology Validation:** To assess the viability and effectiveness of the proposed rapid prototyping **methodology**, evaluating its ability to streamline the path from a high-level model to a functional hardware accelerator for low-latency, edge-based inference.

This work was conducted in collaboration with CEA Saclay, utilizing and extending the capabilities of the AIdge deep learning framework.

1.2 Thesis'structure

The remainder of this thesis is organized as follows:

- **Chapter 2** introduces the theoretical background needed for this work, covering the fundamental principles and concepts relevant to embedded neural network deployment.
- **Chapter 3** provides an overview of the **NEUROKIT2E** environment and the **Aidge Framework**, outlining their roles, components, and integration within the proposed workflow.
- **Chapter 4** focuses on the hardware design and implementation, describing the system architecture, custom hardware modules, and the overall development methodology.
- **Chapter 5** details the Vivado design process, including block design creation, synthesis flow, and the validation procedures used to verify system correctness.
- **Chapter 6** presents the experimental results and provides a detailed performance evaluation of the implemented accelerator, including latency, throughput, and energy measurements.
- **Chapter 7** summarizes the conclusions drawn from this work and outlines several directions for future development, highlighting potential improvements and extensions of the proposed methodology.

Chapter 2

Theoretical Foundations

Convolutional Neural Networks (CNNs) are among the most influential architectures in the field of *deep learning*, forming the basis of major breakthroughs in visual recognition and a wide range of applications, including natural language processing, biomedical imaging, and autonomous systems [1]. The concept of CNNs is biologically inspired by the human visual cortex, where neurons respond to local receptive fields in the visual space. This structure allows CNNs to automatically learn hierarchical representations of data, reducing the need for handcrafted feature extraction [2].

2.1 The Convolution Operation: Definition and Mathematical Foundations

Convolution is a fundamental mathematical operation that describes how two functions combine to produce a third one expressing their local overlap as a function of displacement. It is extensively employed in fields such as signal processing, physics, and applied mathematics to model systems where an input interacts with a localized response or impulse function [3, 4]. Conceptually, convolution quantifies the similarity between an input signal and a shifted version of a kernel or filter, thereby emphasizing particular structures or patterns embedded in the signal.

2.1.1 Formal Definition in Continuous and Discrete Domains

In the continuous one-dimensional case, the convolution between two real-valued functions $f(x)$ and $g(x)$ is defined as:

$$(f * g)(x) = \int_{-\infty}^{+\infty} f(u)\, g(x-u)\, du, \tag{2.1}$$

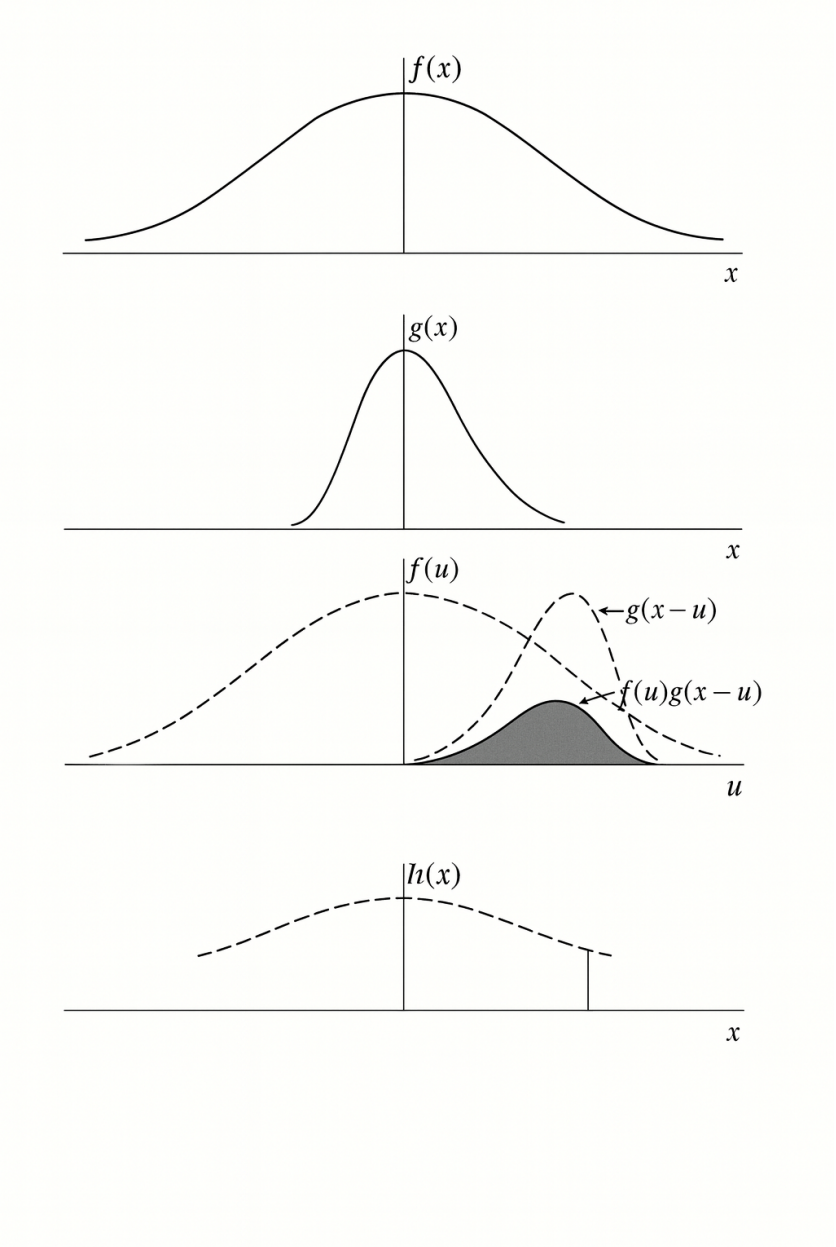

Figure 2.1: Illustration of a one-dimensional convolution between a signal and a kernel [3].

where $*$ denotes the convolution operator. The integral represents the degree of overlap between $f(u)$ and a reversed and shifted version of $g(u)$ [3]. The reversal arises from the dependence on $(x - u)$, distinguishing convolution from cross-correlation, where the kernel is not reversed.

In the discrete case, given a signal $I(i)$ of length n and a discrete kernel $K(u)$ of length s, convolution is expressed as:

$$(I * K)(i) = \sum_{u=0}^{s-1} I(i-u)\, K(u), \tag{2.2}$$

where appropriate boundary conditions (e.g., zero-padding or circular extension) are applied to handle indices outside the signal's domain [5].

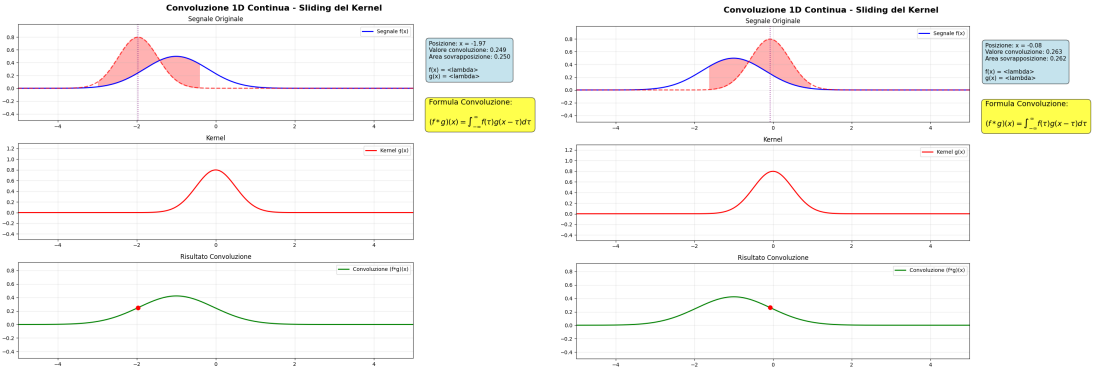

Figure 2.2: Visualization of the convolution operation: the kernel undergoes time reversal (red) before being translated across the input sequence (blue), computing the output (green) at each position.

Convolution naturally generalizes to higher dimensions. For a two-dimensional input signal (e.g., an image) $X(m, n)$ and a two-dimensional kernel $W(i, j)$, the convolution is defined as:

$$Y(m,n) = \sum_{i} \sum_{j} X(m-i, n-j)\, W(i,j), \tag{2.3}$$

where $Y(m, n)$ is the resulting filtered signal. Each element of Y corresponds to a weighted sum of localized regions of X, with the weights specified by the kernel W. This operation can be extended to three-dimensional signals, such as volumetric data or spatiotemporal sequences, by adding an additional summation index [5, 6].

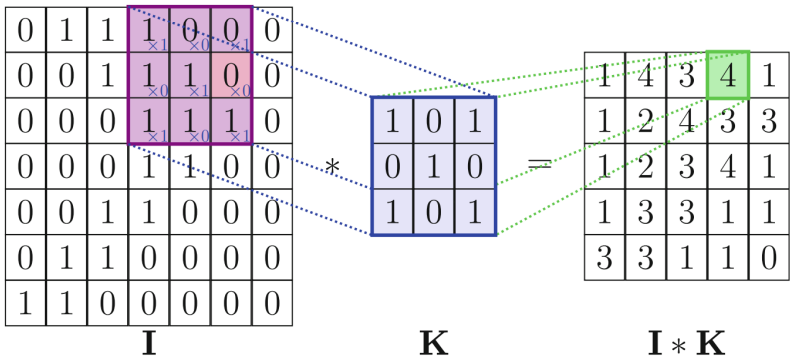

Figure 2.3: Two-dimensional convolution: the kernel slides spatially over the input domain, producing the output map [7].

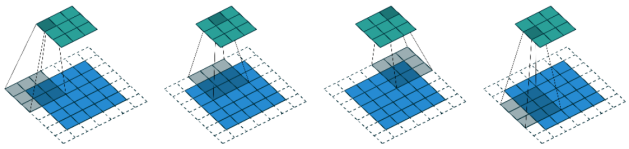

Figure 2.4: Image showing a convolutional filter sweeping over an image, with a padding of size 1 shown by the white squares. Note that the filter moves two squares for each step, which represents a stride of 2 [8].

2.1.2 Physical and Signal-Processing Interpretation

From a signal-processing viewpoint, convolution represents the response of a linear time-invariant (LTI) system to an arbitrary input [4]. If $x(t)$ is the input signal and $h(t)$ the system's impulse response, then the output $y(t)$ is given by:

$$y(t) = (x * h)(t) = \int_{-\infty}^{+\infty} x(\tau)\, h(t-\tau)\, d\tau. \tag{2.4}$$

In this context, the kernel acts as a detector or template: high output values occur when the input locally resembles the kernel, a property closely related to matched filtering [9].
In spatial domains, such as image or field processing, convolution is interpreted as a localized aggregation operation that smooths, enhances, or extracts specific structures, depending on the kernel configuration (e.g., Gaussian, Laplacian, Sobel), as can be seen in fig.2.5. Different kernel shapes correspond to distinct physical or mathematical transformations.

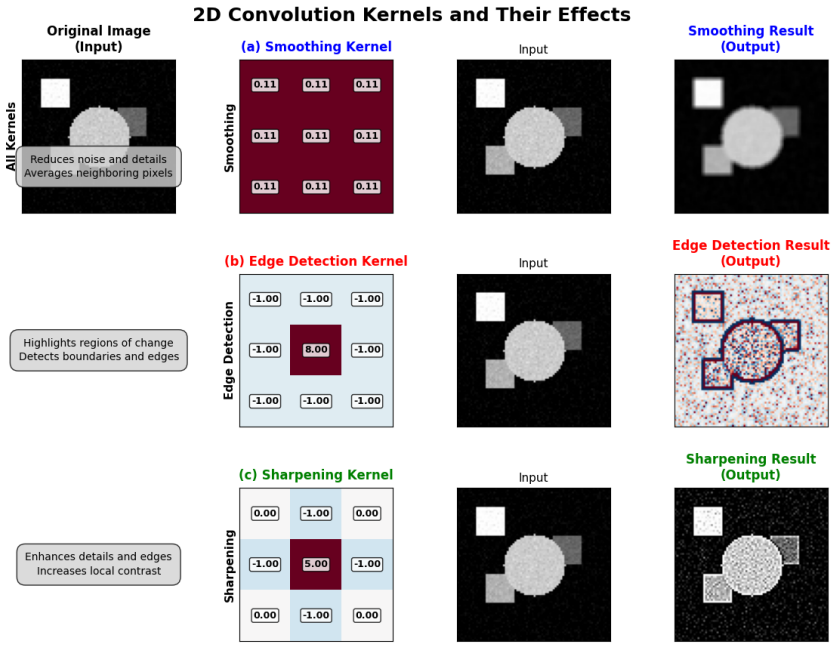

Figure 2.5: Examples of two-dimensional convolution kernels and their effects: (a) smoothing, (b) edge detection, (c) sharpening.

2.1.3 Properties of Convolution

Convolution exhibits several key mathematical properties that make it particularly useful for analytical and computational purposes [3, 6]:

1. **Commutativity:** $f * g = g * f$
2. **Associativity:** $(f * g) * h = f * (g * h)$
3. **Distributivity over addition:** $f * (g + h) = f * g + f * h$
4. **Linearity:** $a(f * g) + b(f * h) = f * (ag + bh)$
5. **Shift Invariance:** A translation in the input corresponds to a translation in the output.

Furthermore, convolution in the time or spatial domain corresponds to pointwise multiplication in the frequency domain, as established by the *Convolution Theorem*:

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}, \tag{2.5}$$

where $\mathcal{F}\{\cdot\}$ denotes the Fourier transform. This duality forms the basis of efficient computational algorithms such as the Fast Fourier Transform (FFT) convolution method [10].

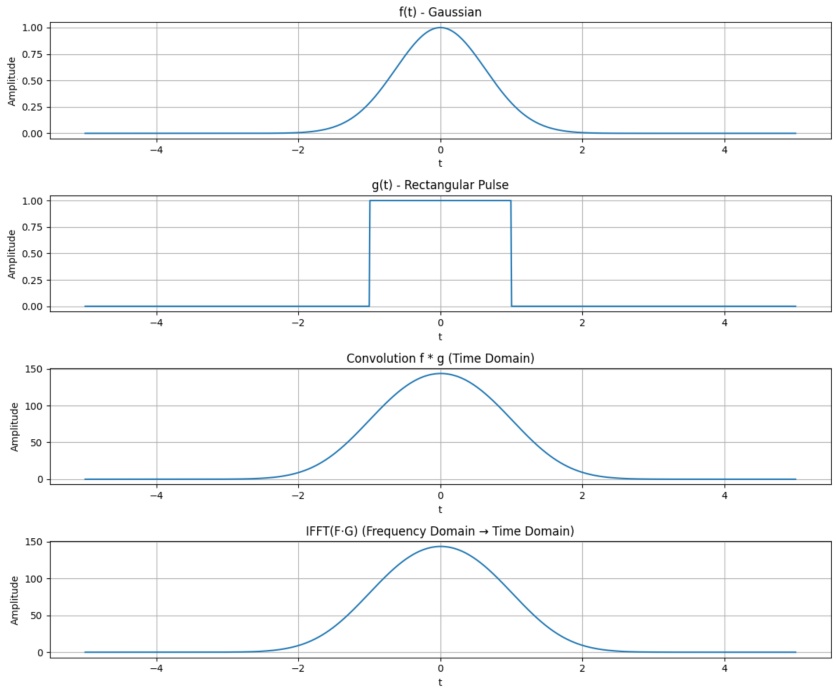

Figure 2.6: Graphical illustration of the Convolution Theorem: convolution in the spatial domain corresponds to multiplication in the frequency domain.

2.2 Symmetric Uniform Quantization and Fixed-Point Arithmetic

Quantization is a core building block of efficient deployment of deep neural networks on constrained hardware such as FPGAs, ASICs, and embedded processors [11, 12, 13]. Among various quantization schemes, *symmetric uniform quantization* has become the most widely adopted formulation, owing to its simplicity, predictable numerical behavior, and tight correspondence with signed fixed-point arithmetic [14, 15].

This approach discretizes both weights and activations onto uniformly spaced integer grids centered around zero, enabling the use of deterministic and hardware-friendly fixed-point arithmetic [11, 12]. Unlike asymmetric schemes that require offset handling, symmetric quantization preserves sign symmetry and simplifies arithmetic

circuits, which is particularly advantageous in FPGA and ASIC implementations where hardware efficiency, determinism, and latency are critical constraints.

2.2.1 Mathematical Formulation of Symmetric Uniform Quantization

Let $x \in \mathbb{R}$ be an element of a tensor X. In symmetric uniform quantization, x is mapped to an integer x_q using a quantization step size s (often called *scale*) as follows:

$$x_q = \text{Clip}\left(\text{Round}\left(\frac{x}{s}\right), -Q_{\max}, Q_{\max}\right), \qquad \hat{x} = s\, x_q,$$

where $Q_{\max} = 2^{b-1} - 1$ for a b-bit signed representation as shown in 2.7.

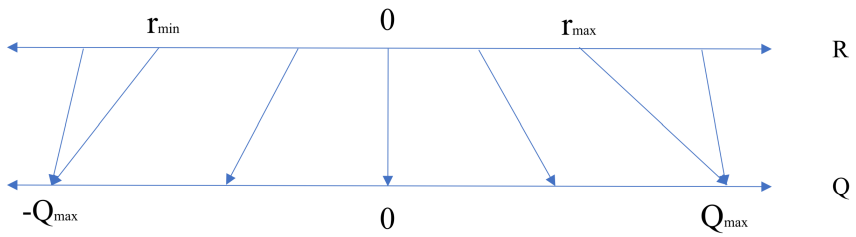

Figure 2.7: Symmetric uniform quantization grid. The real axis is discretized into uniformly spaced levels of width s, symmetric around zero. Inputs x are rounded to the nearest integer multiple of s and clipped to $\pm Q_{\max} s$, where $Q_{\max} = 2^{b-1} - 1$. The quantization error $e = \hat{x} - x$ is bounded by $\pm s/2$ within the dynamic range.

The quantizer thus partitions the real axis into uniform bins of width s, symmetrically distributed about zero, and the rounding operation assigns each input to the nearest discrete level. The reconstruction $\hat{x}$ represents the closest representable value to x within the quantization grid.
The quantization error is defined as

$$e = \hat{x} - x,$$

and as demonstrated in [11, 12], the quantization error for non-saturated values ($|x| \leq Q_{\max} s$) can be modeled as a uniform random variable $e \sim \mathcal{U}(-s/2, s/2)$, leading to:

$$\mathbb{E}[e] = 0, \qquad \text{Var}(e) = \frac{s^2}{12}.$$

This model is widely adopted in analytical studies of quantization noise in deep neural networks [11]. For saturated values, a clipping error occurs when $|x| > Q_{\max} s$, which motivates careful selection of s to balance resolution against dynamic range.

A common calibration heuristic, as stated in [12], sets

$$s = \frac{\max(|x|)}{Q_{\max}},$$

which ensures that the largest-magnitude value in X lies exactly at the quantization limit. This simple scale rule yields near-optimal quantization when the distribution of x is symmetric and approximately unimodal, such as Gaussian or Laplacian distributions observed in CNN weights after training [12].

2.2.2 Implementation in Fixed-Point Arithmetic

Once tensors are quantized, all subsequent operations—multiplications, additions, and accumulations—can be performed in fixed-point arithmetic. Let W_q and X_q denote quantized integer weights and activations with scale factors s_W and s_X, respectively. A convolutional output channel can be expressed as:

$$Y_q = \sum_{m=1}^{M} W_{q,m} \cdot X_{q,m},$$

and the corresponding real-valued output is reconstructed as:

$$\hat{Y} = s_W s_X \, Y_q.$$

The term $s_W s_X$ represents the effective scaling factor mapping integer accumulations back to real-valued outputs [13].

In practical implementations, the intermediate accumulation Y_q is performed at a higher precision (e.g., 32-bit integer) to prevent overflow, while W_q and X_q are stored in low precision (e.g., 8-bit). This mixed-width computation model ensures that dynamic range is preserved in the accumulator without incurring floating-point overhead.

Goyal et al. [13] emphasized that this formulation makes quantized inference both numerically stable and hardware-efficient. Since scale factors are powers of two in many FPGA implementations (i.e., $s = 2^{-k}$), multiplications by s or $1/s$ can be realized as bit-shifts, eliminating costly multipliers altogether. This is consistent with earlier work on Incremental Network Quantization (INQ) by Zhou et al. [16], where weights are constrained to power-of-two values, effectively converting convolutions into shift–accumulate operations. Such representations drastically reduce hardware utilization, as shift operations map directly to simple logic elements on FPGA fabrics.

2.2.3 Fixed-Point Formats and Dynamic Range Considerations

In hardware terms, a fixed-point number in $Q_{i.f}$ format dedicates i bits to the integer part (including the sign) and f bits to the fractional part. For instance, an 8-bit signed quantized value with $f = 7$ corresponds to $Q_{1.7}$, representing numbers in $[-1, 1 - 2^{-7}]$ with resolution 2^{-7}. The scale factor s defines the implicit binary point placement between the integer and fractional regions.
The dynamic range of representable values in a symmetric b-bit quantizer is $[-(2^{b-1} - 1)s, (2^{b-1} - 1)s]$. If s is too small, values may saturate (clipping error); if s is too large, quantization noise dominates. The goal of calibration is therefore to select s that balances these two sources of distortion: reducing s enhances resolution at the cost of higher clipping risk, while increasing s limits clipping but leads to poorer precision.
Nagel et al. [12] demonstrated that for CNN weights, the optimal range often corresponds to a small multiple of the standard deviation of the weight distribution (typically 2–3σ), while activations require empirical calibration from representative input batches.

2.2.4 Arithmetic Operations and Scaling Management

During inference, quantized layers require careful scaling propagation. Given:

$$Y = \mathrm{Conv}(X, W) + b,$$

with quantized operands, the integer computation proceeds as:

$$Y_{\mathrm{int}} = \sum_m W_{\mathrm{int},m} \cdot X_{\mathrm{int},m},$$

and the real-valued equivalent is $\hat{Y} = s_W s_X \, Y_{\mathrm{int}} + b$. To feed the next layer, the activation tensor must be re-quantized to the new integer scale s_Y, which requires:

$$Y'_{\mathrm{int}} = \mathrm{Round}\Big(\frac{s_W s_X}{s_Y} \, Y_{\mathrm{int}} \Big),$$

where the ratio $\frac{s_W s_X}{s_Y}$ is usually precomputed as a fixed-point constant or, in hardware-friendly designs, constrained to a power-of-two approximation [16, 13]. This step, known as *requantization*, is critical for maintaining consistent scaling throughout the network [11, 12].
In symmetric quantization, no zero-point adjustments are necessary, so the requantization path simplifies to a single multiply–accumulate followed by a bit-shift or scaling constant. This simplification reduces latency, simplifies control logic, and ensures deterministic numerical behavior across platforms.

2.2.5 Hardware Implications on FPGAs

Symmetric uniform quantization aligns exceptionally well with the fixed-point arithmetic resources available on FPGA devices. Modern FPGAs feature dedicated DSP slices that natively perform signed integer multiply–accumulate (MAC) operations. By quantizing weights and activations to INT8 or INT4, the number of effective MAC units per DSP slice can be increased through packing—executing multiple low-bit MACs within a single DSP block [14]. This allows parallel convolutional computations and higher throughput under identical resource constraints [13].
Because symmetric quantization produces zero-centered integer tensors, the arithmetic pipeline can omit bias offsets, which simplifies the datapath design [17]. Moreover, when quantization scales are powers of two, fixed-point multipliers degenerate into shift operators, and accumulation proceeds through integer addition only. This property drastically reduces dynamic power consumption and latency, while ensuring precise bit-level control of numerical behavior—attributes critical in real-time embedded inference.
FPGA toolchains further benefit from the deterministic range of fixed-point arithmetic: overflow conditions are predictable and can be handled by compile-time saturation logic or wrap-around arithmetic, depending on the design target. In hardware–software co-design flows, quantization parameters (bit-widths, scales, and Q-formats) are typically exposed as compile-time constants or C++ template parameters, allowing each layer to be synthesized with its optimal numeric precision.

2.2.6 Error Propagation and Network Robustness

While symmetric uniform quantization introduces quantization noise, empirical studies demonstrate that deep convolutional networks exhibit strong robustness to small-scale perturbations in weights and activations [11, 12]. The variance of accumulated quantization error across layers tends to remain bounded due to normalization and ReLU sparsity effects [11]. However, special attention must be given to residual connections (as in ResNet architectures), where mismatched scales between branches can lead to destructive interference. Ensuring consistent symmetric quantization across skip and main paths—by aligning s values or inserting a shared requantization step—preserves the additive integrity of the residual sum [12].

Summary Symmetric uniform quantization provides a theoretically clean and hardware-efficient foundation for integer inference. It aligns naturally with signed fixed-point arithmetic, minimizes arithmetic complexity, and supports deterministic implementations on reconfigurable logic. The principles outlined here form the numerical backbone of quantized neural inference pipelines studied in [11, 12, 13,

16, 18].

2.3 Pooling in CNNs: theory, practice, and the roles of Max Pooling and Global Average Pooling

Pooling layers play a crucial role in convolutional neural networks (CNNs) by reducing spatial resolution while preserving the most discriminative characteristics of feature maps. Through the aggregation of local or global neighborhoods into compact descriptors, pooling enhances the effective receptive field, promotes a degree of translation tolerance, and reduces computational load. Despite the many variants proposed in recent literature, *max pooling* and *global average pooling* (GAP), shown in fig 2.8, remain the canonical choices in contemporary architectures, largely due to their complementary inductive biases and empirically demonstrated effectiveness across a wide range of datasets [19].
Recent comprehensive evaluations confirm that the choice between max and average pooling has non-trivial consequences for accuracy, robustness, and spatial fidelity [19]. Likewise, modern adaptive pooling strategies—such as Avg-TopK, detail-preserving pooling, or the learnable T-Max-Avg operator—show that traditional, fixed pooling rules can be extended to mitigate information loss while remaining lightweight [20].

2.3.1 Max pooling as selective evidence routing

Given a feature map $X \in \mathbb{R}^{H\times W}$ and a local pooling window Ω_u with stride s, max pooling computes

$$y(u) = \max_{v\in\Omega_u} X(v).$$

This hard selection mechanism routes gradients exclusively toward the argmax locations during backpropagation:

$$\frac{\partial \mathcal{L}}{\partial X(v)} = \mathbf{1}\Big[v = \arg\max_{w\in\Omega_u} X(w)\Big] \cdot \frac{\partial \mathcal{L}}{\partial y(u)}.$$

As a result, learning focuses on the most salient activations. Zafar et al. [19] show that max pooling performs strongly on datasets dominated by sparse, high-contrast features (e.g. MNIST, CIFAR-10), where sharp edges and strong local cues are especially informative.
However, the exclusivity of max pooling has notable drawbacks. Sub-maximal responses are entirely discarded, which can erase subtle textures and distributed evidence—an issue particularly relevant in fine-grained recognition and small-object

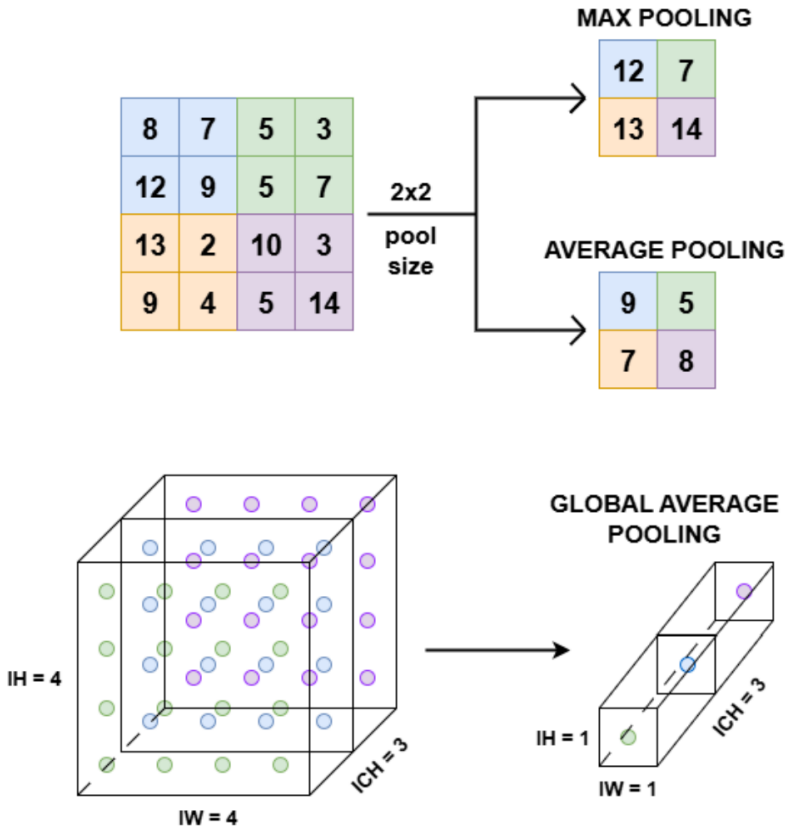

Figure 2.8: Comparison of max pooling, average pooling, and global average pooling.

localization. Max pooling also combines non-linear selection with subsampling, which can introduce aliasing when high-frequency content is present.

2.3.2 Global average pooling as a structural regularizer

Global average pooling collapses each feature map into a single scalar:

$$z_c = \frac{1}{HW} \sum_{v} X_c(v),$$

producing a vector $z \in \mathbb{R}^C$ summarizing the global activation per channel. GAP eliminates the need for fully connected classification layers, substantially reducing the number of parameters and the risk of overfitting. Because GAP measures the overall presence of high-level features, it enforces a strong inductive bias: each channel behaves like a class-specific detector whose spatial location is irrelevant.

This spatial invariance is beneficial for classification tasks and improves model interpretability through class activation mapping techniques. However, it also discards fine spatial structure. On tasks involving precise localization, rare local cues, or strong positional dependencies, GAP may underperform relative to max pooling or hybrid approaches.
Importantly, GAP distributes gradients uniformly across the entire feature map: every location receives a share proportional to $\frac{1}{HW}$. This leads to stable training dynamics and smooth representations, as documented in several comparative studies [19].

2.3.3 Architectural considerations and training dynamics

The placement and type of pooling influence both representational quality and gradient flow. In early layers, where high-frequency features dominate, aggressive striding paired with max pooling may remove essential detail; substituting smaller windows, stride-1 pooling, or strided convolutions often improves performance. In deeper layers, max pooling supports the propagation of decisive local evidence, while GAP serves as an effective global summarizer at the classifier interface.
From an optimization standpoint, max pooling yields sparse gradient paths, which can sharpen feature detectors but may slow the learning of near-miss features. In contrast, average-based pooling provides dense gradient assignment, promoting smoother feature maps but potentially suppressing rare discriminative peaks.
Evidence from Zafar et al. [19] and Zhao & Zhang [20] confirms that lightweight learnable pooling can improve both accuracy and robustness without increasing computational cost.

2.4 Convolutional Neural Network: Definition and General Architecture

A *Convolutional Neural Network* (CNN) is a specialized type of deep neural network primarily designed for processing grid-like data structures such as images, videos, and time series. CNNs extend the classical concept of *feedforward neural networks* by exploiting the spatial and temporal correlations that exist within structured data [1]. The key design principle behind CNNs is inspired by the organization of the visual cortex in biological systems, where neurons are sensitive to local receptive fields that capture specific spatial hierarchies of visual stimuli [2]. This biological analogy is reflected in the convolutional layer, where local connections and weight sharing enable the model to detect hierarchical patterns ranging from simple edges to complex objects.

2.4.1 Structural Overview

A typical CNN architecture is composed of multiple types of layers, each with a distinct computational role:

1. **Convolutional Layers** — perform local feature extraction;
2. **Activation Layers** — introduce non-linearity to the model;
3. **Pooling (or Subsampling) Layers** — perform spatial downsampling and feature compression;
4. **Fully Connected Layers** — integrate the extracted features for classification or regression tasks.

The arrangement of these layers enables CNNs to learn hierarchical feature representations: low-level filters capture edges and textures, mid-level filters capture shapes and motifs, and high-level filters represent semantic object parts.

2.4.2 Convolutional Layers

The convolutional layer is the fundamental building block of CNNs. In general terms, convolution is a linear operation that measures the similarity between an input signal and a localized filter or kernel, producing an output that emphasizes the presence of specific patterns. This operation can be interpreted both as a sliding dot product in the spatial domain and as a matched-filtering process in the signal-processing sense [21].
Given an input image $x \in \mathbb{R}^{H \times W \times C_{in}}$, the convolutional layer applies a set of K learnable filters $w_k \in \mathbb{R}^{m \times n \times C_{in}}$ (where m and n denote the kernel size) to produce K output feature maps:

$$y_k = f(w_k * x + b_k),$$

where $*$ denotes the discrete convolution, b_k is a bias term, and $f(\cdot)$ is a nonlinear activation function. Each convolutional kernel operates on a local region (the *receptive field*) of the input tensor, ensuring spatial locality and greatly reducing the number of parameters compared to fully connected layers.
In two dimensions, convolution generalizes to:

$$Y(m,n) = \sum_i \sum_j X(m+i, n+j)\, W(i,j), \tag{2.6}$$

where X is the input image, W the kernel, and Y the resulting feature map. The operation is repeated for multiple kernels, each extracting a distinct feature such as an edge, corner, or texture pattern [22].

Weight sharing across spatial locations enables CNNs to maintain translational invariance, i.e., the ability to recognize an object regardless of its position within the input frame. This property makes CNNs particularly effective for image recognition tasks, where patterns may appear in various regions of an image [19].

Physical Interpretation and Matched Filtering Perspective

From a signal-processing standpoint, convolution acts as a matched filter—a mechanism designed to detect specific features embedded in a signal by maximizing their correlation. As shown in [21], the convolution of an input signal $x(n)$ with a kernel $w(-n)$ is equivalent to a matched-filtering operation:

$$y(n) = x(n) * w(-n) = \sum_{m} x(n+m)\, w(m). \tag{2.7}$$

This process slides a feature-shaped kernel across the signal, producing a strong response whenever the kernel aligns with a similar pattern in the data. Thus, CNN filters can be interpreted as trainable matched filters that learn to recognize data-driven features invariant to translation and small deformations. This physical analogy provides an intuitive view of convolution as a feature-detection mechanism rather than a purely mathematical transformation.

Convolution in CNN Architectures

In neural networks, convolutional layers replace the dense, fully connected layers traditionally used in shallow architectures. Each convolutional layer consists of multiple filters whose parameters (weights and biases) are optimized during training. Unlike fully connected layers, convolutional layers exploit local spatial correlations by applying the same set of weights across all spatial positions, offering two key advantages: (i) a significant reduction in the number of parameters, and (ii) translational invariance of the learned features [22].
The output of a convolutional layer is a set of feature maps encoding various aspects of the input, such as edges in early layers or object parts in deeper layers. As shown in [23], this process naturally extends to one-dimensional data, where convolution is used for time series and spectral signals, effectively capturing temporal dependencies. Their analysis highlights the distinction between causal and non-causal convolution, as well as the influence of stride and padding in controlling output size and receptive field.

2.4.3 Activation Layers

Non-linear activation functions are essential for enabling CNNs to model complex mappings beyond linear transformations. Without them, the composition of

multiple linear layers would still result in a linear function, severely limiting the model's expressive capacity. The most commonly used activation is the Rectified Linear Unit (ReLU), defined as:

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

which introduces sparsity in activations and accelerates convergence by mitigating the vanishing gradient problem [2]. Other variants, such as Leaky ReLU, sigmoid, and hyperbolic tangent (*tanh*), are sometimes employed depending on the task and architecture.

2.4.4 Pooling Layers

Pooling layers are used to progressively reduce the spatial dimensions of the feature maps while retaining their most informative content. Given an input feature map $Y \in \mathbb{R}^{H \times W}$, a pooling operation outputs a downsampled representation:

$$Y'_{i,j} = \text{pool}\left(Y_{p \cdot i : p \cdot i+q,\, p \cdot j : p \cdot j+q}\right),$$

where pool$(\cdot)$ is typically either a maximum (*max pooling*) or average (*average pooling*) operator, p is the stride, and q is the pooling window size [19]. This operation enhances the model's robustness to small translations, rotations, and distortions, while reducing computational complexity and overfitting.

2.4.5 Fully Connected Layers

At the end of the convolutional pipeline, feature maps are flattened and fed into one or more fully connected (dense) layers. These layers perform high-level reasoning by combining the extracted features to form the final prediction. Mathematically, the fully connected layer performs a standard affine transformation:

$$z = f(Wx + b),$$

where W and b denote the weight matrix and bias vector, respectively, and $f(\cdot)$ is a non-linear activation.
Although fully connected layers contribute the majority of parameters in a CNN, they can often be replaced or reduced by *global average pooling (GAP)* layers, improving computational efficiency and interpretability — a design choice popularized in architectures such as ResNet and MobileNet [24, 1].

2.4.6 Forward and Backward Propagation

During the forward pass, each layer transforms its input feature maps into higher-level representations. The backward pass, based on the backpropagation algorithm,

computes gradients of the loss function with respect to each parameter using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_k} = \frac{\partial \mathcal{L}}{\partial y_k} \cdot \frac{\partial y_k}{\partial w_k}.$$

This process allows the network to iteratively update parameters using optimization algorithms such as Stochastic Gradient Descent (SGD) or Adam.
Residual connections, introduced later in ResNet [24], mitigate the vanishing gradient problem by providing shortcut paths for gradient flow, enabling deeper and more stable CNN architectures.

2.4.7 Architectural Considerations for Hardware Implementation

From a hardware perspective, the modular and parallelizable nature of CNNs makes them well-suited for deployment on *Field-Programmable Gate Arrays* (FPGAs). The repetitive and highly parallel convolution operations can be efficiently mapped to reconfigurable logic blocks, while techniques such as *loop unrolling*, *quantization*, and *template-based layer design* allow for optimization of latency, throughput, and energy consumption. In the context of this thesis, a templated CNN structure facilitates compile-time customization of convolutional kernels, feature map dimensions, and data precision, enabling a balance between accuracy and hardware efficiency.

2.4.8 Historical Evolution and Key Models

The evolution of CNNs began with LeNet-5 (LeCun et al., 1998), followed by AlexNet (Krizhevsky et al., 2012), which won the 2012 ImageNet Challenge and dramatically reduced classification error. Subsequent architectures, such as VGGNet and GoogLeNet, deepened the network and improved feature abstraction. However, very deep architectures suffered from vanishing gradients and performance degradation problems [24].

2.4.9 Residual Networks (ResNet)

He et al. (2016) introduced *Residual Networks (ResNet)*, which revolutionized the training of deep networks through the introduction of *residual blocks*. Instead of directly learning an underlying mapping $H(x)$, a residual block learns a residual function:

$$F(x) = H(x) - x \quad \Rightarrow \quad H(x) = F(x) + x$$

The use of *skip connections* (or identity mappings) allows direct information flow across layers, mitigating vanishing gradient issues and enabling efficient training of networks exceeding hundreds of layers [24].

Zaeemzadeh et al. (2018) further demonstrated that these identity shortcuts preserve gradient norms during backpropagation, improving optimization stability and convergence — a property known as *norm preservation* [25]. These theoretical insights explain why architectures such as ResNet-18, used in this project, offer an effective balance between depth, stability, and computational cost.

2.4.10 Modern Extensions of CNNs

Modern architectures extend CNN capabilities through additional mechanisms. Hu et al. (2019) proposed *Squeeze-and-Excitation Networks (SENet)*, which recalibrate channel-wise feature responses via adaptive attention, enhancing representational power at minimal computational overhead [26]. Other architectures, such as MobileNet and EfficientNet, introduced depthwise separable convolutions and compound scaling, improving efficiency and enabling deployment on embedded or FPGA-based systems — a goal aligned with this thesis.

2.5 FPGA Platforms for Convolutional Neural Network Acceleration

Field-Programmable Gate Arrays (FPGAs) have become increasingly relevant for the acceleration of Convolutional Neural Networks (CNNs), offering a unique combination of fine-grained parallelism, architectural flexibility, and energy efficiency. Unlike GPUs, which rely on fixed SIMD architectures, FPGAs allow designers to customize datapaths, parallelism strategies, and memory hierarchies to the specific computational patterns of CNNs. Recent surveys confirm that FPGA-based accelerators have matured significantly, becoming competitive with conventional GPU solutions for both edge and cloud inference workloads [27, 28].

2.5.1 Computation Structure of CNNs on FPGA Architectures

The core operation of a CNN is the discrete convolution, which applies a set of learnable kernels to an input feature map to extract spatial and semantic information. This process consists of cascaded multiply–accumulate (MAC) operations arranged in highly regular but deeply nested loops. Such regularity exposes several levels of parallelism, which FPGAs can exploit by distributing independent MAC operations across DSP blocks or by constructing deeply pipelined processing elements tailored to the kernel dimensions.
Architectures such as ZynqNet exemplify how convolutional layers can be mapped onto FPGA logic through aggressive loop unrolling and careful management of

on-chip memory resources [29]. In this design, each convolutional layer is expressed as a streaming computation in which input pixels flow through a sequence of pipelined operators. The high degree of pipelining allows new output values to be produced every cycle once the pipeline is full, significantly reducing latency compared to non-streaming approaches.

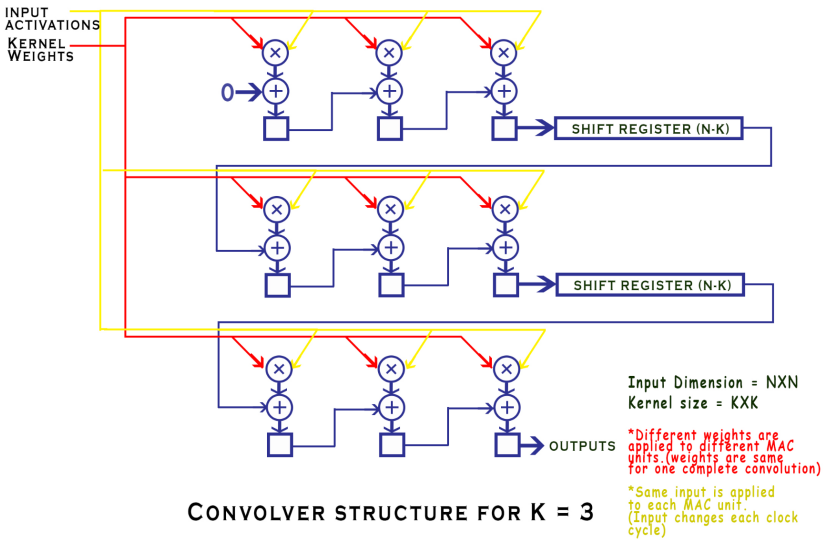

Figure 2.9: An illustration of a pipelined 2D convolution engine [30].

2.5.2 Memory Hierarchies and Dataflow Optimization

One of the most significant design challenges in FPGA-based CNN accelerators concerns memory bandwidth and data reuse. While FPGAs offer substantial on-chip memory resources through BRAM and URAM blocks, their off-chip bandwidth is usually far lower than that of high-end GPUs. As a consequence, efficient CNN accelerators must minimize external memory accesses and maximize data locality. To address this, designers often rely on deeply pipelined dataflow architectures in which data is continuously streamed through computation units. For instance, the Systolic-CNN architecture [31] employs a systolic array of processing elements that maintains a uniform flow of activations and weights, ensuring that the compute fabric remains fully utilized and that memory stalls are minimized. This approach demonstrates how a synchronous dataflow design can significantly increase throughput and energy efficiency, especially when compared to architectures that repeatedly fetch data from external memory.

Binary and low-precision CNNs provide an additional mechanism to reduce memory pressure. By encoding weights and activations using only one or a few bits, the

memory footprint decreases dramatically, allowing more data to fit on-chip and enabling wider parallelism. Li et al. show that binary neural networks on FPGA can, in some cases, outperform GPUs because they reduce both the computational and memory requirements of convolutional layers [32]. This demonstrates how algorithmic simplification and hardware specialization can synergize particularly well on reconfigurable platforms.

2.5.3 Precision Reduction and Arithmetic Customization

One of the defining capabilities of FPGAs is support for customized numerical formats. Whereas GPUs are constrained to a small range of standardized types such as FP32, FP16, or INT8, FPGAs allow designers to choose arbitrary bit widths for both integer and fixed-point representations. This fine-grained control enables the construction of accelerators that match the exact precision requirements of the target CNN, reducing both resource usage and power consumption.

The work of Wang [33] highlights how fixed-point arithmetic (typically between 8 and 16 bits) can maintain accuracy comparable to floating-point implementations while significantly improving efficiency. In contrast, fully binary CNNs further reduce resource demands by eliminating multipliers altogether, replacing them with inexpensive XNOR and popcount operations. The flexibility to integrate such custom datapaths at the hardware level is one of the main reasons why FPGAs remain competitive in domains requiring tight power and bandwidth budgets.

2.5.4 High-Level Synthesis and Modern FPGA Toolflows

Historically, FPGA-based accelerators were developed through RTL design in languages such as VHDL or Verilog. While this approach provides full control over timing and microarchitecture, it is time-consuming and requires considerable hardware expertise. The emergence of High-Level Synthesis (HLS) has significantly changed the development landscape. Tools such as Xilinx Vitis HLS, Intel OpenCL SDK, and domain-specific frameworks like FINN and hls4ml allow developers to describe CNN computations in C/C++ or specialized graph representations, leaving the generation of optimized RTL to the compiler.

The Systolic-CNN work [31] is a representative example of how modern HLS flows facilitate the construction of complex accelerators while maintaining high performance. This shift toward HLS drastically reduces development time and enables rapid design-space exploration, making FPGA acceleration more accessible to machine learning practitioners who may not have extensive hardware backgrounds.

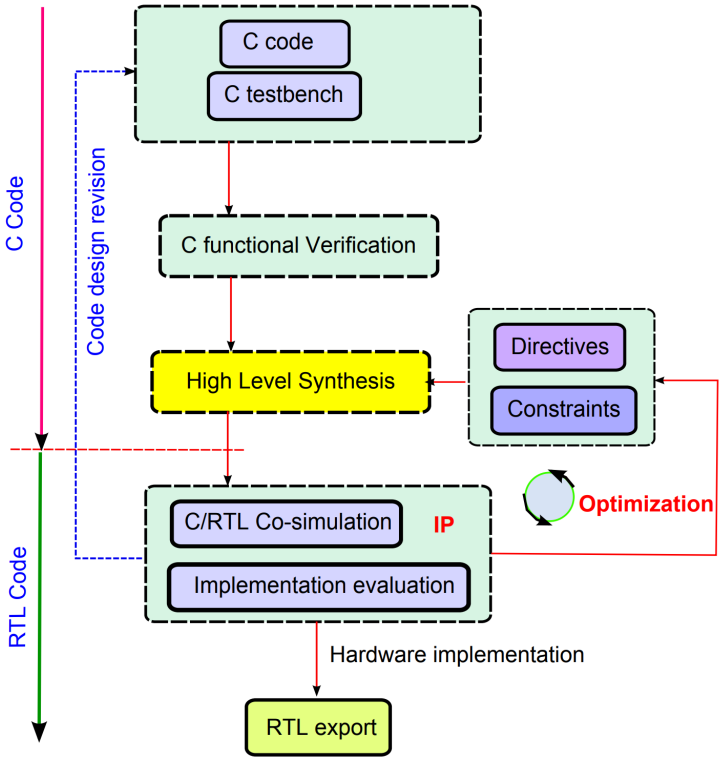

Figure 2.10: Visual summary of an HLS-based FPGA design flow [34].

2.6 High-Level Synthesis for CNN Acceleration on FPGA Platforms

2.6.1 Traditional RTL Approaches for CNN Acceleration

RTL-based design using VHDL or Verilog provides cycle-accurate control over datapaths, pipelining, and resource allocation. In the context of convolutional neural networks, RTL methodologies typically involve designing dedicated processing engines for convolutions, pooling, activation, and memory orchestration. Manually crafted RTL allows the designer to optimize kernel-specific pipelines, implement carefully tuned systolic arrays, and match timing constraints with precision. Implementations such as early FPGA CNN accelerators described in the literature demonstrate that handwritten RTL can meet stringent timing requirements through finely optimized control logic and exact scheduling of memory movements, but at the cost of significant design effort [35].
However, CNNs consist of deeply nested loops—over channels, filters, spatial

windows, and output positions—that require designers to explicitly manage loop unrolling, parallelism, and streaming. Implementing even a single convolutional layer in RTL can involve significant architectural specialization, making the design process brittle when adapting to model changes such as variations in:

- kernel size and number of filters,
- feature-map resolutions,
- quantization levels (FP32, FP16, INT8, fixed-point),
- pruning or structured sparsity patterns.

This inflexibility makes classic RTL development particularly costly for modern CNN designs, which evolve rapidly. Furthermore, every architectural modification—such as introducing approximate MAC units or adopting low-precision arithmetic—requires non-trivial RTL restructuring and verification overhead.

2.6.2 High-Level Synthesis Methodology

High-Level Synthesis (HLS), and in particular AMD/Xilinx Vitis HLS, enables designers to specify CNN computations in C/C++ while instructing the compiler using pragma directives to generate pipelined and parallelized RTL. HLS exposes the same hardware optimization mechanisms as RTL—loop pipelining, loop unrolling, array partitioning, bit-accurate fixed-point arithmetic—but allows them to be controlled declaratively.
HLS abstracts away many low-level timing considerations, allowing rapid exploration of alternative pipeline depths, memory tiling strategies, and parallelization degrees. Sudvarg et al. demonstrate how the insertion of HLS pragmas (e.g. `#pragma HLS PIPELINE`, `UNROLL`, and `DATAFLOW`) can improve performance by three orders of magnitude in complex data-processing pipelines, with over 3400× speedup observed between naïve and optimized versions [36]. This illustrates how HLS performance is highly sensitive to pragma-level tuning, but also how it dramatically reduces development time compared to an equivalent RTL optimization effort.

2.6.3 Stream-Based CNN Implementations in HLS

The hls4ml framework provides a detailed example of applying HLS to convolutional architectures. Aarrestad et al. extend hls4ml to include streaming convolutional layers that avoid materializing the full im2col matrix, reducing memory overhead and enabling fully on-chip processing [37]. Key design elements include:

- use of `hls::stream<>` FIFOs to buffer sliding windows,

- K^2 parallel streams to maintain kernel-aligned window state,
- single-cycle access to column vectors for GEMM-like processing,
- sequential feature-map ingestion with minimal BRAM pressure.

This design meets microsecond-scale latency constraints required in CERN Level-1 trigger pipelines by exploiting fine-grained pipelining across convolution windows and decoupling memory flow from compute flow. The approach highlights the strength of HLS for expressing dataflow architectures that would require substantial, error-prone RTL code if implemented manually.

2.6.4 Precision Optimization and Approximate Arithmetic

CNN accelerators rely heavily on MAC operations, making numerical representation a critical design dimension. HLS toolchains enable flexible exploration of reduced or custom precision formats. Cho and Kim propose an approximate fixed-point MAC unit implementable entirely via LUT-based logic rather than DSP blocks, achieving:

- 78% reduction in DSP usage,
- 66% reduction in memory footprint,
- nearly 50% latency reduction compared to FP32,

while maintaining sufficient accuracy for LeNet-5 classification [38]. These optimizations exploit HLS-level bit manipulation functions and C-based representations of fixed-point data types, which greatly reduce the implementation complexity compared to hand-written RTL equivalents.

2.6.5 Quantization and Compiler-Assisted Mapping with Vitis-AI

In contrast to manual RTL design, the Vitis-AI toolchain automates several steps of the optimization pipeline:

- post-training quantization to INT8,
- layer fusion (e.g. Conv+BN folding),
- instruction scheduling for the Deep Learning Processing Unit (DPU),
- deployment of the compiled `.xmodel` to MPSoC platforms.

Li et al. show that this HLS-compatible workflow enables CNNs (52 Conv2D layers in their evaluation) to achieve 3.33–5.82× higher throughput and 3.39–6.30× higher energy efficiency on a Zynq UltraScale+ ZCU104 board compared to CPU/GPU execution [35]. Because Vitis-AI internally generates optimized RTL for the DPU, the developer focuses primarily on quantization and network construction rather than hardware micro-details.

2.6.6 HLS Dataflow Architectures and Comparison with RTL

Beyond CNN workloads, HLS excels in applications dominated by dataflow parallelism. Kapetanakis et al. demonstrate that an HLS-based FEM solver—optimized through task-level parallelism, memory-aware pipelining, and initiation interval minimization— achieves:

- 7.9× higher performance than optimized Vitis-HLS baselines,
- 45% latency reduction versus CPU execution,
- 3.64× lower power consumption,

on an Alveo U200 accelerator [39]. Their results show that HLS-generated architectures can reach or exceed the efficiency of manually optimized RTL when the computation is inherently pipeline-friendly. CNN inference maps naturally to this class of workloads: convolution, activation, and pooling layers all follow streaming dataflow patterns.

2.6.7 Comparison Summary: RTL vs. HLS for CNN Acceleration

A qualitative comparison between traditional RTL methodologies (VHDL/Verilog) and High-Level Synthesis (HLS in C/C++) reveals several fundamental differences in design philosophy, workflow efficiency, and scalability. In terms of microarchitectural control, RTL offers complete, explicit management of pipelines, resource allocation, and signal timing. Every structural detail of the hardware must be manually defined by the designer. By contrast, HLS automatically generates the underlying RTL from a higher-level algorithmic description, with optimization guided through pragma directives. This greatly reduces low-level effort while enabling rapid design iterations, albeit at the cost of less fine-grained control.
Flexibility in responding to model changes represents another major point of divergence. CNN architectures evolve frequently, with adjustments to kernel sizes, feature-map dimensions, quantization formats, or sparsity patterns. In an RTL

workflow, such modifications usually require significant redesign effort. HLS, on the other hand, allows the same C/C++ codebase to be adapted with minimal intervention, often by adjusting pragma settings or restructuring a limited portion of the algorithmic description.

The difference in development time is equally substantial. RTL design cycles are typically long, often spanning weeks or months for complex accelerators, whereas HLS enables functionally equivalent designs to be produced within days or even hours. This acceleration in workflow is one of the primary motivations for adopting HLS in the context of fast-evolving machine learning applications.

Regarding the exploration of numerical precision, RTL requires manual and error-prone handling of bit widths and custom fixed-point formats. HLS natively supports arbitrary-precision integer and fixed-point types, allowing designers to experiment with low-bitwidth arithmetic and quantization-friendly representations with minimal implementation overhead.

Dataflow optimization also highlights a clear methodological distinction. In RTL, expressing a streaming architecture requires explicit finite state machines, hand-crafted scheduling of memory accesses, and careful coordination of pipeline stages. HLS provides built-in constructs such as `DATAFLOW` and `hls::stream` types, which enable a more natural expression of pipelined and parallel execution models, particularly well suited to CNN workloads.

Finally, scalability across different FPGA architectures tends to favor HLS. While RTL designs are often tightly coupled to specific device characteristics, HLS descriptions exhibit higher portability, with the toolchain handling most of the device-specific adaptation. In summary, RTL offers maximum performance potential but requires significant engineering effort, whereas HLS provides near-optimal performance with much shorter development time and superior adaptability to modern machine learning requirements.

Chapter 3

NEUROKIT2E and the Aidge Framework

3.1 The Role of CEA Paris–Saclay

The CEA Paris–Saclay centre is one of the key research hubs of the French Alternative Energies and Atomic Energy Commission. It hosts several programmes dedicated to microelectronics, embedded systems and artificial intelligence, and it plays a central role in European collaborative projects. Among these, *NEUROKIT2E* occupies a distinctive position. Coordinated by CEA, the project aims to build a sovereign European ecosystem for embedded AI by developing tools and methodologies that allow the design and deployment of neural networks on highly constrained hardware platforms. The initiative emerges from the broader ambition of strengthening Europe's competitiveness in a field that is rapidly becoming strategic for industrial innovation.

3.2 NEUROKIT2E: Objectives and Context

NEUROKIT2E is funded by the European Union through the Horizon Europe *Chips Joint Undertaking*. It gathers a consortium from five member states and brings together research institutions, universities and industrial actors involved in semiconductors and embedded intelligence. The project was conceived in response to two complementary needs. On one hand, the adoption of AI is rapidly moving from cloud computing toward edge devices, where computations must be executed under strict constraints of energy, latency and memory. On the other hand, most of the toolchains currently used for neural network development are dominated by non–European ecosystems, which can limit technological autonomy and long–term

industrial independence.
In this context, NEUROKIT2E proposes the creation of an open–source platform dedicated to deep learning on embedded hardware. The platform is meant to offer a complete workflow, from model design to hardware deployment, while ensuring compatibility with the diversity of European hardware technologies. It builds on previous developments at CEA, most notably the *Neural Network Design & Deployment* (N2D2) framework, and extends them toward a more structured and industrial-grade environment.

3.3 From N2D2 to Aidge

The transition from N2D2 to *Aidge* reflects a broader evolution in both industrial expectations and technological requirements. N2D2 was initially designed as a research tool, capable of modelling neural networks and deploying them onto embedded architectures. Over the years, however, its user community expanded beyond research laboratories, and the need emerged for a more comprehensive and modular framework. Aidge was introduced as the answer to this need.
Developed within the NEUROKIT2E project and maintained at CEA–List, Aidge reorganises the concepts introduced by N2D2 into a modern, extensible structure. It remains open–source, but its architecture is more robust, allowing it to integrate new operators, new optimisation methods and new hardware targets. Importantly, it is designed from the outset to cooperate with standard AI ecosystems, so that models trained in frameworks like PyTorch or Keras can be imported, analysed, transformed and deployed through Aidge with minimal effort.

3.4 Aidge: Philosophy and Capabilities

The official CEA–List description of Aidge presents it as a collaborative framework that provides a complete toolchain for the design, optimisation and deployment of neural networks in constrained environments. The emphasis is not only on performance, but also on transparency and controllability. In contrast to cloud-oriented AI platforms, Aidge is meant to operate close to the hardware, giving developers full visibility over the transformations applied to their models.
Aidge offers a range of features that support this philosophy. It includes mechanisms for graph-level manipulation, such as tiling or structural rewriting, and integrates several families of optimisation techniques. Quantisation plays a particularly central role: both post-training quantisation and quantisation-aware training are available, enabling models to be adapted to fixed-point arithmetic without excessive loss in accuracy. Pruning, compression and other forms of model simplification

complement these techniques. Aidge also provides a deployment flow that supports microcontrollers, CPUs, GPUs, dedicated accelerators, ASICs and FPGAs.

Beyond technical tools, Aidge is designed as a cornerstone of a European ecosystem. The framework is hosted by the Eclipse Foundation, ensuring an open and neutral governance model. CEA–List emphasises that Aidge is part of a broader strategy, supported by initiatives such as DeepGreen and NEUROKIT2E, to establish a sovereign alternative to non–European AI platforms and to support the development of advanced digital technologies in Europe.

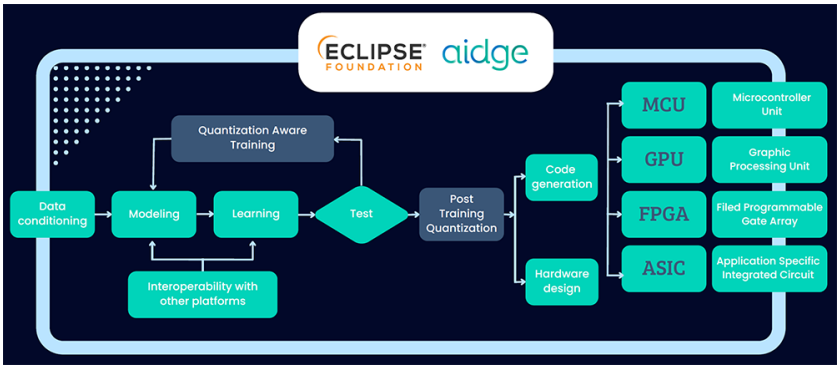

Figure 3.1: Artificial Intelligence Development Flow on Embedded Hardware [40].

3.5 A Unified View: How NEUROKIT2E and Aidge Interact

Although NEUROKIT2E and Aidge can be described separately, their relationship is fundamental. NEUROKIT2E defines the overall vision: it identifies use cases, hardware requirements, application constraints and performance objectives. Aidge embodies this vision by translating it into a concrete software framework.

The project's use cases — which range from industrial inspection to smart infrastructure, automotive safety or assistive technologies — provide an ideal testbed for validating Aidge's features. Networks are first designed and trained using common AI tools, then imported into Aidge, where they undergo a sequence of optimisations tuned for the target hardware. Once transformed, they can be exported as optimised C code, FPGA-ready modules or accelerator-specific binaries. Evaluations performed by industrial partners then inform the next iteration of the design, creating a feedback loop between the conceptual platform (NEUROKIT2E) and the operational framework (Aidge).

3.6 Thesis Contribution Within the Aidge Context

Within this ecosystem, the work carried out in this thesis focuses on the integration of custom FPGA-oriented modules into the Aidge deployment flow. Using High-Level Synthesis, a set of reusable templated functions was developed to implement the core operations of convolutional neural networks. Attention was given to memory organisation, which is often the limiting factor in FPGA deployments. A strategy based on input reuse was first explored, taking advantage of on-chip memory to minimise external data transfers. As the network grows deeper, however, kernel storage becomes more prominent, and a hybrid approach was adopted to balance input reuse in the early layers with kernel reuse in the later stages.
This methodology was validated by mapping several layers of a ResNet–18 model onto the FPGA and analysing the resulting resource utilisation. The experiments demonstrated that HLS-based modules can be effectively integrated within the Aidge workflow, providing a viable path for accelerating neural networks on embedded hardware with strict performance and energy constraints.

Chapter 4

Design and Hardware Implementation of CNNs in HLS

In this section, the hardware implementation of convolutional neural networks (CNNs) is described using High-Level Synthesis (HLS). HLS enables FPGA circuits to be designed directly from high-level C++ descriptions, significantly reducing development time and allowing rapid design-space exploration compared to traditional hand-written RTL. The implementation focuses primarily on ResNet-18, one of the most widely used CNN architectures in computer vision. Particular attention is dedicated to the key operations forming the implemented layer and to the dataflow structure that orchestrates the exchange of data among them. The developed functions are general enough to support additional networks, including LeNet; however, due to its higher structural complexity, ResNet-18 was selected as the primary case study for this thesis.

4.1 ResNet-18

ResNet-18 belongs to the broader family of Residual Networks (ResNets), which introduce residual connections to alleviate the vanishing-gradient problem and enable the training of deeper architectures. The vanishing-gradient phenomenon occurs when gradients progressively diminish as they propagate backward through many layers, preventing earlier layers from receiving meaningful updates. ResNets address this issue by reformulating the learning task: rather than forcing each block to approximate a direct mapping $H(x)$, the network learns a residual function

$$F(x) = H(x) - x,$$

and produces the block output

$$y = x + F(x),$$

via an identity skip connection. These shortcuts preserve gradient flow, ease optimization, and allow deeper, more expressive networks to be trained effectively. Figure 4.1 illustrates a standard residual block with an identity skip connection.

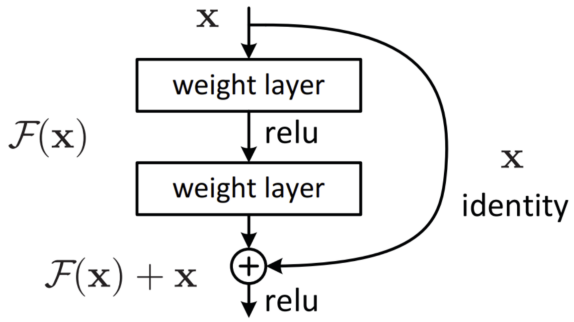

Figure 4.1: Residual block with identity skip connection [41].

Following an analysis of memory-partitioning strategies, the core operators of ResNet-18 were implemented in HLS. Figure 4.2 shows a block-level diagram of the network. The study focuses in particular on the final residual stage, which comprises 512 filters and represents one of the most computationally intensive segments of the architecture.

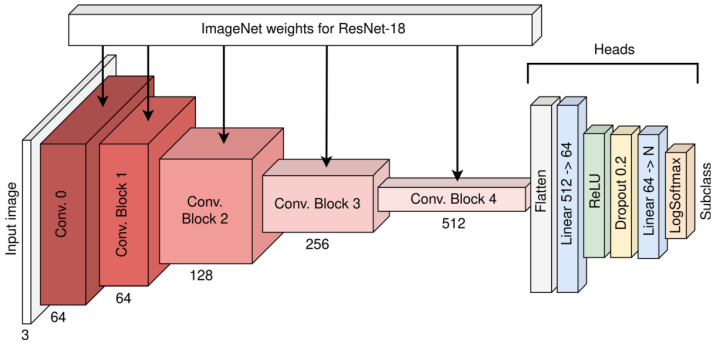

Figure 4.2: Overall architecture of ResNet-18 [42].

4.2 Convolution

The operating principles of convolution were introduced in 2.1. Within ResNet-18, the convolution operator corresponds to a standard multi-channel, multi-filter

2D convolution: the kernel slides along the spatial dimensions (H and W) while spanning the entire input-channel depth. Figure 4.3 illustrates such an operation with C input channels and M output channels. Each filter aggregates weighted sums across all input channels and generates one output-channel feature map. Since the network contains M distinct filters, the output tensor has dimensions M × Q × P.

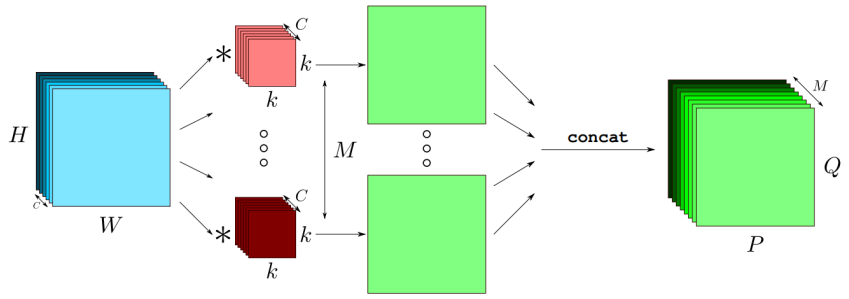

Figure 4.3: Multiple Channel Multiple Kernel (MCMK) Convolution [43].

The spatial dimensions of the output feature map are determined by:

$$\mathrm{P} = \frac{\mathrm{W} + 2 \times \mathrm{PADDING} - \mathrm{FW}}{\mathrm{STRIDE}} + 1, \tag{4.1}$$

$$\mathrm{Q} = \frac{\mathrm{H} + 2 \times \mathrm{PADDING} - \mathrm{FH}}{\mathrm{STRIDE}} + 1, \tag{4.2}$$

where W and H denote the spatial dimensions of the input feature map, FW and FH are the kernel dimensions, and STRIDE and PADDING control the sampling pattern of the convolution. These two parameters play a fundamental role in both computational cost and spatial-resolution management and are therefore carefully integrated into the HLS implementation.

4.2.1 Stride

The stride parameter defines the step size with which the convolution kernel moves across the input. Convolution typically begins with the kernel positioned at the top-left of the input tensor and proceeds by sliding horizontally and vertically. A stride of one corresponds to the kernel shifting by a single element at each step, resulting in densely sampled feature maps. Larger strides are often used either to downsample the spatial resolution or to reduce computational cost by skipping intermediate positions.
Figure 4.4 compares convolutions with stride 1 and stride 2, showing the resulting changes in output size. As observed in the previous equations, stride and output

resolution are inversely related. Increasing the stride reduces the number of operations performed, improving efficiency, but may cause fine-grained features to be lost due to the coarser sampling.

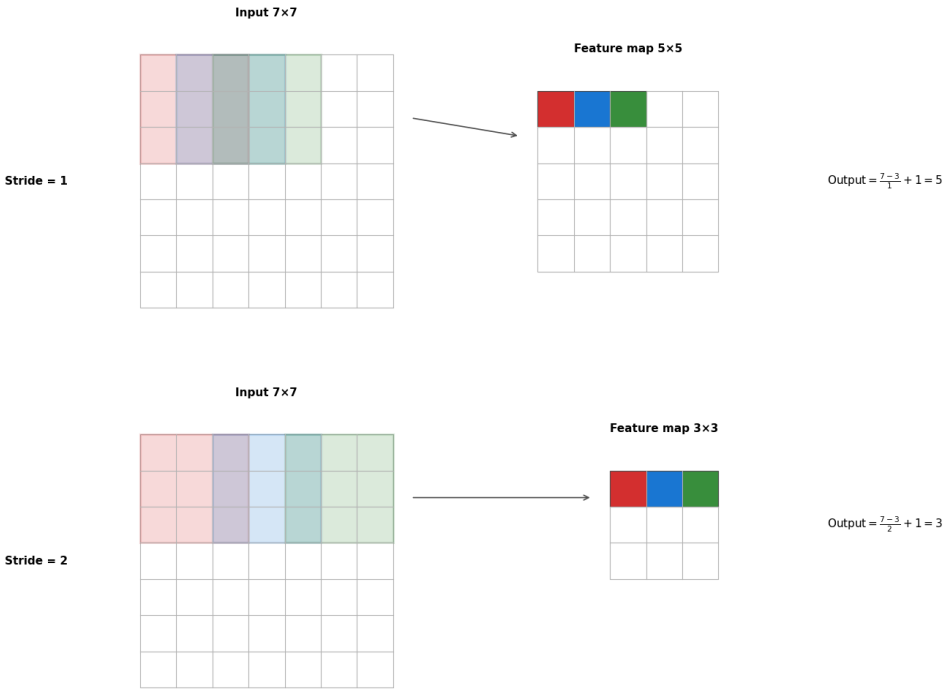

Figure 4.4: Effect of stride on convolution output resolution.

4.2.2 Padding

Without padding, output dimensions tend to shrink as the kernel cannot be applied near the borders without reducing the spatial coverage. This leads to decreasing feature-map sizes across successive layers and may cause the loss of important spatial information, especially near the boundaries of the image. Padding mitigates this issue by artificially extending the input with additional pixels—typically zeros—along its borders. This ensures that the kernel can be applied uniformly across the entire spatial domain.
Two common padding modes are used in CNNs:

- **Valid padding ("no padding")**: no pixels are added; the output becomes smaller than the input.
- **Same padding**: zero-padding is applied so that the output maintains the same spatial dimensions as the input.

Figure 4.5 illustrates an example in which a padding of one is applied to a 5×5 input, ensuring complete kernel coverage.

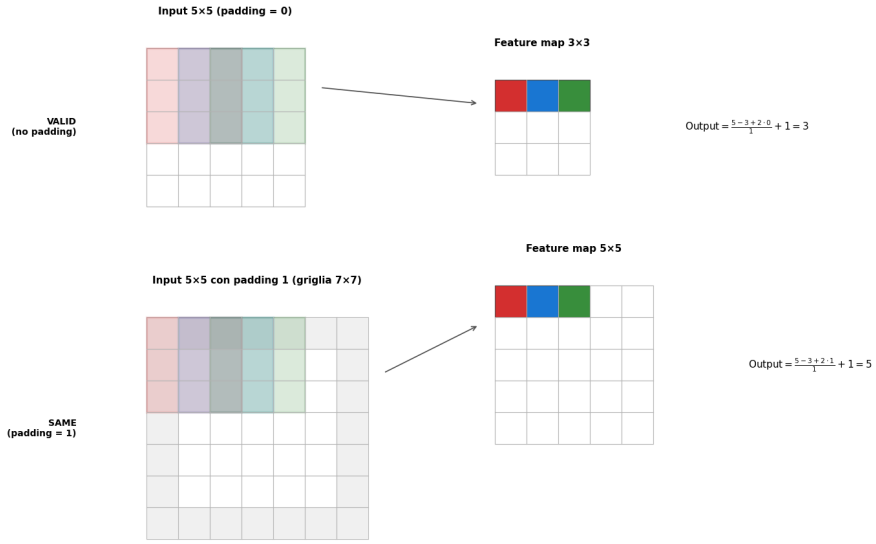

Figure 4.5: Example of applying padding to preserve spatial dimensions.

4.2.3 Bias

Each output channel includes an additive bias term. The bias is added after the convolution sum has been accumulated. This operation is lightweight and easily pipelined within the convolution function.
The bias values are stored in BRAM or LUTRAM depending on availability and are accessed sequentially during computation.

4.2.4 Template-based Convolution Interface

The `conv` operator is implemented as a highly parameterised C++ template, so that a single source definition can be specialised at compile time for different layers and even different networks. The template signature exposes both the logical tensor dimensions and the architectural choices made for the HLS implementation:

```
template<
    int ICH,          // input channels
    int IW,           // input width
    int IH,           // input height
    int FW,           // filter width
    int FH,           // filter height
    int OCH,          // output channels
    int OW,           // output width
    int OH,           // output height
    int ICH_PAR,      // input-channel parallelism
    int STRIDE,       // convolution stride
    int WINDOW_IN,    // input tiling factor
    int ICH_PAR_OUT,// output-channel parallelism
    int FW_OUT,       // output tile width
    int FH_OUT,       // output tile height
    int OCH_OUT,      // output tile channels
    int WINDOW_OUT, // output tiling factor
    int ReLU          // fused ReLU enable flag
>
void conv(
    hls::stream<conv_packet_t<FW, FH, ICH_PAR>> &conv_data_stream,
    filter_stream_t &filter_stream,
    const ap_int<32> bias[OCH],
    memory_out_conv_t<
        FW_OUT, FH_OUT,
        ICH_PAR_OUT, OCH_OUT,
        WINDOW_OUT
    > &out_mem
);
```

Listing 4.1: Template and function signature of the convolution operator

`ICH`, `IW`, `IH` denote the number of input channels, input width and height, respectively, while `FW`, `FH` define the spatial extent of the convolution kernel. The parameters `OCH`, `OW`, `OH` encode the number of output channels and the spatial dimensions of the output feature map. In addition to these shape parameters, the template includes several knobs related to parallelism and tiling. The parameter `ICH_PAR` specifies the degree of input-channel parallelism exploited in the inner loops: instead of processing one channel at a time, the kernel operates on blocks of `ICH_PAR` channels, which is reflected both in the internal buffering and in the input packet type `conv_packet_t<FW, FH, ICH_PAR>`. Similarly, `ICH_PAR_OUT` governs the degree of parallelism along the output-channel dimension, allowing multiple output channels to be accumulated concurrently. The parameters `FW_OUT`, `FH_OUT`, `OCH_OUT` and `WINDOW_OUT` describe how the output space is tiled into windows and how these tiles are arranged in the output buffer type `memory_out_conv_t<FW_OUT,`

FH_OUT, ICH_PAR_OUT, OCH_OUT, WINDOW_OUT>, facilitating integration with subsequent layers that expect a specific layout. Finally, STRIDE, WINDOW_IN and the boolean ReLU act as compile-time configuration flags: STRIDE encodes the sampling pattern chosen at the network level, WINDOW_IN can be used to describe how input tiles are produced by the previous stage, and ReLU enables or disables the fused activation at the end of the accumulation. On the interface side, the function consumes input feature windows from an hls::stream<conv_packet_t<FW, FH, ICH_PAR» (conv_data_stream), receives quantised filter coefficients through filter_stream_t& filter_stream (where each element is stored as ap_int<8>), and initialises the output accumulation with a per-channel bias array const ap_int<32> bias[OCH]. The final results are written into the templated memory object out_mem, which acts as a structured on-chip buffer or as an abstraction of the interface towards the next processing stage. By expressing all sizes, tiling factors and parallelism degrees as template parameters, the same conv definition can be instantiated for different convolutional layers in ResNet-18 and for other architectures such as LeNet, while allowing the HLS tool to fully specialise loops, unroll factors and memory partitioning at synthesis time.

Hardware Architecture

The hardware architecture generated for the convolution operator reflects the design choices introduced at the C++ level and materializes them into a set of cooperating hardware blocks synthesized by Vitis HLS. Rather than viewing the convolution as a monolithic unit, the implementation decomposes the computation into a sequence of micro–architectural elements that operate in a streaming fashion. This organization facilitates high throughput and enables the synthesis tool to schedule operations with fine-grained parallelism.

A central component of the architecture is the input interface, which ingests activation tiles from the conv_data_stream. The incoming data are already packed according to the template parameters (FW, FH, ICH_PAR), allowing the hardware to receive an entire convolutional window per cycle when the pipeline operates with II=1. As soon as a packet arrives, its values are distributed across a private set of on-chip registers or, when beneficial, across partitioned memory banks whose structure is determined automatically by the template configuration. This early demultiplexing step is crucial: by placing each element of the window in a distinct storage element, the architecture ensures parallel access to all operands during the multiply-accumulate phase.

The weights follow a similar principle. The filter_stream conveys quantised coefficients in a strictly defined order, and the hardware loads them into an internal filter_mem array. Template parameters such as ICH_PAR, ICH_PAR_OUT, and the kernel dimensions establish the precise layout of this array and dictate how many

weights can be consumed in parallel. Vitis HLS performs a complete partition of the weight buffer, effectively transforming the array into a set of independent registers. This enables the architecture to execute multiple inner-product operations simultaneously whenever the associated template parameters request increased parallelism.
Once both activations and weights are staged, the architecture performs the multiply-accumulate loop. The datapath typically comprises a tree of DSP blocks (or synthetically generated multipliers) feeding an accumulation register. The structure of this tree is inferred from the template parameters: a higher unrolling factor produces a wider adder tree and thus reduces latency, whereas more conservative parameters lead to a narrower but more resource-efficient configuration. At the end of the accumulation cycle, the partial sum is combined with the corresponding bias term, and, if enabled via the template parameter `ReLU`, passed through an integrated activation unit. Embedding the activation function inside the convolution block reduces the number of pipeline stages and simplifies downstream dataflow.
Finally, the computed outputs are written into the object `out_mem`, which represents a structured collection of output tiles. The physical organization of `out_mem` including the number of banks, their depth, and the addressing scheme is once again determined by the template. This abstraction allows the convolution module to be reused with different network topologies, as subsequent layers only need to adhere to the same parameterized interface rather than a fixed memory format.
Overall, the hardware architecture is thus not a static design but a template-driven blueprint capable of generating multiple specialized instances of the convolution engine. Each instance is optimized for the dimensions, parallelism level, and dataflow constraints of the target CNN layer, enabling seamless integration within the broader pipeline of the ResNet–18 implementation.

4.3 ReLU

The Rectified Linear Unit (ReLU) is a simple activation function defined as:

$$\mathrm{ReLU}(x) = \max(0, x).$$

Its hardware implementation is extremely efficient, requiring only a comparator and a multiplexer. The ReLU module is also templated and pipelined, ensuring an initiation interval (II) of 1.
ReLU is inserted inside the convolution function.

4.3.1 ReLU in the HLS Implementation

In many neural network architectures, a ReLU activation is typically inserted after each convolutional layer in order to introduce non-linearity; without such

activations, a series of convolutions would behave as a single linear transformation, limiting the expressive power of the network. In ResNet-18, however, as observed in the ONNX representation, a ReLU does not necessarily follow every convolution. For this reason, a template parameter was added to the convolution routine to specify whether the activation should be applied.
From an implementation standpoint, ReLU is very simple to compute, and it has been directly embedded inside the convolution function as shown below:

```
if (RELU) {
  if (sum < 0) {
    sum = 0;
    }
}
out_mem[s_mem_o][s_mem_o_depth] = sum;
```

Before storing the result into the partitioned output buffer, the value of the RELU template parameter is evaluated. When the parameter is enabled, negative accumulator values are set to zero; otherwise, the computed sum is written as-is. Placing the ReLU operation inside the convolution routine eliminates the need for a separate activation stage and reduces memory traffic between layers. Additionally, the implementation relies only on simple conditional logic, which the HLS compiler translates into comparators and multiplexers. Consequently, the initiation interval of the convolution kernel stays at one and is not influenced by the extra control logic.

4.4 Global Average Pooling Implementation

The final stage of the accelerator pipeline is the global average pooling (GAP) layer, which reduces each output channel to a single representative value by averaging all spatial positions of the corresponding feature map. Unlike convolution and quantization, whose behaviour depends on multiple template parameters and dataflow paths, the GAP stage is structurally simple but must be implemented carefully to maintain throughput and ensure compatibility with the internal memory layout adopted in previous stages.

4.4.1 Input Structure and Data Duplication

Before pooling is applied, the input tensor feeding the residual block is duplicated using the templated function `duplicate_input_stream`. Although this operation is mostly used earlier in the pipeline for handling the skip connection, it introduces a pattern that is conceptually similar to what GAP requires: the ability to broadcast

elements from a single input stream to multiple consumers. The duplication function reads one element per cycle from the source stream and writes it to two independent streams. This behaviour is enforced by the directive `#pragma HLS PIPELINE II=1`, which ensures that the streaming subsystem maintains an initiation interval of one and can process values at the full bandwidth of the accelerator.

4.4.2 Global Average Pooling Kernel

The core of the pooling stage is implemented in the templated function `global_avg_pool`, parameterized by the output tile dimensions (`FW_OUT`, `FH_OUT`), the channel-parallelism factor (`ICH_PAR_OUT`), the number of output channels `OCH`, and the spatial window size `WINDOW_OUT`. The design assumes a configuration where the packed input memory has width one:

$$\texttt{MEM\_WIDTH} = \texttt{FW\_OUT} \times \texttt{FH\_OUT} \times \texttt{ICH\_PAR\_OUT} = 1,$$

which matches the layout produced by the final quantization stage. Under this assumption, each spatial element of the tensor is stored in a different memory depth, making the design conceptually equivalent to a flat array of length

$$\texttt{MEM\_DEPTH} = \texttt{OCH} \times (\texttt{WINDOW\_OUT} \times \texttt{WINDOW\_OUT}).$$

This structure allows GAP to iterate over spatial positions in a predictable, linear order.
The function initializes one accumulator per channel. Since each accumulator is reused independently, it is fully partitioned using `#pragma HLS ARRAY_PARTITION`, allowing parallel access and enabling both loops over channels to be pipelined or unrolled without creating access conflicts. The pooling loop consists of two nested loops: the outer loop iterates over output channels, while the inner loop traverses all spatial positions for that channel. The inner loop is pipelined with `II=1`, ensuring that GAP reads one value per cycle from the input memory and updates the accumulator accordingly.
Access to the memory uses the layout

$$\texttt{value} = \texttt{mem\_in}[0][\texttt{spatial\_idx} \times \texttt{OCH} + \texttt{ch}],$$

which guarantees correct channel interleaving. During accumulation, values are promoted to 32-bit integers to prevent overflow. The total sum for each channel is then normalized by dividing by the spatial area. The implementation applies a rounding scheme equivalent to that used by NumPy and ONNX, achieved by adding `SPATIAL_SIZE`/2 before the division:

$$\texttt{mean} = \frac{\texttt{accumulator[ch]} + \texttt{SPATIAL\_SIZE}/2}{\texttt{SPATIAL\_SIZE}}.$$

The result is then cast back to `ap_int<8>` and stored in `gap_out[ch]`, which holds all channel-averaged values in a dense format.

4.4.3 Streaming of GAP Results

The final step converts the pooled values into an AXI4-Stream format, performed by the function `gap_to_stream`. For each channel, the function constructs a `mem_out_t` packet containing the 8-bit result as the data payload, sets all bytes as valid via `keep = -1`, and asserts the `last` flag only for the final channel, allowing downstream modules to detect the end of the stream. The loop is pipelined with `II=1`, ensuring that one output value is emitted every cycle and matching the rate of the previous stages in the accelerator.

4.4.4 Position of GAP in the Overall Pipeline

Within the top-level wrapper, the GAP stage is invoked after the second convolution, quantization, and residual addition stages have produced the final feature map of the ResNet residual block. Because the memory layout produced by quantization is already partitioned and organized in channel-major form, GAP can operate directly on it without any intermediate reformatting. This makes global average pooling an efficient final stage of the accelerator, requiring only simple additions and a single division per channel.
Figure 4.7 shows the GAP block as the terminal stage of the pipeline, consuming the output of the add–ReLU block and producing the feature vector fed to subsequent classification layers or exported to software.

4.5 Quantization Kernel Implementation

The quantization stage is implemented as a templated HLS module that applies a fixed-point conversion to every element of the feature map. The software testbench invokes the top-level function `top_wrap`, which receives the pre-quantized activation tensor (`relu_quantized`) and produces an output array of type `ap_uint<BIT_QUANT>` with dimensions `output[BATCH][CHANNELS][HEIGHT][WIDTH]`. After execution, the testbench compares each output element with the corresponding reference value stored in `expected`. Any deviation larger than a predefined `TOLERANCE` is reported, ensuring that the hardware implementation matches the expected behaviour of the quantization model.
The core computation is carried out by the templated function

`matrix_wrapper` < `HEIGHT`, `WIDTH`, `BIT_INPUT`, `BIT_QUANT`, `SHIFT_FACTOR_TOT` > ();

which processes a two-dimensional slice of the tensor. The input is a matrix of type `ap_int<BIT_INPUT>` with size `HEIGHT` × `WIDTH`, and the output is a matrix of identical shape stored in `ap_int<BIT_QUANT>`. The template parameters determine both the geometry of the computation and the format of the fixed-point conversion: `BIT_INPUT` sets the bit-width of the incoming data, `BIT_QUANT` defines the target precision, and `SHIFT_FACTOR_TOT` represents the scaling factor used during dequantization and requantization. Since this factor is resolved at compile time, the hardware applies the scaling using inexpensive bit-shift operations rather than multipliers.
The function consists of two nested loops over height and width, each annotated with `#pragma HLS PIPELINE II=1` to ensure that the accelerator accepts one new value per clock cycle once the pipeline is filled. For each element (h, w), the wrapper calls:

`deq_and_quant` < `BIT_INPUT`, `BIT_QUANT`, `SHIFT_FACTOR_TOT` > (`input[h][w]`),

which applies the actual numerical conversion. This helper function first dequantizes the incoming fixed-point value using the inverse scale, then requantizes it using the target scale, enforcing saturation or truncation as needed. Keeping the numerical logic inside `deq_and_quant` allows `matrix_wrapper` to remain a pure dataflow operator.
During software simulation (i.e., when `__SYNTHESIS__` is not defined), the wrapper also generates a debug log in `matrix_check.txt`. For each processed element, it records the input value, its coordinates, and the quantized result. This mechanism helps during verification but is automatically removed during synthesis, ensuring that no additional I/O hardware is generated in the FPGA implementation.

4.6 Skip-Add Layer in ResNet-18

One of the key ideas behind ResNet-18 is the use of the *skip-add* layer, which plays a crucial role in how information flows through the network. Instead of forcing each block to transform its input completely from scratch, ResNet-18 allows the original input to "skip" the sequence of convolutions and rejoin the processed signal at the end of the block through a simple element-wise addition. This design may appear minimal, yet it has a profound impact: by preserving a direct path for the gradients during backpropagation, the skip-add connection helps the model avoid the vanishing-gradient problem that typically affects deeper networks. At the same

time, it encourages the layers to learn only the residual part of the transformation, making training more stable and efficient. In practice, this mechanism allows ResNet-18 to reach greater depth and representational power without incurring the optimization difficulties that usually arise in deep architectures.

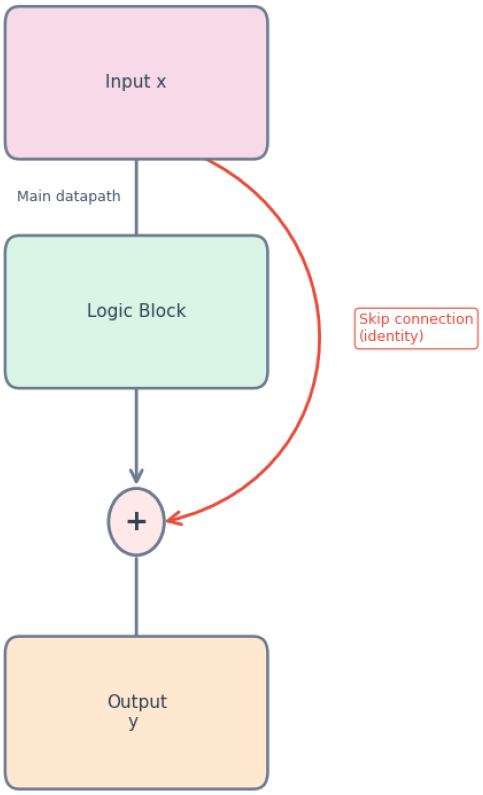

Figure 4.6: Conceptual illustration of the skip-add operation within a residual block of ResNet-18.

4.6.1 Hardware Implementation of the Skip-Add Layer

The skip-add layer in the ResNet-18 basic block is implemented directly in hardware using a streaming and memory-based design. At the top level, the function *top_wrapper* receives the input feature maps over an AXI4-Stream interface (*memory_in_stream*) and immediately duplicates this stream through the helper function `duplicate_input_stream`. One copy of the data (`input_stream_conv`) is fed into the main convolutional pipeline, while the second copy (`input_stream_add`) is preserved for the residual path and consumed later by the skip-add stage. This duplication makes it possible to retain the original input activations on chip without

reloading them from external memory, while still allowing the convolutional pipeline to operate independently within the `#pragma HLS DATAFLOW` region.
On the main path, the input is first reorganised into a suitable layout for convolution by the `input2conv` and `mem_conv2stream` functions, and then processed by two consecutive convolutional layers (`conv`) with bias addition and nonlinearity. The results of these layers are quantised and stored in partitioned on-chip memories (`out_mem_quant_1` and `out_mem_quant_2`) through the `matrix_wrapper` component. After the second convolution, the tensor `out_mem_quant_2` contains the transformed features corresponding to the residual mapping. In order to realise the skip-add operation, the design applies the function `add_layer_to_mem`, which takes as inputs the cloned original stream (`input_stream_add`) and the quantised convolution output (`out_mem_quant_2`), and produces a new memory `out_mem_quant_3`. Inside `add_layer_to_mem`, the feature maps are traversed in a deterministic order that matches the layout used for `memory_out_quant`. For each spatial position and channel, one element from the original input (read from `input_stream_add`) is added to the corresponding element produced by the convolutional pipeline (read from `memory_in`). The sum is computed with an intermediate 9-bit accumulator (`ap_int<9>`) to avoid overflow during addition, and then cast back to 8 bits. Immediately after the addition, a ReLU activation is applied in hardware by clamping negative values to zero. The result (`sum_relu`) is written into the output memory `memory_out` using exactly the same layout as the convolutional output. In this way, the residual connection is implemented as a pure element-wise addition followed by ReLU, mirroring the functional behaviour of the skip-add layer in the software model, but expressed in a fully pipelined, resource-aware hardware style. The combination of stream duplication, on-chip buffering, and carefully scheduled loops allows the skip-add layer to be executed with an initiation interval of one clock cycle (`II=1`), so that a new element of the residual sum is produced on every cycle. From a system-level viewpoint, the residual path therefore introduces almost no additional latency compared to the convolutional pipeline, while preserving the representational benefits of the ResNet architecture. The final tensor `out_mem_quant_3`, containing the result of the skip-add operation, is then passed to the `global_avg_pool` module and subsequently converted back to an output stream, completing the hardware implementation of the ResNet-18 block with its skip-add layer.

4.7 Loop Unrolling and Pipelining as Hardware Design Strategies

In FPGA-based acceleration, performance is determined not only by the algorithmic structure of the design but also by the degree of parallelism exposed to the synthesis

tool. Beyond memory partitioning and data-reuse techniques, two architectural transformations are particularly influential in shaping the micro-architecture generated by Vitis HLS: *loop unrolling* and *pipelining*. These strategies operate directly on the control flow of the program, restructuring loops so that multiple hardware resources can be utilized simultaneously, thereby improving throughput and reducing overall execution time. Because loops are ubiquitous in CNN operators—from convolution to quantization, reduction, and activation—the manner in which they are transformed has a direct impact on the attainable performance of the accelerator.

4.7.1 Loop Unrolling

Loop unrolling is a spatial parallelization technique that replicates loop iterations at compile time. Conceptually, unrolling transforms a loop into several independent computations that can execute concurrently. For example, a loop that normally performs eight iterations sequentially can be fully unrolled into eight parallel operations, removing the loop control logic entirely. This is achieved in Vitis HLS using the directive

```
#pragma HLS UNROLL
```

optionally extended with a user-defined `factor`, which specifies how many iterations are to be executed in parallel. Full unrolling (*complete unroll*) occurs when the entire loop is replicated. Partial unrolling (`factor=k`) replicates only k iterations, thus striking a balance between achievable parallelism and available resources.

From a hardware perspective, unrolling directly multiplies the datapath: if the loop body performs an addition, unrolling by a factor of four generates four independent adders, each operating on different data. This leads to substantial speedups in loops with large iteration counts, provided that the FPGA can supply enough LUTs, FFs, or DSP blocks to accommodate the replicated logic. At the same time, unrolling imposes strict requirements on memory bandwidth. Parallel operations require parallel data access, and without adequate memory partitioning the compiler may serialize accesses, negating the benefits of unrolling. This tight coupling between unrolling and the memory architecture is particularly evident in convolution kernels, where reading *kernel windows* in parallel requires distributing activation data across multiple BRAM banks.

Thus, loop unrolling is most effective when applied selectively to portions of the design where the loop body is simple and where the memory layout supports multiple concurrent reads. In the context of CNN accelerators, this includes inner loops over filter taps, feature-map windows, and small channel groups, but typically excludes outer loops where resource reuse is preferred.

4.7.2 Pipelining

Pipelining is a temporal parallelization technique that restructures a loop so that multiple iterations can overlap in execution. Instead of replicating hardware, pipelining divides the computation into stages; while one iteration progresses to the next stage, another iteration can enter the pipeline. This dramatically increases throughput without requiring additional functional units. In Vitis HLS, pipelining is invoked through

`#pragma HLS PIPELINE II=1`,

where *II* (Initiation Interval) defines how frequently new iterations are launched. Achieving II = 1 means that the pipeline accepts one iteration per clock cycle, which is typically the ideal operating point for streaming architectures.
Unlike unrolling, pipelining preserves the loop structure and performs no hardware replication beyond what is needed to separate pipeline stages. However, it imposes constraints on the presence of data dependencies between consecutive iterations. If an iteration requires data produced by the previous one, the compiler may be forced to increase the II to maintain functional correctness. Similarly, insufficient memory bandwidth may prevent the initiation of new iterations every cycle, causing *II violations*. For these reasons, successful pipelining often relies on minimizing dependencies and ensuring that memory is partitioned so that all required data can be read within a single cycle.
In the context of convolutional neural networks, pipelining is especially beneficial for loops over spatial coordinates, channel blocks, and data-packing operations. These loops typically involve repeated, independent operations, making them suitable for overlapping execution. For example, the construction of convolution packets or the application of quantization over large activation matrices can often be pipelined with II = 1, allowing the accelerator to accept a new pixel or feature value every cycle.

4.7.3 Combined Use in CNN Hardware Acceleration

Loop unrolling and pipelining complement each other and are frequently used together to exploit both spatial and temporal parallelism. A common pattern in HLS designs consists of unrolling inner loops where the loop body is small and computationally dense while pipelining outer loops to maintain high throughput. This hybrid strategy is particularly effective in convolution kernels, where the inner loops over filter parameters ($FW \times FH \times ICH_$PAR) can be unrolled to compute partial sums in parallel, while the outer loops over output spatial positions are pipelined to ensure a steady processing rate. The combination yields a design in

which each pipeline stage carries out several computations concurrently, maximizing utilization of both DSP resources and on-chip memory.
Achieving these optimizations in practice requires aligning all aspects of the design: memory must be partitioned to support the access patterns generated by unrolled loops; dataflow regions must be structured so that pipelined computations are continuously fed with valid data; and accumulation or reduction operations must be organized to avoid hazardous dependencies. When properly balanced, unrolling and pipelining transform the high-level algorithm into a deeply parallel hardware architecture that can sustain the desired initiation interval and effectively exploit the capabilities of the FPGA fabric.

4.8 Top-level Wrapper Architecture

The `top_wrapper` function constitutes the top-level entry point of the HLS design and defines the external interface of the accelerator. It exposes four AXI4-Stream ports: one for the input activations, two for the convolutional weights of the two layers, and one for the output results.
Each port is declared with `#pragma HLS INTERFACE axis`, while the directive `#pragma HLS INTERFACE ap_ctrl_none port=return` configures the accelerator as a free-running streaming module without explicit control handshakes. Internally, all communication is also performed through HLS streams, and the entire function is annotated with `#pragma HLS DATAFLOW` so that the main processing stages can operate concurrently as soon as their input data become available.
From a structural perspective, `top_wrapper` is organized as a sequence of functional blocks, each responsible for a specific transformation. Together, these blocks form a streaming pipeline that implements the core computation of a ResNet-18 residual module. The overall structure can be interpreted as follows:

- **Kernel streaming.** Two instances of a lightweight kernel-loading function extract the convolution weights from the external AXI streams and transfer them into internal streams with a linear memory order. These streams serve as weight sources for the convolution stages and are consumed sequentially by the associated convolution cores.
- **Input duplication.** The input activation stream is replicated into two identical internal streams. One copy feeds the main convolutional path, while the other is preserved for the skip connection used in the residual addition stage. Duplication occurs entirely within the streaming domain and does not require intermediate buffering.
- **Input tiling and buffering.** The activations destined for convolution are written into a partitioned on-chip memory. This stage reorganizes the

streamed HWC-ordered data into windowed tiles that match the spatial extent of the convolution filters and the degree of channel parallelism. The memory layout is determined by template parameters describing the kernel size, input dimensions, tiling strategy and channel grouping.

- **Memory-to-stream repacking.** A dedicated block reads the tiled activations from on-chip memory and constructs *convolution packets*. Each packet contains all samples required for one convolution window, including the full set of filter taps and the appropriate subset of channels. Stride and padding are applied when computing the effective coordinates of each window, and out-of-range accesses are safely mapped to zeros. The resulting stream of fixed-size packets feeds the convolution operator.

- **First convolution and quantization.** The first convolution block consumes the activation packets, the associated weight stream and the bias values, producing 32-bit accumulated outputs stored in a partitioned memory. A quantization stage then converts these accumulators into 8-bit fixed-point values using the procedure described in Section 4.5. The quantized feature maps form the input to the second convolution layer.

- **Second convolution and quantization.** The same sequence is applied to the second convolution: the quantized output of the first convolution is repacked into convolution packets, processed by the second convolution block and then quantized again. The resulting memory contains the 8-bit feature maps produced by the second convolution in the residual block.

- **Residual addition with ReLU.** In parallel with the main path, the cloned input stream is buffered in a layout matching the quantized output of the second convolution. A simple add-and-ReLU stage traverses both memories in lockstep: for each element, it computes the sum between the skip-path input and the second convolution output, applies the ReLU activation and stores the result in a new memory. This memory contains the final feature maps of the residual block.

- **Global average pooling and output streaming.** The last stage performs global average pooling over all spatial positions. For each output channel, all spatial samples are accumulated and divided by their total count. The resulting vector of pooled values is then converted into AXI4-Stream packets, with the last element marked using the `TLAST` signal, and written to the external output stream.

Due to the `DATAFLOW` directive, these blocks do not wait for one another to complete; instead, they overlap in time whenever possible. For example, while the input

tiling block fills the activation memory, the kernel-loading functions may already be streaming weights, and the first convolution block can begin processing as soon as the initial packets are available. This form of pipelined parallelism maximizes throughput and minimizes buffering requirements.
Figure 4.7 illustrates this organization through a block diagram, showing the datapath from the input activations and filter streams, through the two convolution–quantization stages and the residual addition, to the global average pooling and final output.

4.9 TCL Automation for HLS Project Execution

The entire build process of the accelerator is automated through a TCL script, which drives the Vitis HLS toolchain from project creation to design export. This scripting approach ensures reproducibility, simplifies experimentation with different configurations, and eliminates manual steps during synthesis and simulation.
The script begins by defining the solution name (here `solution_0`) and opening a corresponding HLS project. The `set_top` command assigns `top_wrapper` as the top-level function to be synthesized, so that all interfaces, pragmas and dataflow structures are interpreted with respect to the design described in Section 4.8. The relevant design files (`top_wrapper.cpp`, `conv.h`, `parameter.h`, and related headers) are added to the project with `add_files`, while the testbench is provided separately using the `-tb` flag. This instructs the tool to include the testbench only during simulation, excluding it from synthesis and implementation.
The target FPGA device is selected with `set_part`, and a 5 ns clock period is specified via `create_clock`. These constraints guide both the HLS scheduler and the resource allocator during synthesis. Once the project is fully configured, the script launches the two main phases of the HLS flow: high-level simulation (`csim_design`) and C-to-RTL synthesis (`csynth_design`). The optional co-simulation step is commented out but can be re-enabled to verify that the generated RTL behaves identically to the high-level C++ description.
Finally, `export_design` packages the synthesized RTL into an implementation-ready directory structure suitable for integration into a full FPGA design flow. The script then terminates with `exit`. By encapsulating the entire sequence in TCL, the process can be repeated consistently across different runs, making it easier to track the impact of code changes, pragma adjustments or architectural refinements.

```
# Select the solution
set impl_sel "solution_0"

# Open (or create) the HLS project
open_project conv_2
open_solution ${impl_sel}

# Set the top-level function
set_top top_wrapper

# Add design files
add_files top_wrapper.cpp
add_files top_wrapper.h
add_files conv.h
add_files parameter.h

# Add the testbench
add_files -tb tb_gap.cpp

# Target FPGA device
set_part {xczu9eg-ffvb1156-2-e}

# Create the design clock
create_clock -period 5

# Run C simulation
csim_design

# Run C synthesis
csynth_design

# Optional RTL co-simulation
# cosim_design -trace_level all

# Export synthesized RTL
export_design -flow impl

# Exit the script
exit
```

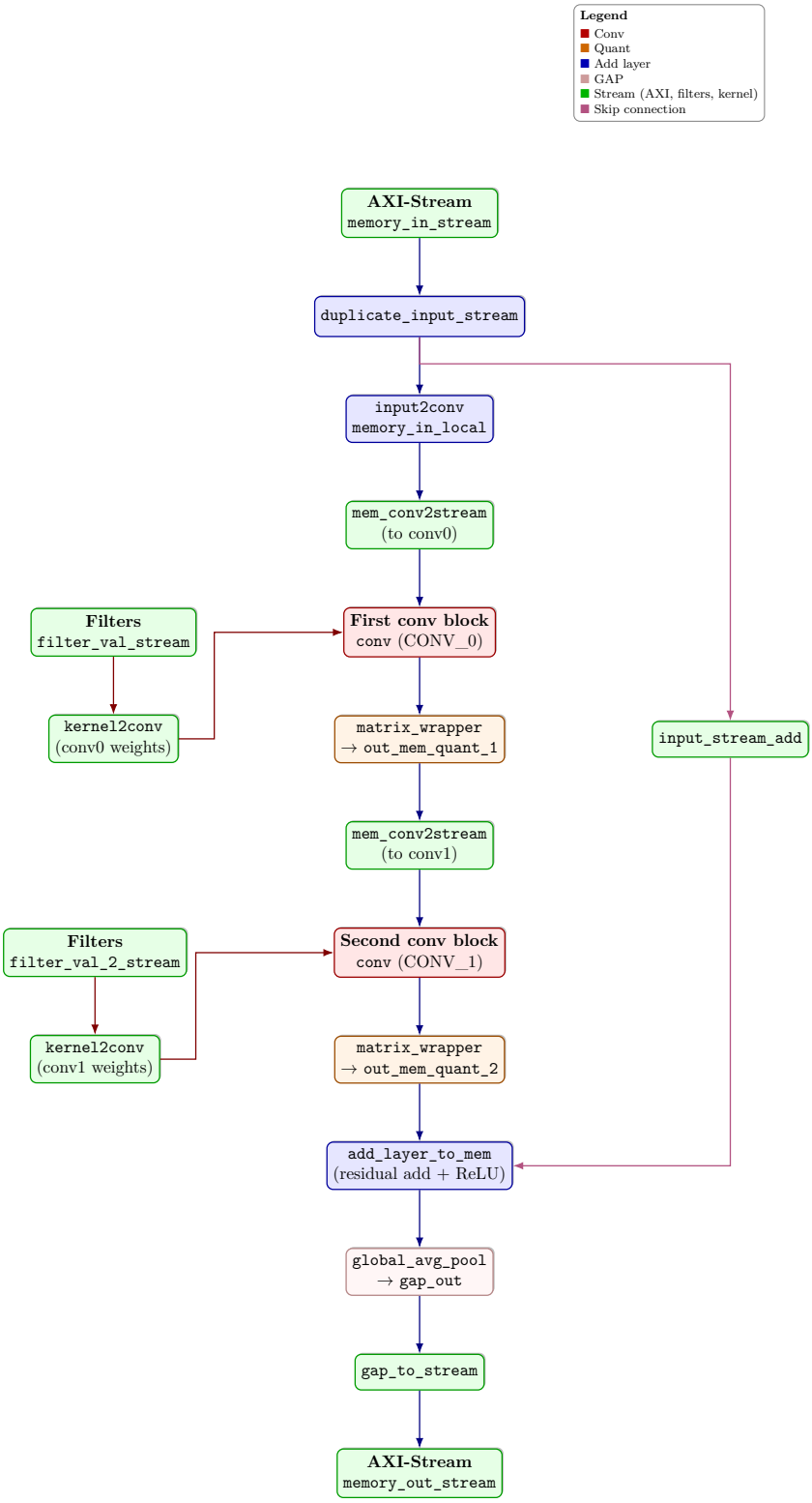

Figure 4.7: Architecture of the `top_wrapper` HLS module.

Chapter 5

Vivado Synthesis Flow and FPGA Implementation

This chapter describes the complete hardware synthesis flow adopted for the implementation of the CNN accelerator on FPGA, focusing on the transition from high-level C/C++ design to a fully deployed hardware engine. The development process relies on a combination of Vitis HLS and Vivado: the former is used to translate the algorithmic description of the accelerator into RTL, while the latter is responsible for transforming that RTL into a device-specific implementation. Starting from the high-level model, the design is first verified through functional simulation and then synthesised into hardware, producing an RTL architecture in which the computational kernels—such as convolution, skip-add, pooling, and quantisation—are expressed as pipelined, cycle-accurate modules. Vivado then imports this RTL and performs logic synthesis, placement, and routing, mapping the accelerator onto the configurable logic blocks, DSP slices, and memory primitives of the selected FPGA.

Once the implementation meets timing constraints and satisfies resource requirements, Vivado generates the bitstream that configures the programmable logic. The accelerator is then integrated within a complete system using the Vivado Block Design environment, which provides a graphical representation of the hardware platform and enables the connection of the custom accelerator to standard IP cores. Through this mechanism, the processing system and the programmable logic are linked via AXI interfaces, forming a unified SoC architecture capable of executing control software on the embedded processors while the accelerator operates in hardware. The resulting bitstream is deployed on the target FPGA platform, in this case an AMD ZCU102 board, which combines ARM CPUs and a large UltraScale+ programmable logic fabric. Finally, the chapter introduces the runtime interface used to program the device and exchange data with the

accelerator, based on a Python workflow built on the PYNQ framework. This interface enables efficient validation of the hardware design, supporting real-time execution of the CNN layers and direct comparison with software reference outputs.

5.1 Vivado Design Suite

Vivado Design Suite is AMD's integrated environment for digital hardware development on FPGA and SoC platforms, providing a unified flow that spans from RTL elaboration to full device configuration. As described in the official user documentation, the toolchain replaces traditional netlist-based flows with a modern, constraint-driven and data-centric architecture that supports large-scale designs and enables advanced optimisation techniques. Vivado accepts HDL sources such as VHDL or Verilog, as well as pre-built or user-defined IP blocks, and transforms them into a device-ready implementation through a sequence of synthesis and physical design steps. The environment also keeps track of design runs and constraints, allowing multiple synthesis or implementation attempts to be compared and tuned efficiently.

At the RTL level, Vivado performs logic synthesis by converting behavioural or structural descriptions into FPGA primitives such as lookup tables, registers, block memories, and DSP slices. During this stage, the tool applies timing- and area-driven optimisations, interprets user-defined XDC constraint files, and prepares a technology-mapped netlist. XDC files allow designers to express both physical requirements—such as I/O pin assignments, placement directives, or floorplanning constraints—and timing objectives, including clock definitions, target frequencies, and path exceptions. In this way, the synthesis process is strongly guided by the intended operating conditions of the final hardware system.

The next stage of the flow is implementation, where the netlist is placed and routed onto the physical resources of the FPGA. Vivado's implementation engine evaluates multiple design metrics simultaneously, seeking a configuration that satisfies timing closure while optimising routing congestion, total wire length, resource utilisation, and power consumption. To achieve this, the tool may apply structural transformations such as register balancing, duplication of timing-critical elements, or insertion of buffer networks. These physical optimisations often modify the structure generated by high-level synthesis, which is why resource estimates from Vitis HLS should be viewed as preliminary. In practice, utilisation figures can either decrease as redundant logic is merged, or increase if replication is necessary to maintain timing at the target clock frequency.

In this project, the HLS-generated accelerator is packaged as an IP block and imported into Vivado for system integration. Through the Block Design interface, the custom IP is combined with standard components such as the Zynq processing

system, AXI interconnects, DMA engines, memory controllers, and peripheral blocks. This graphical composition environment enables the construction of a complete SoC architecture in which software running on the embedded processors orchestrates data transfers to the programmable logic, where the CNN accelerator executes. Once the block design is validated, Vivado synthesises and implements the entire system, ensuring consistency between the accelerator, the interconnect fabric, and the other IP cores.

After timing closure is reached and the physical layout is finalised, Vivado generates the bitstream that programs the FPGA. The bitstream encodes the configuration of LUTs, routing resources, DSP and BRAM elements, clocking networks, and I/O pads. The resulting `.bit` file is then loaded onto the target hardware platform, enabling execution of the accelerator and allowing the system to be tested and debugged through tools such as the Vivado Hardware Manager and the Integrated Logic Analyzer. This completes the end-to-end flow, from high-level algorithmic description to a validated hardware implementation of the CNN accelerator.

Figure 5.1 illustrates the full development pipeline adopted in this project, highlighting the stages handled within Vivado—namely the integration of the generated RTL, the synthesis and implementation phases, and the subsequent programming and debugging performed on the target FPGA.

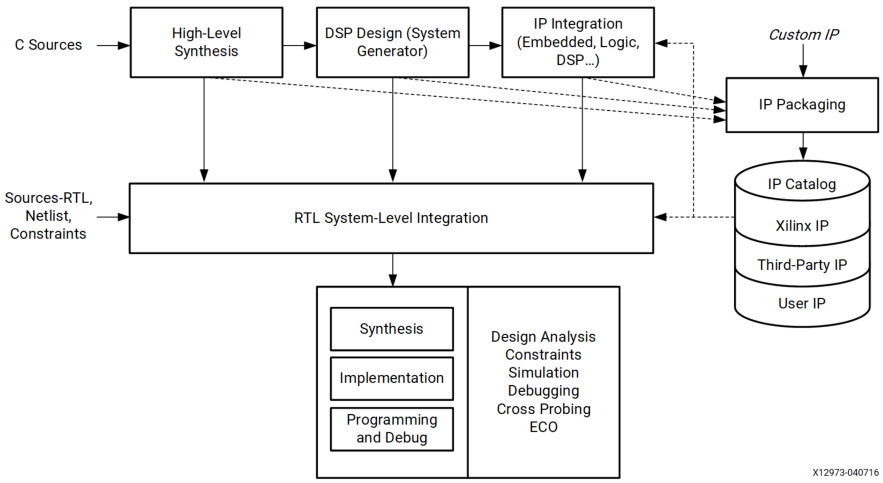

Figure 5.1: Vivado Design Suite High-Level Design Flow [44]

5.2 Block Design

The design flow in Vivado begins with the construction of the Block Design, where the hardware accelerator produced with Vitis HLS is integrated with the remaining system components. Vivado automatically detects AXI-compatible interfaces and

can generate default connections for clocks, resets, and the communication channels among the various modules. Figure 5.2 illustrates the complete Block Design used to implement a single layer of ResNet-18 (including convolution, ReLU activation, and quantization). The main elements of the architecture are summarized below.

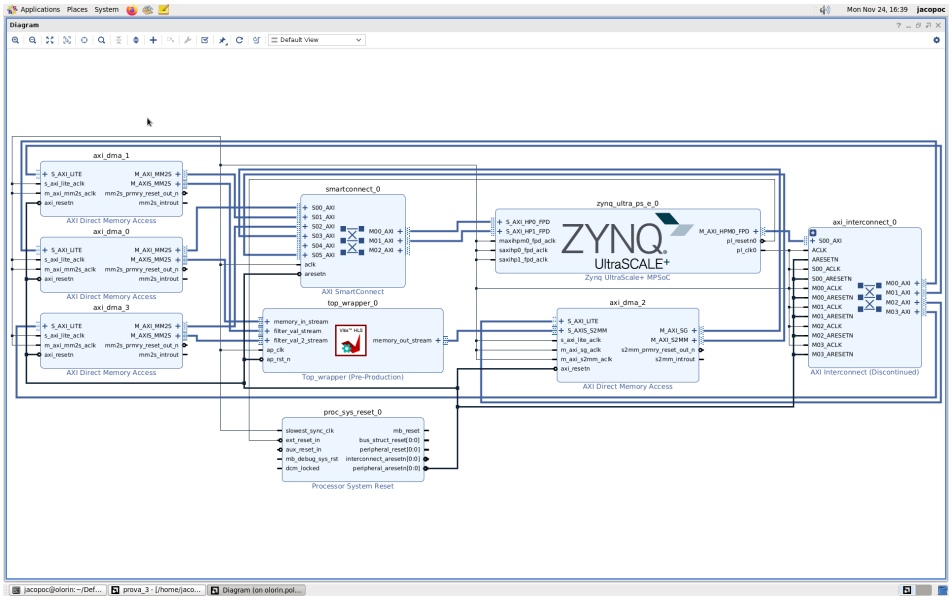

Figure 5.2: Overview of the Block Design for the implemented ResNet-18 layer.

- **Top-level accelerator:** At the core of the design lies the IP exported from Vitis HLS, which encapsulates the C++ description of the layer and its RTL implementation. The module exposes the streaming and control ports discussed in Section 4.8, along with clock and reset inputs.

- **Zynq UltraScale+ MPSoC:** The Processing System (PS) subsystem provides timing and reset to the programmable logic and establishes the communication channels between software and hardware. Configuration commands are issued from the PS to the programmable logic through the *M AXI HPM0 FPD* interface, while the accelerator and the DMAs gain access to the DDR memory through the *S AXI HP* FPD* high-performance ports.

- **Input DMAs (axi dma 0 and axi dma 1):** Two AXI DMA units are responsible for supplying the accelerator with activations and weights. Operating exclusively in the MM2S (memory-to-stream) direction, they retrieve buffers from DDR and forward them to the accelerator through 8-bit AXI4-Stream channels. Their control registers—buffer base address, size, and status—are programmed from the PS using AXI-Lite. The configuration adopted for the

activation DMA is reported in Figure 5.3; the same structure applies to the kernel DMA.

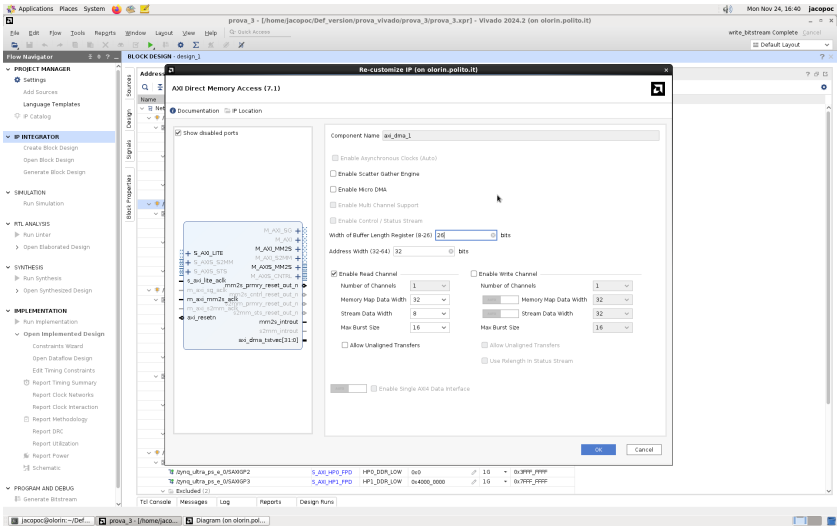

Figure 5.3: Configuration of the DMA responsible for loading activations.

The *Buffer Length Register Width* is set to 26 bits, allowing transfers of up to $2^{26} - 1 \approx 64$ MiB, which is sufficient for entire input tensors even for multi-sample batches. A memory-mapped data width of 32 bits and an 8-bit stream width reflect the quantized nature of the input data. To keep the AXI traffic controlled, bursts are limited to 16 beats. As Scatter-Gather is disabled, each DMA operates in Simple Mode, while continuity of the input stream is maintained at software level using a double-buffering technique.

- **Output DMA (axi dma 2):** A third DMA handles data flowing from the accelerator back to memory. This unit is configured in the opposite direction, S2MM (stream-to-memory), and collects the output stream produced by the accelerator. The memory write parameters are controlled through its AXI-Lite port. Figure 5.4 shows the setup adopted for this DMA.

  In this configuration only the write channel is required, while the Scatter–Gather engine is fully enabled. Instead of relying on a single length register programmed by software, the DMA autonomously processes the chain of buffer descriptors stored in memory, ensuring continuous reception of the output stream without CPU intervention. Given that the output of the GAP of the implemented ResNet-18 layer consists of 512 bytes, each descriptor easily accommodates the entire tensor, and the Scatter–Gather controller automatically triggers the memory writes, manages descriptor completion, and advances to the next entry in the chain.

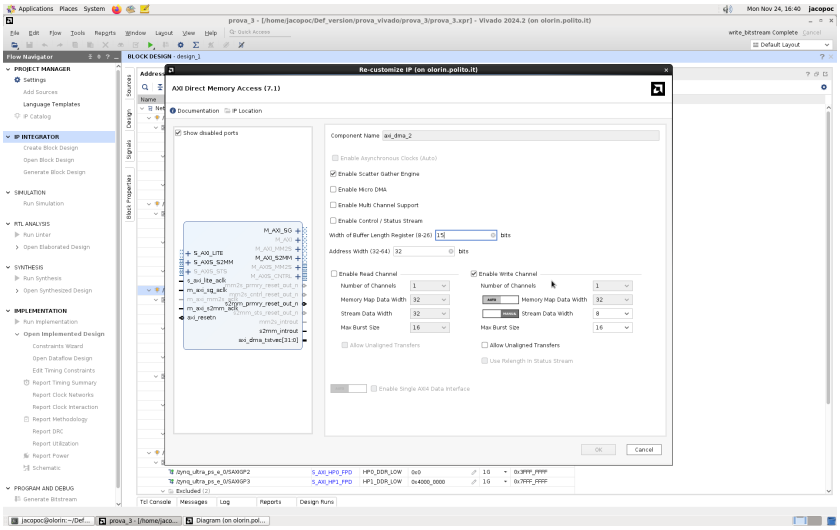

Figure 5.4: Configuration of the DMA used to store the output feature map.

- **AXI SmartConnect:** All memory-mapped connections from the DMAs are routed through an AXI SmartConnect switch, which takes care of arbitration and of inserting any required protocol or data-width conversions.
- **AXI Interconnect (control path):** AXI-Lite transactions issued by the PS to configure the DMAs are dispatched through a dedicated AXI Interconnect, which handles address decoding and routing.
- **Processor System Reset:** Reset synchronization is managed by the Processor System Reset block. It combines the global reset and PL clock to generate stable, clock-aligned reset signals for all components within the programmable logic.

5.3 Managing FPGA Computation with PYNQ

The following section presents the Python script used to execute and evaluate the convolutional accelerator implemented on the Xilinx ZCU102 board. To ensure readability and clarity inside the thesis, the source code is divided into logical units and each fragment is accompanied by a descriptive explanation. This format highlights the operational flow of the script while maintaining a smooth, narrative style.

5.3.1 Initial Imports and Basic Configuration

```
1 import sys, re, time, signal
2 import os
3 import pynq
4 import numpy as np
5 from pynq import Overlay, allocate, PL
```

Listing 5.1: Imports and initial setup

The script begins by importing all modules needed for execution on the ZCU102. Standard Python modules manage file access, timing and regular expression parsing, while NumPy is used for manipulating multidimensional arrays representing inputs and weights of the neural network. The PYNQ framework provides access to the FPGA fabric, DMA channels and contiguous memory allocation, and the call to `PL` prepares the ground for loading the hardware overlay.

5.3.2 Power Sensor Class

```
1 class ZCU102PowerSensor:
2     def __init__(self, name="fpga_power", unit="W", include_ps=
    True, include_mgt=False):
3         ...
4     def _populate_ina_arrays(self):
5         ...
6     def get_value(self, parents_values=None):
7         ...
```

Listing 5.2: Custom ZCU102 power sensor

A significant part of the script consists of the custom class responsible for measuring the instantaneous power consumption of the FPGA. The class scans the Linux virtual filesystem, identifying all INA226 energy monitors mounted on the board. Each sensor is associated with a specific supply rail, such as the core voltage of the programmable logic or the auxiliary voltages supporting peripheral blocks. The class continuously reads the voltage and current values exposed by the operating system, converts them from millivolts and milliamps into standard SI units, and computes the instantaneous electrical power through the product of voltage and current. By integrating this class with the PYNQ `DataRecorder`, the script is able to collect power samples at regular intervals and associate them precisely with each batch of inference executed on the accelerator.

5.3.3 Logging Mechanism

```
class Tee:
    def __init__(self, *streams): self.streams = streams
    def write(self, data):
        for s in self.streams: s.write(data); s.flush()
    def flush(self):
        for s in self.streams: s.flush()
log_fp = open("~/risultati_block_design.txt", "w", buffering=1)
sys.stdout = Tee(sys.__stdout__, log_fp)
sys.stderr = Tee(sys.__stderr__, log_fp)
```

Listing 5.3: Tee logging system

To guarantee complete reproducibility of the experimental results, the script replaces the standard output with a wrapper that duplicates every printed line. As a result, all terminal output is simultaneously available on screen and stored to a log file. This mechanism makes it possible to trace back any unexpected behaviour of the hardware or the software execution.

5.3.4 Utility for Loading Network Parameters

```
def load_array_from_h(path, dtype):
    with open(path, "r") as f: txt = f.read()
    m = re.search(r"\{(.*)\}", txt, re.DOTALL)
    if not m: raise ValueError("File .h non contiene array tra {}")
    nums = re.findall(r"-?\d+", m.group(1))
    return np.array(nums, dtype=dtype)
```

Listing 5.4: Loading arrays from C header files

The network inputs and filter weights used by the accelerator are stored in C-style header files. This function extracts the numerical arrays defined in those files and returns them as NumPy arrays ready to be transferred to the FPGA. By doing so, the script seamlessly bridges the gap between pre-processing performed offline and real-time computation on the board.

5.3.5 Reset of the Programmable Logic and Loading of the Overlay

```
PL.reset()
BITFILE = "/root/jacopoc/overlay/design_1_wrapper.bit"
ol = Overlay(BITFILE)
```

Listing 5.5: Reset and overlay loading

Before any computation can be launched, the programmable logic is reset, ensuring that no residual configuration or DMA transfer remains active from previous runs. The hardware accelerator, developed and synthesized separately, is then loaded onto the FPGA fabric through the PYNQ `Overlay` interface. This bitstream includes the convolutional engines, global average pooling unit, and all the DMA interfaces required to transfer data.

5.3.6 DMA Interface Setup

```
dma_in   = ol.axi_dma_0
dma_w1   = ol.axi_dma_1
dma_w2   = ol.axi_dma_3
dma_out  = ol.axi_dma_2

sc_in  = dma_in.sendchannel
sc_w1  = dma_w1.sendchannel
sc_w2  = dma_w2.sendchannel
rc_out = dma_out.recvchannel
```

Listing 5.6: DMA interfaces

Four DMA engines are instantiated: three of them feed input data or weights to the FPGA, while the remaining one retrieves the processed output vector. This separation allows different data streams to flow toward the accelerator in parallel, without forcing them through a single bottleneck.

5.3.7 Allocation of Buffers and Loading of Network Parameters

```
W, H, C = 7, 7, 512
in_one = load_array_from_h("input_first_conv.h", dtype=np.int8)
in_batch = allocate(shape=(PIXELS_PER_IMAGE*batch_size,), dtype=np
    .int8)
in_batch[:] = np.tile(in_one, batch_size)
w1_vals = load_array_from_h("filter_val_1.h", dtype=np.int8)
w1_batch = allocate(shape=(f1_size*batch_size,), dtype=np.int8)
w1_batch[:] = np.tile(w1_vals, batch_size)
w2_vals = load_array_from_h("filter_val_2.h", dtype=np.int8)
w2_batch = allocate(shape=(f2_size*batch_size,), dtype=np.int8)
w2_batch[:] = np.tile(w2_vals, batch_size)
```

Listing 5.7: Buffer allocation and parameter loading

At this stage the script loads the actual numerical values representing the feature map and the convolutional filters. Since the accelerator expects batches of input

images and weights, the script replicates each single input or filter set as many times as required to fill a batch. All buffers are allocated using `allocate()`, which ensures physical contiguity in memory, a necessary condition for the DMA engines to operate correctly.

5.3.8 Initialization of the Power Recorder

```
sensor = ZCU102PowerSensor(name="fpga_power", unit="W")
recorder = pynq.DataRecorder(sensor)
```

Listing 5.8: Power recorder setup

A single instance of the power sensor is bound to the PYNQ data recorder, which will periodically sample the FPGA power during each batch of inference. This allows energy consumption and average power to be computed precisely for every iteration.

5.3.9 Batch Processing and Accelerator Execution

```
for bi in range(batches):
    recorder.reset()
    recorder.record(0.01)

    rc_out.transfer(out_batch)

    t_in_start  = time.perf_counter_ns()
    sc_in.transfer(in_batch)
    sc_w1.transfer(w1_batch)
    sc_w2.transfer(w2_batch)
    t_in_done = time.perf_counter_ns()

    rc_out.wait()
    t_out_done = time.perf_counter_ns()

    recorder.stop()

    batch_time_s = (t_out_done - t_in_start) / 1e9
    batch_power_W = float(recorder.frame["fpga_power"].mean())
    batch_energy_J = batch_power_W * batch_time_s
```

Listing 5.9: Main processing loop

The core of the script is the loop that processes all batches. For every batch, the recorder begins sampling power, and the output buffer is armed to receive the accelerator's result. The script then triggers three parallel DMA transfers that send the input images and both sets of convolutional weights to the programmable

logic. Once the output DMA signals completion, the recorder stops and the script computes the latency, average power, and total energy for that batch. This mechanism ensures a precise temporal correspondence between the power measurements and the actual accelerator workload.

5.3.10 Output Verification

```
if np.array_equal(out_all, output_ref[:out_all.size]):
    print("Output FPGA uguale al reference!")
else:
    idx = np.where(out_all != output_ref[:out_all.size])[0]
    print(f"Prime differenze idx: {idx[:20]}")
```

Listing 5.10: Numerical comparison with reference

If a reference output has been provided, the script verifies the numerical correctness of the FPGA computation. This step confirms that the hardware accelerator reproduces the expected behaviour of the corresponding software model.

5.3.11 Final Summary and Metrics

```
print("===== POWER & ENERGY =====")
print(f"Mean power: {mean_power_W:.3f} W")
print(f"Total energy: {total_energy:.3f} J")

print("===== SUMMARY =====")
print(f"FPS: {fps:.2f}")
print(f"End-to-End latency avg: {e2e_m:.3f} ms")
```

Listing 5.11: Final performance and energy metrics

At the end of execution the script computes and prints a global summary including average power, total energy consumption, mean latencies and overall throughput. These results offer a complete perspective on the performance and the energy characteristics of the implemented accelerator and constitute the basis for the analysis presented in this thesis.

5.4 The AMD Xilinx ZCU102 Evaluation Board

The hardware platform used for the implementation and testing of the accelerator is the AMD Xilinx ZCU102 evaluation board, a high–end development platform based on the Zynq UltraScale+ MPSoC architecture. This device integrates a heterogeneous processing system that combines a quad-core ARM Cortex-A53 application processor, a dual-core Cortex-R5 real-time processor, and a Mali-400

GPU, together with a large programmable logic (PL) fabric based on UltraScale+ FPGA technology. The coexistence of general-purpose processors and reconfigurable logic makes the ZCU102 particularly suitable for embedded acceleration of deep learning workloads, as control and pre-processing tasks can run on the ARM cores while computationally intensive kernels are offloaded to custom hardware implemented in the PL.

The board provides a comprehensive set of peripherals and memory resources, including 4 GB of DDR4 memory for the processing system, an additional DDR4 bank dedicated to the programmable logic, high-speed transceivers, DisplayPort and HDMI interfaces, several USB and Ethernet ports, and a rich set of FMC connectors that allow the integration of external modules. Its power subsystem is instrumented with multiple INA226 current and voltage sensors connected to the board's supply rails, enabling accurate, real-time monitoring of the energy consumption of both the PS and the PL. This feature is essential for experimental work involving hardware acceleration, as it enables fine-grained evaluation of the energy cost of custom architectures.

Thanks to its combination of computational heterogeneity, high memory bandwidth, and rich instrumentation, the ZCU102 is widely adopted in research and industry for prototyping complex hardware accelerators for machine learning, signal processing, and high-performance embedded computing. In the context of this thesis, it serves as the execution platform for the custom convolutional accelerator, providing the necessary infrastructure to load the bitstream, manage DMA-based data transfers, and collect accurate performance and power measurements during inference.

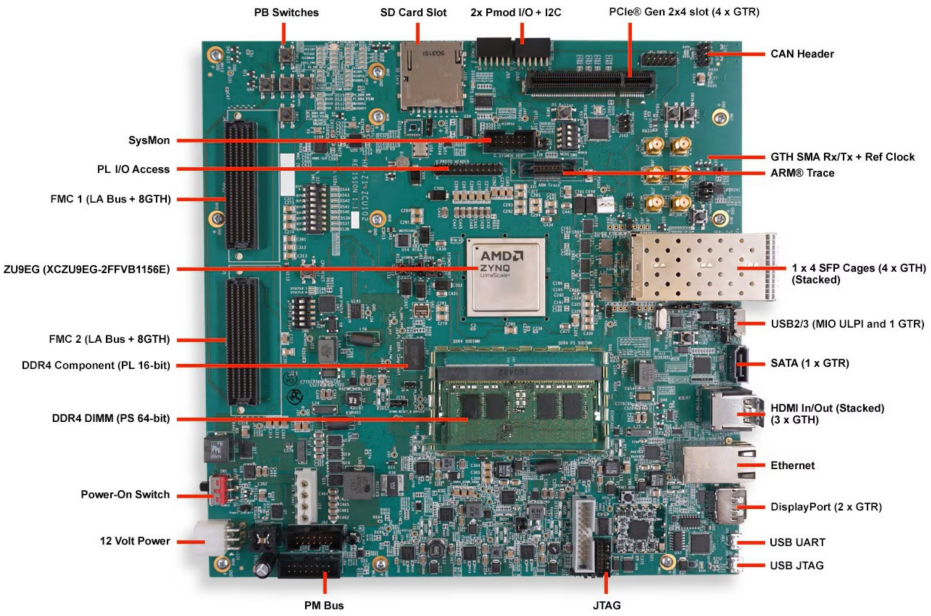

Figure 5.5: The ZCU102 evaluation board, depicted according to its official documentation [45].

Chapter 6

Experimental Results and Analysis

This thesis presented a complete High-Level Synthesis (HLS)–based methodology for rapidly prototyping FPGA accelerators for quantized Convolutional Neural Networks (CNNs). The work demonstrated the feasibility of generating efficient hardware for a ResNet-18 inference pipeline on the AMD Xilinx ZCU102 platform. The design leveraged templated C++ operators, symmetric uniform quantization, and fully streaming dataflow architectures. A key feature of the proposed methodology is the configurability of the input-channel parallelism factor (ICH_PAR), which enables systematic design–space exploration with minimal hardware redesign effort.

Two implementations were produced and evaluated: a minimally parallel design (ICH_PAR = 1) and a moderately parallel configuration (ICH_PAR = 4). Their synthesis results, runtime behaviour, and overall energy efficiency were analysed in depth.

6.1 Summary of Experimental Results

Each configuration was synthesized using Vitis HLS and Vivado, deployed onto the ZCU102 board, and evaluated through an automated inference pipeline implemented with PYNQ. Power consumption was measured via the INA226 sensors at 10 kHz sampling frequency, while latency metrics correspond to the average end-to-end runtime across a batch of 100 images.

Table 6.1 summarizes the synthesis resource utilization and performance results.

Table 6.1: Comparison of resource utilization and performance for different ICH_PAR configurations.

Metric	**ICH_PAR = 1**	**ICH_PAR = 4**
LUT Utilization	135,580 (49.4%)	144,576 (52.7%)
FF Utilization	258,342 (47.1%)	352,807 (64.3%)
BRAM Usage	262 (28.7%)	342 (37.5%)
DSP Usage	19 (0.8%)	76 (3.0%)
Latency per Image [ms]	1201.898	308.179
Throughput [FPS]	0.83	3.24
Average Power [W]	2.904	3.145
Energy per Image [J]	377.949	96.910

6.2 Interpretation of Results

The comparison provides clear evidence of the architectural and energy benefits associated with increasing channel parallelism.

1. Throughput Scaling Increasing input-channel parallelism from 1 to 4 yields a latency reduction of:

$$\frac{1201.898}{308.179} \approx 3.9\times,$$

which closely matches the theoretical speedup expected from replicating four convolution datapaths. Because the entire architecture is implemented as a streaming pipeline, the higher parallelism directly improves the initiation interval of the convolution kernels, thus increasing throughput without introducing unnecessary buffering or control overhead.

2. Resource Utilization and Bottlenecks The parallel design requires additional LUTs, FFs, and BRAM blocks to sustain the increased activation bandwidth. DSP usage grows proportionally with the number of replicated MAC units, yet remains extremely low relative to available hardware resources (3% of DSPs). This confirms that the design is primarily limited by data movement and on-chip memory requirements, rather than by arithmetic capacity.

3. Energy Efficiency The corrected energy measurements reveal a substantial efficiency benefit associated with higher parallelism. Even though average power increases slightly (+8.3%), the much shorter execution time leads to a dramatic

reduction in energy per inference:

$$E = P \cdot t.$$

The energy consumption drops from

$$377.949 \text{ J} \quad \text{to} \quad 96.910 \text{ J},$$

equivalent to a

$$\approx 3.9 \times \text{ improvement.}$$

This scaling matches the reduction in latency and confirms that, for deeply pipelined FPGA accelerators, *increasing parallelism improves both performance and energy efficiency*. The reason is structural: the static power component and the baseline system overhead dominate the energy cost when execution time is long. Reducing total runtime therefore has a compounded positive effect.

4. Suitability of FPGAs for Quantized CNN Inference Both configurations operate entirely in fixed-point arithmetic and meet timing constraints at 200 MHz. The extremely low DSP utilization demonstrates the advantage of quantization-aware design: multipliers are small, accumulators are efficient, and the dominant resource is memory bandwidth. This aligns well with the strengths of modern FPGAs and validates their role as energy-efficient platforms for edge inference.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis presented a complete and experimentally validated methodology for the rapid prototyping of FPGA-based accelerators targeting quantized Convolutional Neural Networks (CNNs). Through the combination of High-Level Synthesis (HLS), template-based hardware operators, and a fully streaming dataflow architecture, the work demonstrated that a software-defined CNN can be translated into a performant and energy-efficient hardware implementation with limited engineering effort and high predictability.
A key objective of the study was to analyze how the degree of architectural parallelism influences the performance and efficiency of the accelerator. Two configurations were evaluated, corresponding to `ICH_PAR = 1` and `ICH_PAR = 4`, which respectively represent a minimally parallel design and a moderately parallel variant. The experimental campaign conducted on the AMD Xilinx ZCU102 platform yielded several noteworthy outcomes.

- **Near-linear performance scaling.** Increasing the input-channel parallelism from 1 to 4 resulted in a latency reduction from 1201.898 ms to 308.179 ms, corresponding to a speedup of approximately 3.9×. This behavior confirms that the accelerator effectively exploits pipeline-level concurrency and that HLS-generated hardware can preserve the deterministic throughput characteristics associated with traditional hand-written RTL implementations.
- **Significant energy-efficiency improvement.** Although the higher-parallelism configuration exhibits a slightly higher average power consumption (3.145 W vs. 2.904 W), the drastically reduced execution time leads to a substantial

decrease in energy per inference:

$$E_{\text{ICH\_PAR=1}} = 377.949\,\text{J}, \qquad E_{\text{ICH\_PAR=4}} = 96.910\,\text{J}.$$

This corresponds to a 3.9× energy reduction, underscoring the effectiveness of parallelism not only in improving latency but also in lowering the overall energy footprint—an essential requirement for edge AI deployments.

- **Moderate and predictable resource scaling.** Resource utilization increases with parallelism, but all resource categories (LUTs, FFs, BRAMs, DSPs) remain comfortably within the capacity of the ZCU102 device. DSP usage stays particularly low due to quantization and fixed-point arithmetic, confirming that modern FPGAs can accommodate far more extensive parallelism or larger network architectures without compromising numerical precision.
- **Validation of the HLS-centered design flow.** The proposed methodology dramatically reduces design time while still enabling the generation of optimized hardware. Architectural parameters such as `ICH_PAR` can be modified directly at the C++ level, eliminating the need for rewriting low-level RTL. This level of flexibility enables rapid and systematic exploration of the design space, which would be cumbersome and time-consuming using traditional hardware design methodologies.

Overall, the results confirm the suitability of FPGAs—when paired with quantization and HLS-driven design—for edge-oriented CNN inference, where latency, power, and flexibility must be simultaneously optimized. The methodology developed in this thesis constitutes a practical and effective approach for building fast, energy-efficient, and scalable neural network accelerators.

7.2 Future Work

Although the work achieved significant results, several research directions remain open and provide opportunities to further expand the capabilities and scalability of the proposed accelerator architecture.

- **Ultra-low-precision arithmetic.** Extending the operator library to support INT4, INT2, or even binary-weight arithmetic could drastically reduce power consumption and further increase throughput. Coupling these formats with quantization-aware training would ensure accuracy remains acceptable while enabling extremely compact hardware designs.
- **Support for depthwise-separable and modern convolutions.** Many efficient architectures—such as MobileNet and EfficientNet—rely heavily on

depthwise-separable convolutions. Implementing optimized HLS kernels for these operators would broaden the applicability of the accelerator to lightweight models commonly used in embedded inference.

- **Automatic tiling and cross-layer fusion.** For large CNNs, feature maps and parameters may exceed on-chip memory capacity. Introducing automated tiling strategies, along with cross-layer fusion (e.g., Conv+ReLU+Quantization), would reduce data movement, minimize memory footprint, and significantly improve efficiency.
- **Analytical modeling for predictive design-space exploration.** Developing analytical models for latency, resource utilization, and energy would allow designers to estimate the best architectural parameters prior to synthesis. This would greatly accelerate the design cycle and help identify optimal configurations early in the process.
- **Full automation within the AIDGE toolchain.** A fully automated flow—from a trained PyTorch model to FPGA bitstream generation—would further materialize the NEUROKIT2E vision of an industry-grade, European, sovereign embedded-AI ecosystem. Automating graph transformations, quantization, HLS code emission, and FPGA synthesis represents a significant but achievable goal.
- **Scaling to deeper or full-network pipelines.** Extending the streaming approach to multi-layer blocks or even entire networks would demonstrate the generality and robustness of the methodology. Such a system-level implementation would also reveal new optimization opportunities related to activation scheduling, inter-layer buffering, and pipeline balancing.

In summary, this thesis established a solid foundation for HLS-based FPGA acceleration of quantized CNNs. The demonstrated performance, energy efficiency, and design flexibility position this approach as a strong candidate for next-generation edge-AI systems. The future extensions outlined above represent natural and promising directions toward increasingly powerful, compact, and automated FPGA accelerators.

Bibliography

[1] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. «A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects». In: *IEEE Transactions on Neural Networks and Learning Systems* 33.12 (2022), pp. 6999–7019. DOI: 10.1109/TNNLS.2021.3084827 (cit. on pp. 4, 16, 19).

[2] Daniel Hugo Cámpora Pérez. *A Practical Approach to Convolutional Neural Networks*. Slides presented at the CERN School of Computing, Universidad de Sevilla. 2019. URL: https://indico.cern.ch/event/803258/contributions/3359603/ (cit. on pp. 4, 16, 19).

[3] Ronald N. Bracewell. *The Fourier Transform and Its Applications*. McGraw-Hill, 2000 (cit. on pp. 4, 5, 8).

[4] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, 1999 (cit. on pp. 4, 7).

[5] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice Hall, 2002 (cit. on p. 6).

[6] Stéphane Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, 1999 (cit. on pp. 6, 8).

[7] Maria Vakalopoulou, Stergios Christodoulidis, Ninon Burgos, Olivier Colliot, and Vincent Lepetit. «Deep Learning: Basics and Convolutional Neural Networks (CNNs)». In: *Machine Learning for Brain Disorders*. Ed. by Olivier Colliot. Vol. 197. Neuromethods. Springer, 2023, pp. 77–110. ISBN: 978-1-0716-3195-9. DOI: 10.1007/978-1-0716-3195-9_3 (cit. on p. 7).

[8] Linus Pettersson. «Convolutional Neural Networks on FPGA and GPU on the Edge: A Comparison». In: (June 2020). UPTEC F 20028, ISSN 1401-5757 (cit. on p. 7).

[9] Harry L. Van Trees. *Detection, Estimation, and Modulation Theory*. John Wiley & Sons, 2001 (cit. on p. 7).

[10] James W. Cooley and John W. Tukey. «An Algorithm for the Machine Calculation of Complex Fourier Series». In: *Mathematics of Computation* 19 (1965), pp. 297–301 (cit. on p. 9).

[11] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. «A Survey of Quantization Methods for Efficient Neural Network Inference». In: *arXiv preprint arXiv:2103.13630* (2021). Comprehensive taxonomy of quantization techniques including PTQ, QAT, mixed-precision, and theoretical error analysis. URL: https://arxiv.org/abs/2103.13630 (cit. on pp. 9, 10, 12, 13).

[12] Markus Nagel, Raoul Amjad, and Tijmen Blankevoort. *A White Paper on Neural Network Quantization*. Tech. rep. Comprehensive guidelines for PTQ and QAT across CNNs and transformers. Qualcomm AI Research, 2021. URL: https://arxiv.org/abs/2106.08295 (cit. on pp. 9–13).

[13] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. «Fixed-point Quantization of Deep Convolutional Networks for Inference on Resource-Constrained Hardware». In: *arXiv preprint arXiv:2102.02147* (2021). Describes practical mixed-precision quantization and hardware deployment for CNN inference. URL: https://arxiv.org/abs/2102.02147 (cit. on pp. 9, 11–13).

[14] Qiang Liu, Han Wang, Zhipeng Huang, and Wei Zhang. «Low-Bit Quantization of Deep Neural Networks: A Comprehensive Review». In: *arXiv preprint arXiv:2505.05530* (2025). Extensive review of sub-8-bit quantization, ternary, and binary methods for modern CNNs and transformers. URL: https://arxiv.org/abs/2505.05530 (cit. on pp. 9, 13).

[15] Alice Cortney, Rahul Bhattacharya, and Anuj Singh. «Quantization Techniques for Efficient Deep Learning: A Review of Trends and Challenges». In: *TechRxiv* (2024). Analyzes trade-offs between quantization precision, accuracy, and energy efficiency across architectures. URL: https://arxiv.org/abs/2205.07877 (cit. on p. 9).

[16] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. «Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights». In: *International Conference on Learning Representations (ICLR)*. Proposes iterative weight quantization to powers-of-two with retraining. 2017. URL: https://arxiv.org/abs/1702.03044 (cit. on pp. 11, 12, 14).

[17] Sergey Ivanov, Artem Khvalkovskiy, Kirill Zhuravlev, and Egor Shalymov. «Adaptive Quantization for Neuromorphic Computing Using Phase-Change Memory». In: *arXiv preprint arXiv:2310.00337* (2023). Discusses quantization robustness and drift compensation in analog/PCM-based inference hardware. URL: https://arxiv.org/abs/2310.00337 (cit. on p. 13).

[18] Vadim Kryzhanovskiy, Mikhail Timofeev, Ilya Makarov, Roman Menshikov, and Sergey I. Nikolenko. «QPP: Real-Time Quantization Parameter Prediction for Deep Neural Networks». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2021, pp. 14801–14811. DOI: 10.1109/CVPR46437.2021.01459. URL: https://openaccess.thecvf.com/content/CVPR2021/html/Kryzhanovskiy_QPP_Real-Time_Quantization_Parameter_Prediction_for_Deep_Neural_Networks_CVPR_2021_paper.html (cit. on p. 14).

[19] Afia Zafar, Noushin Saba, Ali Arshad, Amerah Alabrah, Saman Riaz, Mohsin Suleman, Shahneer Zafar, and Muhammad Nadeem. «Convolutional Neural Networks: A Comprehensive Evaluation and Benchmarking of Pooling Layer Variants». In: *Symmetry* 16.11 (2024), p. 1516. DOI: 10.3390/sym16111516 (cit. on pp. 14, 16, 18, 19).

[20] Lei Zhao and Zhonglin Zhang. «An improved pooling method for convolutional neural networks». In: *Scientific Reports* 14 (2024), p. 1589. DOI: 10.1038/s41598-024-51258-6 (cit. on pp. 14, 16).

[21] Ljubisa Stankovic and Danilo Mandic. «Convolutional Neural Networks Demystified: A Matched Filtering Perspective Based Tutorial». In: *arXiv preprint arXiv:2108.11663* (2022) (cit. on pp. 17, 18).

[22] Yansel Gonzalez Tejeda and Helmut A. Mayer. «Deep Learning with Convolutional Neural Networks: A Compact Holistic Tutorial with Focus on Supervised Regression». In: *Machine Learning and Knowledge Extraction* 6.4 (2024), pp. 2753–2782. DOI: 10.3390/make6040132 (cit. on pp. 17, 18).

[23] Ilaria Cacciari and Anedio Ranfagni. «Hands-On Fundamentals of 1D Convolutional Neural Networks—A Tutorial for Beginner Users». In: *Applied Sciences* 14.18 (2024), p. 8500. DOI: 10.3390/app14188500 (cit. on p. 18).

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90 (cit. on pp. 19, 20).

[25] Alireza Zaeemzadeh, Nazanin Rahnavard, and Mubarak Shah. «Norm-Preservation: Why Residual Networks Can Become Extremely Deep?» In: *arXiv preprint arXiv:1805.07477* (2018). URL: https://arxiv.org/abs/1805.07477 (cit. on p. 21).

[26] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. «Squeeze-and-Excitation Networks». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2019). Originally published as arXiv preprint arXiv:1709.01507. URL: https://arxiv.org/abs/1709.01507 (cit. on p. 21).

[27] Khalid Abdelouahab, Maxime Pelcat, Jérémie Sérot, and François Berry. «Accelerating CNN inference on FPGAs: A Survey». In: *arXiv preprint arXiv:1806.01683* (2018) (cit. on p. 21).

[28] Sparsh Mittal. «A survey of FPGA-based accelerators for convolutional neural networks». In: *Neural Computing and Applications* (2018) (cit. on p. 21).

[29] Daniel Gschwend. *ZynqNet: An FPGA-accelerated embedded convolutional neural network*. 2016 (cit. on p. 22).

[30] Himanshu Singh. *The convolution Engine*. Accessed: 2025-11-15. The Data Bus. 2025. URL: `https://thedatabus.in/convolver/` (cit. on p. 22).

[31] Aman Dua, Yufei Li, and Fengbo Ren. «Systolic-CNN: An OpenCL-defined scalable run-time-flexible FPGA accelerator architecture for CNN inference». In: 2020 (cit. on pp. 22, 23).

[32] Yufei Li, Zhen Liu, Kai Xu, Haoyang Yu, and Fengbo Ren. «A GPU-outperforming FPGA accelerator architecture for binary convolutional neural networks». In: 2017 (cit. on p. 23).

[33] Zhen Wang. «Briefly Analysis about CNN Accelerator based on FPGA». In: *Procedia Computer Science* (2022) (cit. on p. 23).

[34] M. S. Azzaz, A. Maali, M. Saad, R. Kaibou, H. Hamil, and I. Kakouche. «FPGA HW/SW Codesign Approach for Real-time Image Processing Using HLS». In: *2020 International Conference on Communications, Signals and Systems (CCSSP)*. 2020, pp. 1–6. DOI: `10.1109/CCSSP49278.2020.9151686` (cit. on p. 24).

[35] Zhengdong Li, Frederick Ziyang Hong, and C. Patrick Yue. «FPGA-based Acceleration of Neural Network for Image Classification Using Vitis AI». In: *arXiv preprint arXiv:2412.20974* (2024) (cit. on pp. 24, 27).

[36] Marion Sudvarg, Chenfeng Zhao, Ye Htet, Meagan Konst, Thomas Lang, Nick Song, Roger D. Chamberlain, Jeremy Buhler, and James H. Buckley. «HLS Taking Flight: Toward Using High-Level Synthesis Techniques in a Space-Borne Instrument». In: *ACM International Conference on Computing Frontiers*. ACM, 2024, pp. 1–11 (cit. on p. 25).

[37] Thea Aarrestad et al. «Fast Convolutional Neural Networks on FPGAs with HLS4ML». In: *arXiv preprint arXiv:2101.05108* (2021) (cit. on p. 25).

[38] Mannhee Cho and Youngmin Kim. «FPGA-Based Convolutional Neural Network Accelerator with Resource-Optimized Approximate Multiply-Accumulate Unit». In: *Electronics* 10.22 (2021), p. 2859 (cit. on p. 26).

[39] Anastassis Kapetanakis, Aggelos Ferikoglou, George Anagnostopoulos, and Sotirios Xydis. «Dataflow Optimized Reconfigurable Acceleration for FEM-Based CFD Simulations». In: *Design, Automation & Test in Europe Conference (DATE)*. 2025 (cit. on p. 27).

[40] CEA-List. *Aidge: Independent Deep Learning Framework for Embedded AI*. `https://list.cea.fr/en/aidge/`. Accessed: 2025-11-16. 2025 (cit. on p. 31).

[41] Contributing Writer. *What Is ResNet-18? How to Use the Lightweight CNN Model*. Roboflow Blog. Accessed: 2025-11-16. June 2025. URL: `https://blog.roboflow.com/resnet-18/` (cit. on p. 34).

[42] Rodrigo Rico Gómez, Joe Lorentz, Thomas Hartmann, Arda Goknil, Inder Pal Singh, Tayfun Gökmen Halaç, and Gülnaz Boruzanlı Ekinci. «An AI pipeline for garment price projection using computer vision». In: *Neural Computing and Applications* 36 (2024), pp. 15631–15651. DOI: `10.1007/s00521-024-09901-w` (cit. on p. 34).

[43] Aravind Vasudevan, Andrew Anderson, and David Gregg. «Parallel Multi Channel Convolution using General Matrix Multiplication». In: *arXiv preprint arXiv:1704.04428* (2017). URL: `https://arxiv.org/abs/1704.04428` (cit. on p. 35).

[44] AMD Inc. *Vivado Design Suite User Guide: Implementation — SDC and XDC Constraint Support (UG904)*. 2025.1. Document ID: UG904. AMD Inc. San Jose, CA, USA, May 2025. URL: `https://docs.amd.com/r/en-US/ug904-vivado-implementation/SDC-and-XDC-Constraint-Support` (cit. on p. 56).

[45] AMD Xilinx. *ZCU102 Evaluation Board User Guide*. Available online. Advanced Micro Devices, Inc. 2023. URL: `https://docs.xilinx.com/r/en-US/ug1182-zcu102-eval-bd` (cit. on p. 66).