Politecnico di Torino

POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Prototyping and Evaluation of Code Generation for CNN Acceleration on FPGAs For the AIdge ML Deployment Framework

Supervisors
Fabian CHERSI (CEA Saclay)
Prof. Luciano LAVAGNO
Prof. Mihai Teodor LAZARESCU
Co-supervisor Roberto BOSIO
Co-supervisor Teodoro URSO

Candidate
Lorenzo RIZZO

October 2025

Abstract

The ongoing technological revolution is reshaping the way we live, work and communicate, with Artificial Intelligence (AI) emerging as one of the most disruptive and influential forces behind this evolution. Within this domain, Machine Learning (ML) enables systems to learn from data and improve performance without explicit programming. Among the most influential architectures in the field of ML, Convolutional Neural Networks (CNNs) have established themselves as the standard for processing spatially structured data such as images and videos.

The growing complexity of AI models and the demand for real-time processing highlight the limitations of relying solely on centralized cloud infrastructures. Edge computing, in this context, allows data to be processed closer to its source, reducing latency, bandwidth usage, and energy consumption. Field Programmable Gate Arrays (FPGAs), with their reconfigurable architectures and highly parallel computations, are particularly suited for accelerating AI workloads at the edge. Among the most innovative and widely adopted approaches is High-Level Synthesis (HLS). HLS further simplifies the design of FPGA-based accelerators, enabling rapid prototyping of application-specific hardware by allowing designers to describe functionality in high-level languages like C or C++, instead of traditional Hardware Description Languages (HDLs) such as VHDL or Verilog.

The goal of this thesis, conducted in collaboration with French Alternative Energies and Atomic Energy Commission (CEA) Saclay, is to prototype and evaluate the code-generation capabilities of the AIdge ML deployment framework for FPGAs. AIdge is an open-source deep-learning platform specialized in the design of deep neural networks. The work comprises selecting an appropriate CNN model, implementing a small set of layers in C++, and assessing the generated code with Vitis HLS. A detailed investigation addresses the partitioning of the FPGA's on-chip memory that stores activations in Height Width Channels (HWC) format. Partitioning the memory in this manner enables the convolution units to be supplied with the appropriate unrolling factor, allowing multiple computations to execute in parallel and thereby increasing throughput. The objective is to evaluate the viability and efficiency of automatic code generation for FPGA-based CNN acceleration. In particular, template functions were developed that currently support two CNN architectures, LeNet and ResNet-18, and are suitable for integration into the AIdge framework.

L'ansia ti frega, chi vuoi che creda
Se anche tu non credi in te?
Però ricorda che chi sopporta
Tutto il sudore e lo stress
Arriva al top fra', fatica un tot
Fino alla cima, G.O.A.T.
MARRACASH — G.O.A.T. (Il Cuore)

Ai prof. Lavagno e Lazarescu, ai dottorandi Roberto e Teodoro,
per la guida e il sostegno prestati in questo lavoro.

Alla mia famiglia,
che ha permesso tutto ciò e sostenuto in ogni passo.

Ai miei nonni Anna e Rocco,
i miei angeli custodi, che mi avete guidato da lassù.

Ai miei nonni Angela e Renato,
i miei due più grandi sostenitori.

A miei zii e amici,
grazie per essere stati sempre presenti.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence

ML Machine Learning

CNN Convolutional Neural Network

HLS High-Level Synthesis

HDL Hardware Description Language

RTL Register-Transfer Level

FPGA Field Programmable Gate Array

CEA French Alternative Energies and Atomic Energy Commission

GPU Graphics Processing Unit

ASIC Application Specific Integrated Circuit

ANN Artificial Neural Network

ReLU Rectified Linear Unit

FSM Finite State Machine

IC Integrated Circuit

CPU Central Processing Unit

PROM Programmable Read-Only Memorie

PAL Programmable Array Logic

PLA Programmable Logic Arrays

SRAM Static Random-Access Memory

OTP One-time Programmable

BRAM Block RAM

LUT-RAM Look-Up Table RAM

CLB Configurable Logic Block

LUT Look-Up Table

DSP Digital Signal Processing

IOB Input/Output Block

MAC Multiply-accumulate

SoC System-on-Chip

PL Programmable Logic

PS Processing System

DDR Double Data Rate

N2D2 Neural Network Design & Deployment

QAT Quantization Aware Training

PTQ Post-Training Quantization

HWC Height Width Channels

DMA Direct Memory Access

ResNet Residual Network

IP Intellectual Property

XDC Xilinx Design Constraints

GUI Graphical User Interface

Tcl Tool Command Language

1 Introduction

The goal of the thesis, in collaboration with CEA Saclay, is to prototype and test the code generation capabilities of the AIdge ML deployment framework for FPGAs. Specifically, the aim is to implement template functions designed to accommodate various types of neural network models. Moreover, the assignment required the selection of a suitable CNN architecture, along with the definition and implementation, using High-Level Synthesis (HLS), of a set of layers to be supported. After implementing the design in Vitis HLS at a high level (C++), synthesis was performed, RTL was generated, and the Intellectual Property (IP) core was exported; the flow then proceeded in Vivado. Vivado is used to construct the Block Design, where the project IP is connected to existing IP cores, such as AXI Direct Memory Access, for supplying and capturing input/output data. Vivado also performs logic synthesis and implementation, and generates the bitstream, the FPGA configuration file. Subsequently, the design is programmed onto the FPGA and, after verifying correct operation, data are collected on resource utilization, power consumption, and performance.

The adopted strategy is based on Filter reuse, as opposed to Input reuse. This approach is specifically designed for scenarios involving a large number of filters, where all activations can be stored in memory. To enhance throughput, a memory partitioning method has been implemented, allowing for more efficient data access and processing. Specifically, a two-dimensional partitioned memory is created to supply the convolution with data at the appropriate unrolling factor, enabling computations across one or more channels to proceed concurrently.

This first chapter provides a general overview of the work undertaken, offering essential background on the core concepts and technologies that underpin the project. It defines the scope of the thesis, outlines the research questions addressed, and states the primary objectives pursued throughout the study. The chapter concludes with a description of the thesis structure and summarizing the content and purpose of each subsequent chapter.

1.1 Background

1.1.1 Artificial Intelligence & Machine Learning

Artificial Intelligence (AI) and Machine Learning (ML) are often used interchangeably, but machine learning is a subset of the broader category of AI. Put in context, artificial intelligence refers to the general ability of computers to emulate human thought and perform tasks in real-world environments, while machine learning refers to the technologies and algorithms that enable systems to identify patterns, make decisions, and improve themselves through experience and data. [1]

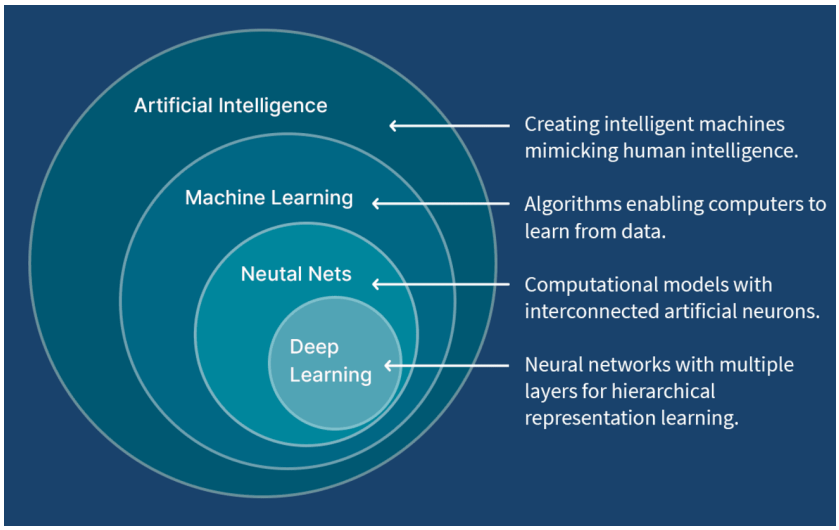


Figure 1: Artificial intelligence vs Machine Learning [2]

Artificial Intelligence

Artificial Intelligence, the ability of a digital computer or computer-controlled robot to perform tasks commonly associated with intelligent beings. The term is frequently applied to the project of developing systems endowed with the intellectual processes characteristic of humans, such as the ability to reason, discover meaning, generalize, or learn from past experience. [3]

Alan Turing, a British mathematician and scientist, is widely regarded as the founding figure of artificial intelligence. In 1950, he published the seminal paper *Computing Machinery and Intelligence* [4], wherein he introduced the concept of the Turing Test, a methodological framework designed to assess a machine's capacity for intelligent behavior comparable to that of a human. In the following decades, the evolution of artificial intelligence led to the development of artificial neural networks, which introduced a computational paradigm inspired by the biological structure of the brain. The rapid progress of machine learning algorithms and, more recently, of deep learning was enabled not only by theoretical advances but also by the exponential increase in available data and computational power. In particular, the use of Graphics Processing Units (GPUs) and later specialized architectures such as FPGAs and Application Specific Integrated Circuits (ASICs) made it possible to train models of unprecedented complexity and scale. This synergy between algorithmic innovation and hardware acceleration has transformed artificial intelligence into a mature engineering discipline, establishing the foundation for the modern field of machine learning.

Machine Learning

Machine learning is a branch of artificial intelligence that enables algorithms to uncover hidden patterns within datasets. It allows them to predict new, similar data without explicit programming for each task. ML finds applications in various fields such as image and speech recognition, natural language processing, recommendation systems, fraud detection, portfolio optimization, and automating tasks. [5] ML algorithms extract patterns and relationships directly from large datasets, adapting their behavior as new information becomes available. This paradigm allows the design of predictive models and decision-

making systems that improve their performance with experience, proving particularly effective in complex scenarios where explicit rule-based programming would be infeasible.

The main categories of ML include supervised learning, where algorithms are trained on labeled data; unsupervised learning, which seeks to discover hidden structures in unlabeled data; and reinforcement learning, where agents learn optimal strategies through interaction with an environment. These methodologies have facilitated significant advancements across a wide spectrum of domains, including medicine, logistics, and marketing. They have enabled a diverse array of capabilities, such as image classification, speech recognition, and industrial optimization, while simultaneously establishing the groundwork for the development of deep learning paradigms.

Deep learning algorithms are based on Artificial Neural Networks (ANNs), which constitute mathematical constructs designed to replicate the operational principles of biological neurons.

1.1.2 CNNs

As previously introduced, ANNs are computational architectures composed of multiple layers of interconnected nodes, commonly referred to as artificial neurons. These networks typically consist of an input layer, one or more hidden layers, and an output layer, each serving distinct roles in the processing and transformation of data. In a classical ANN, each neuron in a given layer is connected to every neuron in the subsequent layer, forming a densely connected architecture. Although this framework allows for the modeling of highly complex relationships within the data, it also requires an extremely large number of parameters and resources. This is one of the critical points of ANNs, and can lead to scalability issues, an increased risk of overfitting, and high computational costs, especially when the input data are highly dimensional, such as images or multidimensional signals.

Convolutional Neural Networks (CNNs) are a class of deep learning models, particularly well suited for processing data with a grid-like topology, such as images and videos. They are widely used in tasks including image classification, object detection, medical image analysis, and video analysis, due to their ability to automatically learn spatial hierarchies of features through convolutional operations.

One of the key differences is that the neurons that the layers within the CNN are comprised of neurons organized into three dimensions, the spatial dimensionality of the input (height and the width) and the depth. The depth does not refer to the total number of layers within the ANN, but the third dimension of a activation volume. Unlike standard ANNs, the neurons within any given layer will only connect to a small region of the layer preceding it. In practice this would mean that for the example given earlier, the input 'volume' will have a dimensionality of 64 x 64 x 3 (height, width and depth), leading to a final output layer comprised of a dimensionality of 1 x 1 x n (where n represents the possible number of classes) as we would have condensed the full input dimensionality into a smaller volume of class scores filed across the depth dimension. [6]

CNNs adhere to the same overarching architecture as ANNs, so they have an input layer, one or more hidden layers, and an output layer. However, the key distinction lies in the structure of the hidden layers, which are not simply fully connected.

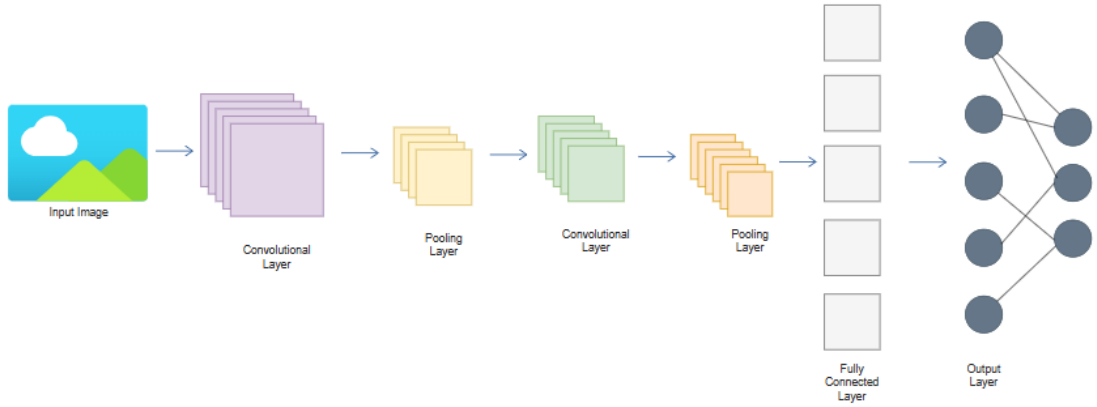


Figure 2: CNN architecture [7]

Figure 2 illustrates a generic architecture of a CNN, where there are various layers:

- **Convolution layers**: This layer is used to extract the feature from the input dataset. The input data are processed through one or more convolutional layers, where a set of trainable filters is applied to extract meaningful features. These layers are designed to identify local patterns such as edges, corners, and textures, which are essential for understanding the spatial structure of the input.
- **Activation function**: Activation functions are non-linear mathematical expressions and constitute an integral part of neural networks. It decides whether a neuron should be activated or not. This implies that the network evaluates the relevance of each neuron's input during the prediction process, determining whether it contributes meaningfully to the final output. There are several commonly used activation functions, one of the most well-known is the Rectified Linear Unit (ReLU).
- **Pooling layers**: Pooling layers are added in between two convolution layers with the sole purpose of reducing the spatial size of the image representation, hence reducing the memory used while training the network.
- **Fully connected layers**: The flattened feature maps are then passed through fully connected layers. They can process high-level features for prediction or classification.

1.1.3 Convolution

Convolution played a fundamental role in the preliminary phase of this research. A detailed understanding of the computational process and data flow was essential for developing effective strategies for memory management and partitioning. This foundational knowledge enabled the implementation of parallel computation techniques, ultimately contributing to improved performance and throughput.

As illustrated in Figure 2, convolution constitutes one of the core operations within CNNs. In mathematics (in particular, functional analysis), convolution is a mathematical operation on two functions f and g that produces a third function $f * g$, as the integral

of the product of the two functions after one is reflected about the y-axis and shifted. [8]

In the context of a CNN, the convolution operation combines the input data, typically pixel values from an image, with a kernel. The kernel is a small matrix of learnable weights that is systematically slid across the entire spatial extent of the input. At each position, it performs an element-wise multiplication with the overlapping region of the input, and the resulting values are summed to produce a single output value. This process is repeated across the entire input, generating a feature map that captures local patterns and spatial hierarchies within the data.

Figure 3 presents a simplified example of a two-dimensional convolution, illustrating the computational interactions between one single kernel and one single input. This representation is essential for understanding the mechanisms underlying memory partitioning.

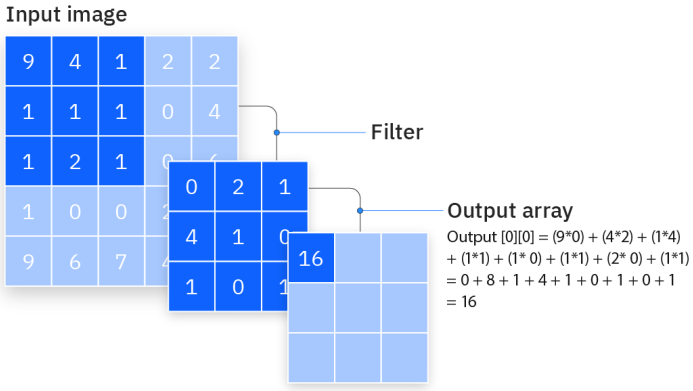


Figure 3: Example of convolution [9]

In CNNs, the input is a color image, consisting of a multi-channel, multi-filter 2D convolutions (standard 2D convolution). The input will have three dimensions: height, width, and depth, which correspond to the RGB in an image.

1.2 High-Level Synthesis

HLS represents a hardware design methodology that enables the description of digital circuits using high-level programming languages, such as C, C++, or SystemC. This approach facilitates the automatic translation of algorithmic descriptions into Register-Transfer Level (RTL) representations, which are compatible with FPGA or ASIC technologies, thus streamlining the design flow and enhancing productivity.

Figure 4 presents the typical workflow for designing hardware using HLS, outlining the key stages from algorithm specification to RTL generation.

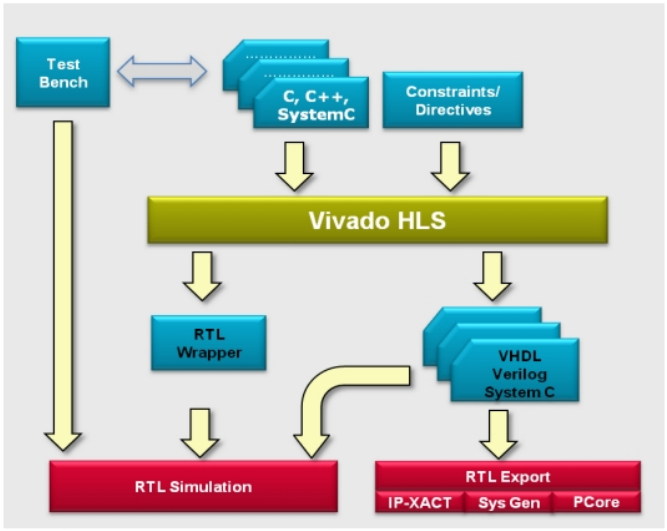


Figure 4: High-level synthesis workflow [10]

The workflow using HLS has the following steps:

- Write the algorithm at a high abstraction level using C/C++ with a target architecture in mind;
- Verify the functionality at the behavioral level, writing the Test Bench, using high-level programming languages (**csim**);
- Use the HLS tool to generate the RTL for a given clock speed, and design constraints;
- Verify the functionality of the generated RTL (**co-sim**);
- Explore different architectures using the same input source code.

The final step in the HLS workflow involves generating Bitstream, a binary file that configures the FPGA hardware. This is accomplished using vendor-specific tools like Xilinx Vivado, which translate the synthesized RTL into a format suitable for FPGA or ASIC programming.

1.2.1 HLS vs RTL

HLS serves as a crucial link between software-level algorithm modeling and hardware implementation, effectively abstracting the complexities associated with traditional hardware description languages. In conventional design flows, engineers are required to explicitly define architectural components, sequential and combinational logic, registers, and signal interactions. In contrast, HLS enables designers to focus on the functional behavior of the system using high-level programming languages, thereby significantly **reducing development time** and **accelerating the prototyping process**. Despite the abstraction offered by High-Level Synthesis, the designer retains the ability to guide the hardware implementation through the use of constraints and synthesis directives. These allow the tool to perform a range of architectural optimizations, including **pipelining**,

loop unrolling, **parallelization**, and **efficient memory management**, thus enhancing performance and resource utilization.

One of the primary **limitations** of HLS is that it may not yield hardware implementations as optimized as those produced through manually written RTL designs. This is largely due to the reduced level of control over architectural details, which are abstracted away in the high-level design process. As a result, the designer cannot explicitly define the underlying structure, including datapaths, control logic, and signal interactions. Moreover, effective use of HLS still demands a solid understanding of hardware architecture, as the designer must guide the synthesis process through the use of pragmas and directives to achieve acceptable performance and resource utilization.

In addition to the significant reduction in development time, a notable **advantage** of HLS is the abstraction away from manually implementing Finite State Machines (FSMs). Traditionally, FSMs are essential for modeling the behavior of digital circuits, capturing how a system transitions between states in response to specific inputs and conditions. In fact, for complex systems, state machines are often very complex and difficult to implement. HLS alleviates this burden by allowing designers to describe behavior using high-level constructs such as loops and conditional statements, with the synthesis tool automatically generating the corresponding control logic.

1.3 FPGA

An **FPGA** is a type of Integrated Circuit (IC) that can be reconfigured to perform specific hardware function. Unlike traditional processors (such as Central Processing Units (CPUs) or GPUs), which execute predefined instructions, an FPGA allows users to design and implement custom hardware architectures tailored to specific applications. FPGAs are ideal for the fastest growing applications today, such as edge computing, artificial intelligence, system security, 5G, factory automation, and robotics.

1.3.1 The Evolution of FPGA

Before the emergence of FPGAs, digital logic design relied on programmable devices such as Programmable Read-Only Memories (PROMs), Programmable Array Logics (PALs), and Programmable Logic Arrayss (PLAs). These early technologies used fuse or antifuse mechanisms to permanently configure logic functions, offering limited flexibility and reusability. The advent of FPGA occurred in 1985, when Xilinx introduced a novel architecture based on Static Random-Access Memory (SRAM), enabling dynamic reconfiguration of logic circuits. In subsequent years, alternative configuration technologies were developed, including antifuse-based FPGAs, characterized by One-time Programmable (OTP) logic, enhanced security, and resistance to radiation, making them suitable for aerospace and military applications, and Flash-based FPGAs, which are non-volatile and retain their configuration throughout power cycles.

Modern FPGAs have evolved beyond simple reconfigurable logic arrays. They now

incorporate a variety of embedded memory resources, such as **Block RAM (BRAM), Look-Up Table RAM (LUT-RAM)**, and **UltraRAM**, which support both configuration storage and runtime data handling. This integration allows FPGAs to function not only as collections of logic gates but as fully capable computing platforms capable of executing complex applications with dedicated internal memory.

1.3.2 FPGA structure

Compared to standard gate arrays, field-programmable gate arrays are larger devices. The basic cell structure for FPGA is bit more complicated than the basic cell structure of standard gate array. The programmable logic blocks of FPGAs are called logic blocks or **Configurable Logic Blocks (CLBs)**. The basic architecture of FPGA consists of an array of logic blocks with programmable row and column interconnecting channels surrounded by programmable **I/O blocks** as shown in Figure 5.

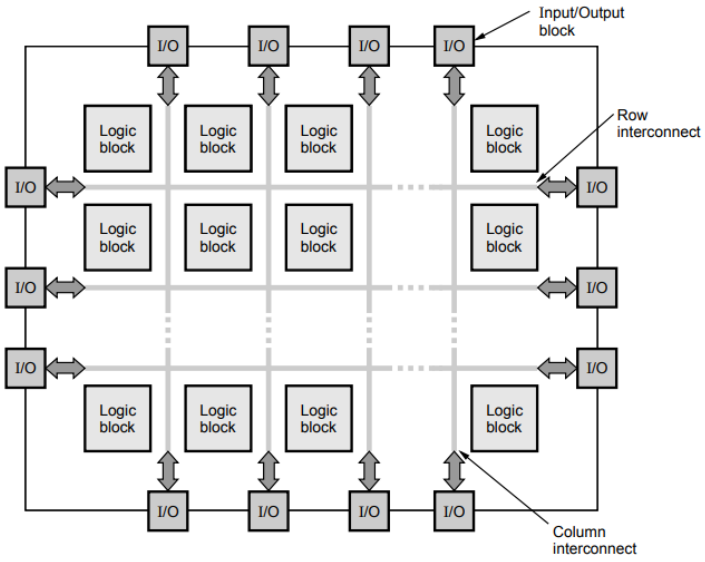


Figure 5: FPGA structure [11]

The main elements of an FPGA are the following:

- **CLBs**: These are the fundamental building blocks of an FPGA. Each CLB can be programmed to implement combinatorial, sequential functions and small memories. These blocks typically include a set of primitive logic elements, such as logic gates, small Look-Up Tables (LUTs), flip-flops, multiplexers, and other basic components. Each logic cell within a CLB generally comprises one or more programmable LUTs, which are responsible for implementing arbitrary Boolean functions. These LUTs are commonly paired with registers, enabling the storage of intermediate values and facilitating sequential logic operations. This modular structure allows for the flexible construction of complex digital circuits by combining multiple logic cells and configuring their interconnections.

- **Programmable Interconnects**: The programmable interconnection network constitutes a critical component of FPGA architecture, enabling the dynamic routing of signals between various internal modules, including CLBs, memory blocks, Digital Signal Processing (DSP) units, and Input/Output (I/O) blocks. This reconfigurable fabric allows user to define custom pathways and data flow within the FPGA.

- **Input/Output Blocks (IOBs)**: The IOBs are responsible for managing communication between the FPGA and external devices. Typically positioned along the perimeter of the chip, IOBs serve as the programmable interface between the internal logic, such as CLBs, BRAM, and DSP units and the physical pins of the device package. IOBs can be configured according to the designer's requirements in three distinct modes: input, when receiving signals from the external environment; output, when transmitting signals generated by the FPGA; and bidirectional, which allows the same pin to function as both input and output through tri-state logic. This last mode is particularly useful for implementing shared buses and half-duplex communication protocols. The flexibility of IOB configuration is essential for adapting the FPGA to a wide range of application-specific interfaces and standards.

- **Dedicated Blocks**: In addition to CLBs, modern FPGAs incorporate a variety of dedicated hardware resources designed to optimize the execution of computationally intensive and frequently used functions. Among the most significant of these are the BRAMs, which consist of high-speed SRAM memory blocks embedded directly within the FPGA fabric. Their proximity to the logic circuitry ensures extremely low access latency, often in the order of a few nanoseconds, which is significantly faster than external memory solutions. Another essential component is the DSP blocks, which are specialized arithmetic units integrated into the FPGA to accelerate complex mathematical operations. These blocks are optimized for tasks such as multiplication, accumulation, and Multiply-accumulate (MAC)) operations, making them particularly valuable in applications involving digital signal processing, image processing, and machine learning. The presence of these dedicated resources enables designers to achieve high performance while maintaining efficient use of logic and memory, contributing to the overall versatility and computational power of modern FPGA devices.

- **Embedded Processors / System-on-Chip (SoC) FPGA**: To address the increasing complexity of modern applications and the need to combine hardware flexibility with software processing capabilities, FPGA manufacturers have developed SoC devices that integrate general-purpose processors directly within the FPGA. These architectures consist of two main domains: the **Programmable Logic (PL)**, which includes CLBs, DSPs, BRAMs, and configurable interconnects used to accelerate specific algorithms and implement custom peripherals; and the **Processing System (PS)**, which incorporates one or more processor cores, typically ARM, along with Double Data Rate (DDR) memory controllers, standard interfaces (USB, Ethernet, SPI, I²C, UART), and in some cases embedded GPUs. This integration enables the development of heterogeneous systems that combine software programmability with hardware-level performance and parallelism.

1.4 Problem Statement

Today, many techniques exist for implementing neural networks on an FPGA or ASIC, and they're always evolving. Some can be performed at compile time, such as quantizing weights and activations to reduce numerical precision and use fewer resources, or pruning and compressing the model, which eliminates insignificant weights or connections (close to zero), making the model smaller and requiring fewer operations. This thesis focuses on **runtime strategies** that optimize how data are processed without altering the model itself, in order to implement neural networks on an FPGA.

In CNNs, the sheer volume of intermediate data creates a major memory bandwidth bottleneck. A core runtime technique to combat this is **data reuse**. Reuse of data in FPGAs is a key concept in optimizing hardware architectures for computationally intensive applications such as machine learning, signal processing, or computer vision. Simply put, it involves reusing data already loaded into fast memory (e.g. BRAM, URAM, or internal FPGA registers) to reduce the number of accesses to external memory (typically DRAM), which are costly in terms of both latency and power consumption.

Optimization techniques such as parallelism and loop unrolling enable the concurrent execution of multiple iterations, while pipelining structures the computation into sequential stages, allowing a new output to be generated at each clock cycle. High-Level Synthesis libraries and tools facilitate the transformation of models developed in frameworks like TensorFlow or PyTorch into efficient FPGA implementations, automating the trade-offs between parallelism, numerical precision, memory usage, and throughput.

With the rapid advancement of AI and ML, neural networks have become increasingly complex, resulting in a significant increase in the memory footprint required for weights and activations. This poses a major challenge when deploying such models on FPGAs, where resource constraints are a critical consideration. To address this issue, various optimization strategies have been developed to improve resource utilization. One of the most widely adopted approaches involves prioritizing input reuse over weight reuse, given that activations typically consume substantially more memory than convolutional kernels. Consequently, it is often more efficient to retain weights on-chip and iterate over the input data. However, due to the limited capacity of on-chip memory resources such as BRAM and URAM, it is not always feasible to load all kernels simultaneously. In such cases, a filter tiling strategy is employed, whereby a subset of kernels is loaded into the FPGA, reused across the input data, and subsequently replaced with the next batch of filters.

1.4.1 FPGA vs ASIC

Among a vast number of NN models, the CNN has a mainstream status in application such as image and sound recognition and machine decision. The convolution operation is the most complex and requires acceleration. A practical method is to optimize the architecture of the deep learning processor (DLP). The traditional CPU architecture lacks parallelism and memory bandwidth and is not suitable for CNN operations. Current researches are focused on GPU, FPGA and ASIC. GPU is the maturest and the most

widely applied, however it is not flexible and has high cost and energy consumption. Even though FPGA possesses **high flexibility** and **low energy consumption**, it is inferior in performance, in particular regarding the frequency, lower than CPUs. ASIC, due to targeted design, is advanced in performance and energy consumption. However, it is highly inflexible. [12]

This thesis focuses on the implementation of a neural network on an FPGA. The decision to utilize an FPGA instead of an ASIC is driven primarily by the need for **flexibility** and **rapid development**. FPGAs enable testing, validation, and, if necessary, modification of the architecture within a matter of minute, an advantage not afforded by ASICs. Although ASICs offer significantly higher performance compared to FPGAs, they are also more costly and require considerably longer development times. Moreover, the use of an FPGA facilitates the exploration of various architectural solutions and the progressive optimization of the design. Specifically, this thesis investigates **memory partitioning** as a means to parallelize computations, thereby increasing the number of operations per clock cycle and enhancing overall throughput.

1.5 Objectives

The primary objective of this thesis is to develop a method for partitioning input memory in such a way that an optimal unrolling factor is achieved. This enables the computation of an entire kernel window on the input data within a single clock cycle, thereby maximizing throughput. When combined with pipelining, this approach facilitates a continuous and efficient flow of data through the processing pipeline.

The proposed strategy involves storing the entire input and a single kernel within the FPGA, performing all necessary computations, and subsequently loading the next kernel for processing. This technique is particularly effective when the number of kernels is large but the input data can be fully accommodated within the FPGA's on-chip memory. Alternatively, a **hybrid approach** may be adopted: input reuse can be applied in the initial layers of the network, where input dimensions are substantial and the kernel-by-kernel strategy can be employed in later stages, where input sizes are reduced.

To support the implementation of various CNN components on FPGA, a set of templated functions, covering operations such as convolution, ReLU activation, and max pooling, has been integrated into the CEA Saclay framework. A suitable neural network architecture should be selected, a subset of its layers implemented, and resource utilization evaluated to validate the effectiveness of the proposed methods.

1.6 Thesis Structure

This thesis is composed of 7 chapters:

- **Chapter 1: Introduction**

This chapter provides an introduction to the work done, provides background and clarifications on the aspects developed in the thesis, specifies the problems in the strategies used nowadays and specifies the objectives of the thesis.

- **Chapter 2: Aidge**

  This chapter explains what CEA Saclay's Aidge project consists of, and how the work of this thesis is integrated into their framework.

- **Chapter 3: Memory Partitioning**

  This chapter addresses one of the main objectives of the thesis: the study of partitioning the memory containing activations. It also shows how it is implemented and the importance of partitioning for improving performance and throughput.

- **Chapter 4: Implementation**

  This chapter provides an explanation of how the implementation of some layers of the neural network is structured, in particular, the templated functions used such as convolution, ReLU and maxpool.

- **Chapter 5: FPGA Design and Synthesis with Vivado**

  This chapter addresses the FPGA implementation of the project. The workflow in Vivado is described in detail: construction of the Block Design by connecting the IP generated with Vitis HLS to existing IP cores (e.g., AXI Direct Memory Access, AXI Interconnect), an explanation of each block's role, and the overall Vivado flow. In particular, the chapter covers logic synthesis, implementation, and bitstream generation, the configuration file used to program the FPGA. It then explains how the bitstream is loaded and how the software drivers for FPGA interfacing are created using PYNQ. The characteristics of the target FPGA are summarized. Finally, the chapter details the dataflow among the various functions using a ping–pong buffer and presents Vivado waveforms that validate the implementation.

- **Chapter 6: Results on ResNet-18**

  This chapter presents the results obtained by implementing layers of a ResNet-18 on an FPGA (AMD ZCU102). It reports resource utilization from the post-synthesis report generated by Vivado. In addition, it details board-level power consumption and performance metrics, specifically latency and throughput.

- **Chapter 7: Conclusions**

  This chapter provides a comprehensive summary of the research conducted, outlining the potential applications of the developed solution. It presents the key results achieved, critically examines the limitations inherent in the proposed methodology, and suggests possible directions for future improvements and refinements.

2 Aidge

The CEA Paris-Saclay center is one of nine centers belonging to the French Alternative Energies and Atomic Energy Commission (CEA). One of the projects of which CEA is coordinator is **NEUROKIT2E**.

2.1 NEUROKIT2E

NEUROKIT2E is a European project that aims at proposing a Deep Learning Platform for Embedded Hardware around an established European value chain (providing AI hardware and software). This open-source platform will provide the necessary tools for Europe to play on the same level with its American and Chinese competitors, and take the lead on a competitive aspect (still underdeveloped but which will quickly prove to be essential): embedded AI. [13] NEUROKIT2E is a European Union funded initiative under the Horizon Europe program (Chips Joint Venture), involving five member states, France, the Netherlands, Austria, Germany, and Italy, and a consortium of universities, research institutions, and private enterprises. The project is coordinated by CEA, with participation from leading European companies including Thales, Infineon, STMicroelectronics, TTTech Auto AG, and Dolphin Design SAS.

The overarching objective of NEUROKIT2E is to develop an open-source platform for deep learning on embedded hardware, capable of rivaling existing American and Chinese solutions in the domain of edge AI. The project aims to strengthen European technological sovereignty by equipping industry with advanced tools for the design, optimization, and deployment of neural networks on resource-constrained devices. At its core, the platform builds upon **Neural Network Design & Deployment (N2D2)**, a framework developed by CEA for neural network modeling and implementation. A key ambition of NEUROKIT2E is to integrate hardware-level abstractions with neural network architectures, enabling holistic optimization and providing a unified, end-to-end development pipeline tailored for embedded AI applications.

2.2 Aidge Framework

NEUROKIT2E emerged as a natural continuation and expansion of the work previously carried out by CEA Saclay on the N2D2 framework, an open-source platform dedicated to the design and deployment of neural networks on embedded hardware. The project's ambition was to industrialize this technology and enhance its accessibility by developing a more comprehensive and modular ecosystem. As part of this evolution, CEA introduced **AIDGE** within the NEUROKIT2E initiative, positioning it as the successor to N2D2 and laying the foundation for a next-generation platform tailored to the demands of edge AI.

Aidge is an open-source deep learning platform specialized in the design of deep neural networks intended to operate in systems constrained by power consumption or dissipation, latency, form factor (dimensions, size, etc.), and/or cost criteria. [14]

The rise of AI and Deep Learning has revolutionized data processing paradigms. Traditionally in the cloud, neural networks are now destined to be embedded directly in the device making them intelligent and autonomous. However, this shift presents a significant challenge: how can large and complex models be adapted for deployment on resource-constrained hardware? Moreover, what trade-offs must be considered to optimize performance in terms of computational resources, development time, and overall project cost? Aidge is a software solution for optimizing and deploying neural networks on embedded targets. It is interoperable with standard AI and modular for easy integration and scalability. Aidge offers innovative quantisation and compression techniques to reduce model complexity and memory requirement for optimal use of architecture resource, whether on an MCU, GPU, FPGA, or ASIC solutions.

In addition, Aidge promotes code transparency and offers full control over application development, minimizing reliance on proprietary tools and enhancing flexibility for developers and system integrators.

In Figure the Aidge project workflow is depicted in a clear and structured way.

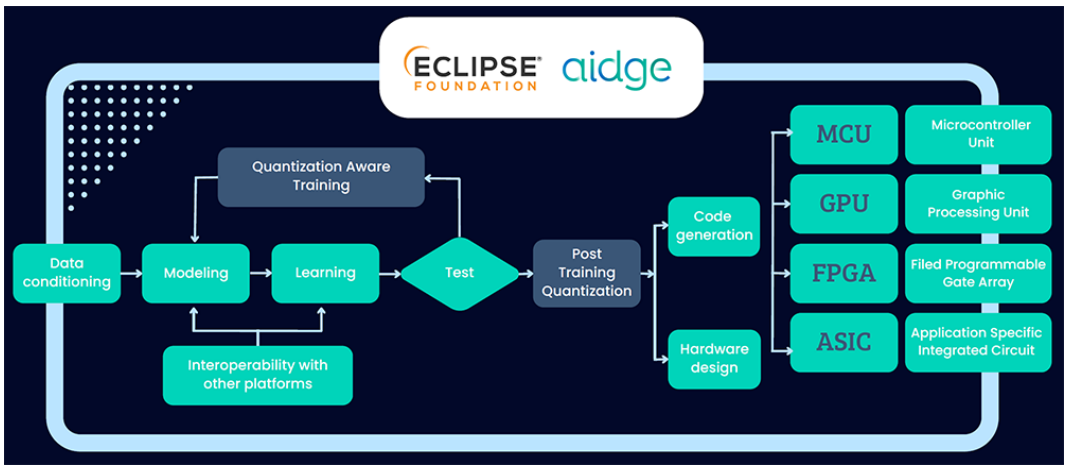


Figure 6: Artificial Intelligence Development Flow on Embedded Hardware [15]

The image shows a flowchart illustrating the main steps for designing, quantizing, and deploying AI models on resource-constrained hardware platforms. The process is divided as follows:

- **Data Conditioning**: This involves preprocessing the raw data, with operations such as normalization, noise reduction, and formatting, to make it suitable for model training.
- **Modeling**: This phase defines the neural network architecture, selecting the type of layers, activation functions, depth, and complexity of the model based on the task at hand.
- **Learning**: This is the actual training process, during which the model learns from the data through optimization and backpropagation algorithms.

- **Testing**: After training, the model undergoes a validation phase to verify its performance on unseen data, evaluating metrics such as accuracy and precision score. This step is crucial before proceeding with quantization.
- Next, there are two quantization techniques:
  - **Quantization Aware Training (QAT)**: A technique that incorporates quantization directly into the training process. It simulates the effects of precision reduction (e.g., from float32 to int8) during training, allowing the model to adapt and maintain high performance even after conversion.
  - **Post-Training Quantization (PTQ)**: Applied to already-trained models, this technique converts weights and activations into low-precision formats without retraining. It is simpler to implement, but can result in a greater loss of accuracy than QAT.
- Finally, there is hardware deployment:
  - **Hardware Design**: The model is adapted to the specifications of the target device, taking into account available resources (memory, computing power, bandwidth) and architectural characteristics.
  - **Code Generation**: Aidge automatically generates code optimized for the target hardware, reducing development time and minimizing manual errors. Deploying the model across different types of embedded hardware: MCU, GPU, FPGA, or ASIC.

2.3 Embedding Thesis Work into the Aidge Framework

This thesis contributes to the Aidge project by **integrating custom HLS code** for the implementation of various neural network components on FPGA-based embedded platforms. Specifically, a set of templated functions has been developed to support core operations commonly found in convolutional neural networks, including convolution, ReLU activation, and max pooling. These functions are designed to be modular and reusable, aligning with Aidge's scalable architecture.

The following sections detail the practical work carried out, beginning with an analysis of memory partitioning strategies to optimize data access and parallelism. A filter reuse strategy was adopted, wherein all input data is stored within the FPGA's internal memory to minimize external memory access and improve throughput. To support the deployment of larger networks and enhance performance, a hybrid approach was implemented: input reuse was applied in the initial layers, where input dimensions are large, and kernel reuse was introduced in later layers, where kernel size begins to dominate memory usage.

To validate the proposed methodology, several layers of a **ResNet-18** architecture were implemented and tested. The results include a detailed evaluation of resource utiliza-

tion (e.g., BRAM, DSPs, LUTs) and performance metrics, demonstrating the feasibility and efficiency of the integration within the Aidge framework.

3 Memory Partitioning

This chapter outlines the **methodology** adopted for memory partitioning. Following an initial investigation of convolution operations and data movement, aimed at optimizing resource utilization and computational efficiency, a strategy was defined to partition the input memory. This approach enables **parallel execution** of operations and contributes to improved system performance. The ultimate objective of this work is the implementation of neural network layers on an FPGA. Although FPGAs generally offer lower performance compared to ASICs, they provide greater flexibility and faster development cycles. To mitigate performance limitations, optimization strategies such as parallelization and pipelining are employed, enabling more efficient computation and improved throughput.

3.1 Strategy adopted

Among the runtime strategies for implementing neural networks on FPGAs, one of the most commonly used is data reuse. Specifically, this thesis adopts **weights reuse**. This choice stems from the adoption of a mixed approach to implementing a neural network. This type of approach consists in using the input reuse strategy at the beginning, where the input is large (and cannot be entirely stored on the FPGA due to hardware resource limitations) and the number of kernels is small. Toward the end, where the input becomes smaller and the number of kernels too large to be stored on the FPGA, the strategy on which this thesis is based is adopted, an innovative form of weights reuse. This consists in storing the entire input in the internal memory of the FPGA, avoiding external reads from slower external memory, and loading one kernel at a time. This allows all computations to be performed with the available kernel, and subsequently loading the next one into internal memory.

The advantages of this approach lie in the significant **reduction of external memory accesses**, which have notable implications on performance, energy consumption, and latency. In particular, latency becomes a critical factor, as external memories such as DDR exhibit considerably longer access times compared to BRAM or URAM, which are embedded within the FPGA. Furthermore, when the system relies heavily on external memory, it becomes challenging to maintain deep pipelines or perform parallel computations without encountering execution stalls.

3.2 Loop Unrolling & Pipelining

Among other runtime optimization techniques, in addition to data reuse, there exist methods that govern how data is processed and restructure the hardware architecture to achieve enhanced performance, while accommodating the simultaneous use of multiple resources. Notable among these are **loop unrolling** and **pipelining**, which are widely

adopted to increase parallelism and reduce execution latency in FPGA-based implementations. Furthermore, Vitis HLS enables the implementation of these techniques through the use of **directives**, also referred to as "pragmas", which are discussed in greater detail in the following sections.

3.2.1 Loop Unrolling

Loop unrolling is a widely used technique in programming and is based on the transformation of a loop. Specifically, as the word itself suggests, it involves unrolling the loop to optimize execution speed by performing multiple operations in parallel, while accepting the use of multiple resources. Indeed, with more hardware, the compiler can perform the loop operations in a single clock cycle. If the loop consists of many iterations and the hardware resources to perform all the operations simultaneously are not available, loop unrolling can be applied with a "k" factor, so that the compiler performs k operations in a single clock cycle.

The transformation can be undertaken manually by the programmer or by an optimizing compiler. In Vitis HLS, loop unrolling can be applied through the use of the *<#pragma HLS UNROLL>* directive, which is placed inside the loop intended for unrolling. This directive instructs the compiler to replicate loop iterations, enabling parallel execution and improving performance during hardware synthesis. The following examples illustrate two loop unrolling strategies in Vitis HLS. Listing 1 demonstrates a full unroll, where all iterations are expanded, while Listing 2 applies partial unrolling with a factor of 4, enabling a balance between performance and resource utilization.

Listing 1: Complete Unrolling

```
for(int i = 0; i < 8; i++) {
    #pragma HLS UNROLL
    c[i] = a[i] + b[i];
}
```

Listing 2: Unrolling with factor 4

```
for(int i = 0; i < 8; i++) {
    #pragma HLS UNROLL factor=4
    c[i] = a[i] + b[i];
}
```

Although both approaches yield the same computational result, they differ significantly in terms of hardware resource utilization and execution latency. Without loop unrolling, the compiler schedules one operation per clock cycle, resulting in a total of 8 cycles to complete the loop. In contrast, applying full unrolling enables the compiler to execute all 8 operations in parallel, reducing the execution time to a single cycle and effectively eliminating the loop structure. When hardware resources are constrained, partial unrolling can be employed. For instance, specifying an unroll factor of 4 allows the compiler to perform 4 operations concurrently, completing the loop in 2 cycles and offering a balanced trade-off between performance and resource consumption.

3.2.2 Pipelining

Pipelining is a widely adopted optimization technique in HLS. It involves partitioning the execution of an operation into multiple sequential stages, thereby enabling concurrent execution of multiple loop iterations, each occupying a distinct stage within the

pipeline. In a non-pipelined loop, iterations are executed sequentially, with each iteration commencing only after the previous one has completed. By contrast, pipelining allows iteration i+1 to begin while iteration i is still in progress, effectively overlapping execution and increasing throughput. Although pipelining does not reduce the latency of individual operations, it significantly decreases the overall execution time of the loop. Unlike loop unrolling, which replicates hardware to achieve parallelism, pipelining preserves the original loop structure and achieves performance gains with minimal hardware duplication.

The figure 7 illustrates the timing requirement for a pipelined and a non-pipelined implementation.

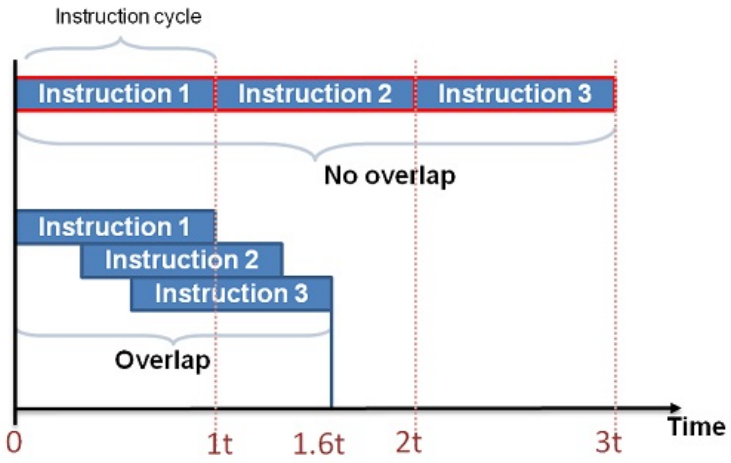


Figure 7: Pipelined vs Non-Pipelined Instruction Execution [16]

In the example presented, each instruction requires a time duration of t for execution. In the non-pipelined implementation, the instructions are executed sequentially, resulting in a total execution time of 3t for three operations. In the pipelined implementation, the operations are distributed across multiple pipeline stages, allowing overlapping execution. As a result, the total time required to complete the same three instructions is significantly reduced to approximately 1.6t. This demonstrates the advantage of pipelining in improving throughput by enabling partial parallelism, even though the latency of individual instructions remains unchanged.

Pipelining can be implemented in HLS through the use of the directive *<#pragma HLS pipeline II=1>*, placed within the loop to be optimized. Parameter **II**, which stands for **Initiation Interval**, defines the number of clock cycles that must elapse before a new iteration of the loop can be initiated within the pipeline. A lower initiation interval corresponds to a higher frequency of iteration launches, thus increasing the overall throughput of the design. Specifically, setting II = 1 allows a new iteration to begin at every clock cycle, indicating that the pipeline is operating at its maximum capacity. This configuration enables efficient parallelism while preserving the loop structure, making it particularly advantageous in resource-constrained hardware environments.

3.3 Parallelization strategy

This section outlines the strategy adopted to partition the internal activation memory of the FPGA. As discussed previously, one challenge with FPGAs is limited resources, especially in terms of BRAM. The use of FPGAs in conjunction with HLS enables the implementation of various optimization techniques, such as loop unrolling and pipelining, which were introduced in the previous section.

Vitis HLS is a development tool capable of translating high-level programming languages, such as C++, into RTL. In order to facilitate effective optimization and minimize synthesis and generation time, it is crucial that the developer possesses a conceptual understanding of the target hardware architecture and the behavior of the applied optimization techniques. This awareness is particularly important when adhering to design constraints. For example, in the context of pipelining, if I choose II=1 as the target, it implies that new data must be available at every clock cycle. Failure to meet this requirement results in II violations, indicating that the desired throughput cannot be sustained and leading to a degradation in overall performance.

To understand the strategy adopted for partitioning the internal memory of the FPGA, it is first necessary to examine the convolution operation (as described in Section 1.1.3), which constitutes one of the fundamental building blocks of CNNs. The primary objective is to ensure that the data required for each convolutional kernel window are distributed across multiple memory blocks, allowing the computation of a full window of activations within a single clock cycle. A key challenge addressed in this context was the mitigation of data dependencies and a strategically placing activation data in distinct memory locations, in order to enable **parallel access** and efficient computation of convolutional windows.

3.3.1 Preliminary Memory Layout

The following example provides a visual and conceptual representation of how memory partitioning is performed. In CNNs, the input is typically derived from an image and is therefore structured as a three-dimensional tensor comprising height, width and depth. As a result, the convolution operation is inherently three-dimensional. The data is organized in HWC format, a convention widely adopted in computer vision and deep learning applications. Where HWC stands for height, the number of rows in the image; width, the number of columns; and channels, the number of feature channels. This format facilitates efficient access and manipulation of image data during convolutional processing. Figure 8 illustrates the input and kernel structure:

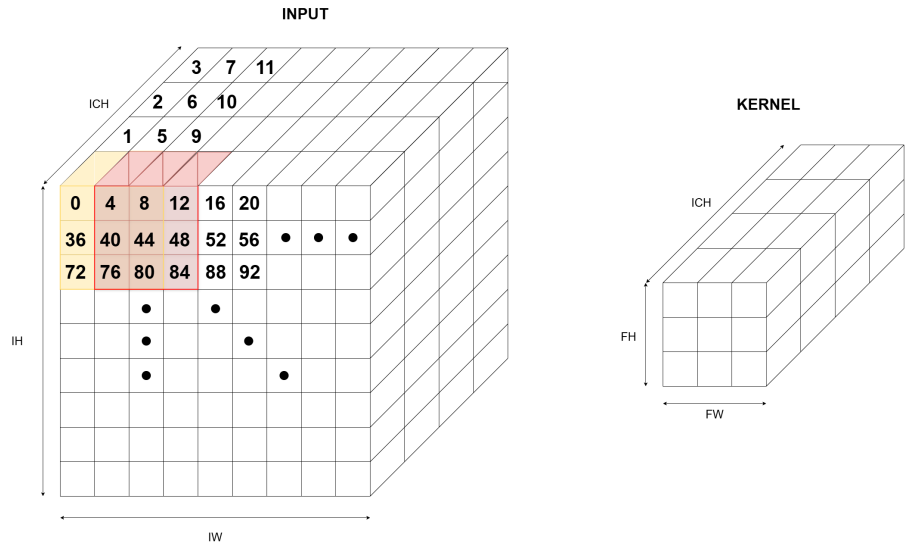


Figure 8: Input and Kernel structure

The input dimensions, denoted as $\boldsymbol{IH \times IW \times ICH}$, can be observed on the left side of the figure. In this simplified example, the input tensor has dimensions $9 \times 9 \times 4$, corresponding to the height, width, and input channels, respectively. For ease of debugging, the input data were entered in sequential order starting from 0. In this way, it was easier to identify errors in the final memory. On the right side, a convolutional kernel is depicted with dimensions $\boldsymbol{FH \times FW \times ICH}$, which in this case is equal to $3 \times 3 \times 4$. This kernel operates across the spatial and channel dimensions of the input, enabling the computation of a single output activation through element-wise multiplication and accumulation. The goal is to store all the data used in the convolution in separate memories. Therefore, for the first window, given that the filter is 3×3, nine distinct memories are required. This applies to the first channel, and the same holds for the other channels. Specifically, it can be observed that the data contained in the first 3×3 input window are all stored in distinct memory locations (highlighted in yellow). Furthermore, a specific pattern can be noticed: the first memory vector contains the first pixel of the first four channels, since ICH $= 4$ (containing 0, 1, 2, and 3). Subsequently, the first pixel is stored at position $4 \times 3 = 12$, which corresponds exactly to the first pixel of the input window shifted by three positions, as FW $= 3$. In this way, it is evident that even when considering the data from the second window, highlighted in red, they are still mapped to distinct memory locations.

Figure 9 shows the partitioned memory of the example analyzed in this case. It can be represented as a 2D memory, where the height is determined by the formula $\boldsymbol{FW \times FH}$, since it depends on the kernel size, which in this case is 3×3. The other dimension is given by the formula $\boldsymbol{INPUT\_SIZE/(FW \times FH)}$. Where $INPUT\_SIZE = FW \times FH \times ICH \times WINDOW\_IN^2$.

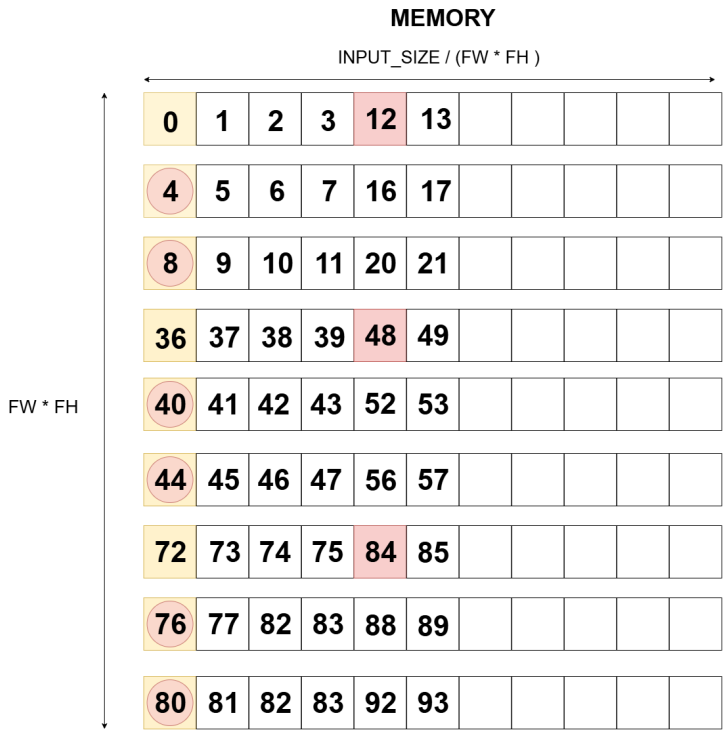


Figure 9: Preliminary Memory Layout

An important note is that, to avoid complicating the computational indexing, the on-chip memory was dimensioned with a small overhead by assuming that the input size is an integer multiple of the filter size, even when it is not. This is achieved by introducing the parameter **WINDOW_IN**, which effectively counts how many times the filter fits into the input tensor; if the division is not exact, it is rounded up to the next integer. For clarity, consider two examples with a 3×3:

1. **Input multiple of the filter**: size 9×9, in this case $9/3 = 3$. Being multiple no memory overhead.

2. **Input not multiple of the filter**: size 7×7; here $7/3 = 2,33$, which is rounded up to $\lceil 7/3 \rceil = 3$

In this way, a small area overhead is accepted in order to keep the index calculations for memory reads/writes simple, while aiming to minimize DSP usage.

Another aspect to note is that the numbers in the first input window have been highlighted in yellow, to emphasize that they are all stored in different memories. The same applies to the second window, highlighted in red, where it can be observed that the data not previously considered are still located in separate memories, such as the numbers 12, 48, and 84. The selection of data to be loaded into memory for performing the convolution is carried out through a specific calculation within the code, designed to account for all possible cases.

3.3.2 HLS Partitioning

As previously mentioned, the memory is two-dimensional. To ensure that the compiler recognizes them as separate memories, Vitis HLS provides the specific directive

#pragma HLS ARRAY_PARTITION. This partitioning generates RTL with multiple small memories rather than a single large one. It also improves performance by enabling simultaneous access and thus a more efficient pipeline, reducing the initiation interval.

The pragma must be inserted into the C source code within the scope of the function where the array variable is defined, with the following options:

```
#pragma HLS array_partition variable=<name> type=<type>
    factor=<int> dim=<int> off=true
```

In which:

- **variable**=< *name* >: specifies the name of the variable to be partitioned.
- **type**=< *type* >: specifies the type of partitioning; by default, it is complete.
  - Complete: complete partitioning decomposes the array into individual elements.
  - Cyclic: cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by placing one element into each new array before returning to the first array, repeating the cycle until the array is fully partitioned.
  - Block: block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks, where N is the integer defined by the factor= argument.
- **factor**=< *int* >: specifies the number of smaller arrays to be created. [17]
- **dim**=< *int* >: specifies which dimension of a multi-dimensional array to partition. It is given as an integer from 0 to N, for an array with N dimensions. If 0 is chosen, all dimensions of the multi-dimensional array are partitioned using the specified type and factor options.
- **off**=true: disables the ARRAY_PARTITION feature for the specified variable. It cannot be used with dim, factor, or type.

3.3.3 Optimized Memory Layout with Channel Parallelism

To maximize the utilization of the FPGA for **performing more parallel computations** and thereby increasing throughput, another parameter called **ICH_PAR** was introduced. As the name suggests, this parameter specifies the degree of parallelism of the input channels. Setting its value to 1 corresponds exactly to the case previously considered. However, increasing its value raises the number of distinct memories required to carry out the computations. The formulas presented earlier have been modified to incorporate the ICH_PAR parameter.

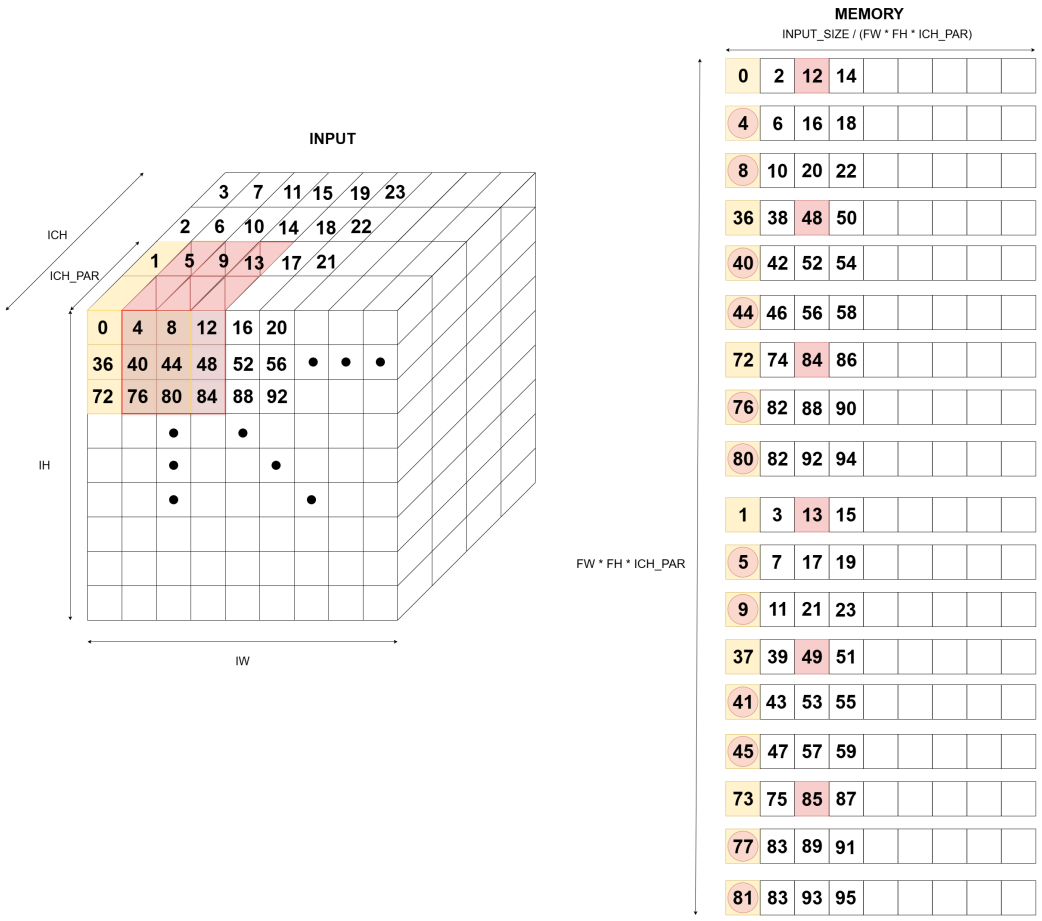


Figure 10: Optimized Memory Layout with Channel Parallelism

Figure 10 shows the memory partitioning implemented with ICH_PAR. The kernel remains unchanged from the previous case. What differs here is that the memories are **doubled**; in fact, the computation is performed on **two channels at a time** rather than on a single channel. As a result, the computation time decreases and the throughput increases, thereby improving overall performance. In this case, the height is calculated using the formula $\boldsymbol{FH \times FW \times ICH\_PAR}$. Setting the ICH_PAR parameter to 2 results in twice the number of memory banks compared to the previous case. As for the depth of the memories, it is naturally reduced, since the other dimension has been increased of ICH_PAR times.

3.4 Auto-Partitioning Function Template

A templated function, called **input_2conv**, has been implemented to output partitioned memory according to the specified parameters. The complete function is available in the GitHub repository referenced in Appendix 7.1, while the function template is shown below:

```
template< int ICH, int IW, int IH,
    int FW_IN, int FH_IN,
    int OCH, int OW, int OH,
```

```
    int ICH_PAR_IN,
    int WINDOW_IN>
void input2conv(hls::stream<mem_in_t>& memory_in_stream, memory_in_conv_t<CONV_0_FW,
    CONV_0_FH, CONV_0_ICH_PAR, CONV_0_ICH, WINDOW_IN>& memory_in_local)
```

The template parameters represent:

- **ICH**: number of input channels in the tensor.
- **IW, IH**: width and height of the input tensor.
- **FW_IN, FH_IN**: width and height of the kernel.
- **OCH**: number of output channels.
- **OW, OH**: width and height of the output feature map.
- **ICH_PAR_IN**: parallelism on the input channels (i.e., how many channels can be read simultaneously).
- **WINDOW_IN**: spatial parallelism on the windows (i.e., how many windows can be processed in parallel).

In this thesis, the input data are in HWC format and are supplied to the input function through a stream from external memory. Since the stream handles one data point at a time, the function can achieve an initiation interval of 1, thereby reading and processing one data point per clock cycle without the need to wait for all the data to be loaded. The input data type, mem_in_t, is defined as a $\boldsymbol{ap\_uint < 8 >}$. The data type $ap\_uint < 8 >$ is an unsigned arbitrary-precision integer of width 8 bits, provided by the HLS library. It is widely used in HLS because, unlike the standard $uint8\_t$, it synthesizes to RTL ports and buses with an exact width of 8 bits. Furthermore in this way, the data type has been made generic, and if a modification is required, it is sufficient to change this data-type in the file parameter.h, which contains all the parameters and data types used in the various functions.

As noted earlier, the function returns the partitioned memory, which is then provided to the convolution by another routine that adds stride and padding (see Section 4.2.3). This routine is likewise defined with a template parameter and yields a two-dimensional structure, exactly as described previously: the vertical dimension is $FW \times FH \times ICH\_PAR$, while the horizontal dimension is given by $FW \times FH \times ICH\_PAR/(FW \times FH \times ICH\_PAR \times WINDOW\_IN^2)$. In which the filter dimensions (FW and FH), the input channel parallelism and the number of channels (ICH_PAR and ICH), and finally the parameter WINDOW_IN was introduced to handle cases where the input dimensions are not multiples of those of the kernel. Specifically, it represents how many times the filter fits into the input and has been useful in avoiding computational complexity in the calculation of the memory index when this condition occurs.

The body of the function aims to increment two indices in order to properly place the data in memory. Particular attention is given to positioning the data so that the required convolution calculations can be performed in a single clock cycle, also taking into account the input parameters.

3.5 Design Constraints and Trade-offs

The partitioning strategy was determined by several design constraints inherent to FPGA implementations. In particular, the limitations in the number of available BRAM and DSP resources required a careful balance between maximizing parallelism and maintaining overall resource feasibility. Moreover, the design was conceived to achieve an initiation interval of 1, thereby ensuring the absence of data dependencies in convolution and other functions. These considerations highlight the intrinsic trade-off between performance and resource utilization: although aggressive partitioning and parallelization can enhance computational efficiency, they may also result in excessive hardware consumption. Consequently, the adopted design represents a compromise that guarantees both the functional correctness of the system and the practical feasibility of its implementation on the selected FPGA device.

In the subsequent chapters, the convolutional neural network implemented in this work is presented, with particular attention devoted to ResNet-18. The template functions are described in detail, along with an explanation of the various parameters that characterize the implemented layers. Finally, the design is synthesized and deployed on the FPGA in order to assess both the resource utilization and the resulting performance.

4 Hardware Implementation in HLS

In this section, the hardware implementation of CNN is described using HLS. HLS allows FPGA circuits to be designed directly from high-level C++ descriptions, significantly reducing development effort and enabling rapid design-space exploration compared to hand-written RTL. The implementation focuses on **ResNet-18**, one of the most widely used convolutional neural networks, particularly in the field of computer vision. Special attention is then given to the operations comprising the implemented layer and to the overall dataflow between them. The implemented functions also support other neural networks, including **LeNet**. However, because ResNet-18 exhibits greater complexity, it was selected as the primary case study for this thesis.

4.1 ResNet-18

ResNet-18 is a member of the Residual Network (ResNet) family, built from **residual blocks** that alleviate the vanishing-gradient problem and allow training of deeper architectures. Vanishing gradients arise during backpropagation when derivatives shrink as they are propagated toward earlier layers, particularly in very deep networks, so those layers receive near-zero updates and learning stalls. As its name suggests, ResNet-18 comprises eighteen weighted layers and is the shallowest of the standard ImageNet ResNets, making it a common baseline for rapid experimentation, deployment on modest hardware, and instructional use. Rather than forcing each block to learn an entirely new mapping *H(x)*, ResNets learn a residual function *F(x) = H(x) - x* and produce the output $y = x + F(x)$ via identity skip connections that bypass one or more layers; these shortcuts preserve gradient flow and simplify optimization. Figure 11 shows the block diagram of the skip connection mechanism.

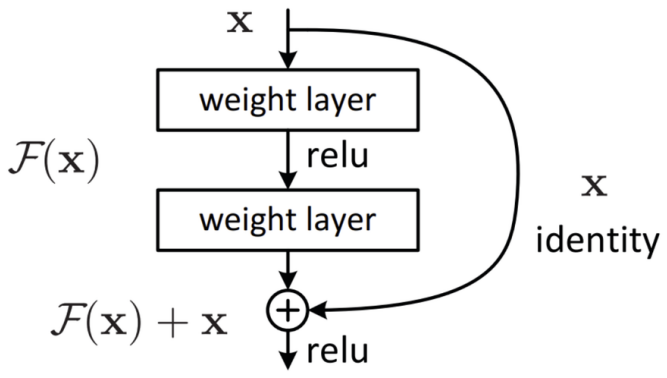


Figure 11: Residual block with identity skip connection [18]

In this thesis, following an analysis of memory partitioning strategies, the core operators of ResNet-18 are implemented. Figure 12 presents a block-level overview of the entire network. The study focuses on the final residual stage, which comprises 512 filters.

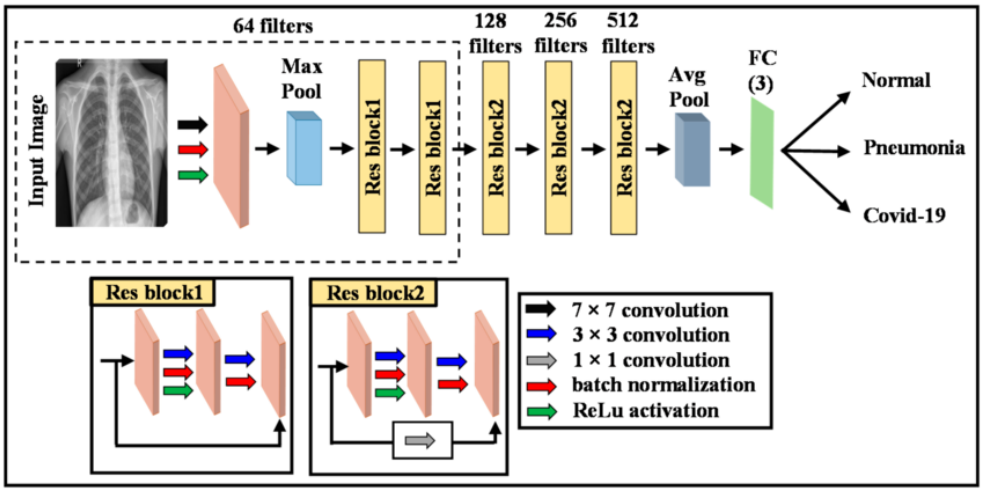


Figure 12: ResNet-18 architecture [19]

4.2 Convolution

The operating principle of convolution was introduced in Section 1.1.3. In ResNet-18, the operator is a standard 2D convolution (multi-channel and multi-filter): the kernel slides over the two spatial dimensions (H and W) while spanning the entire input-channel dimension.

Figure 13 illustrates an example of a 2D convolution with **ICH** input channels and **OCH** output channels. The number of input channels matches the channel dimension of the filters, since each filter aggregates weighted sums across all input channels and projects the result into a single value for each sliding window position. Furthermore, there are OCH distinct filters, and each filter produces a partial-sum feature map of size $OW \times OH$ (output width and output height). Finally, all partial results are collected to form the final output tensor of dimension $\boldsymbol{OCH \times OW \times OH}$.

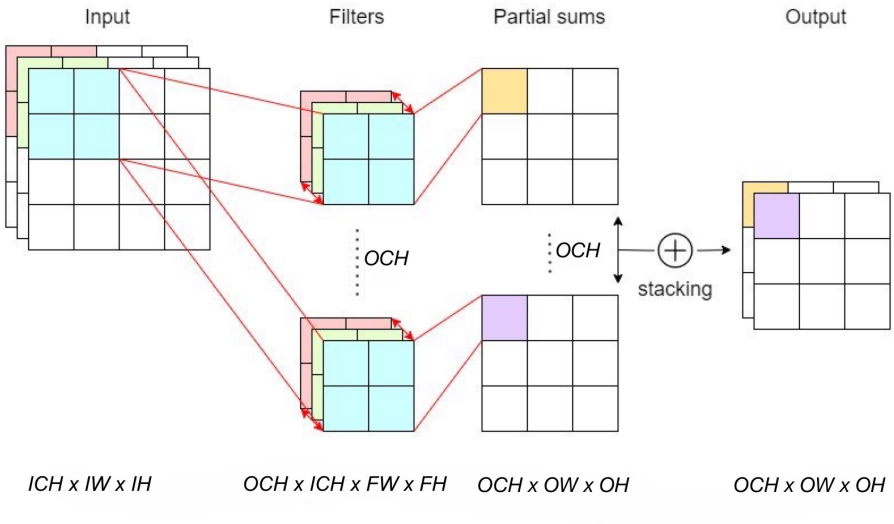


Figure 13: Standard Convolution [19]

The dimensions of the output feature map are given by the following equations:

$$OW = \frac{IW + 2 \times PADDING - FW}{STRIDE} + 1 \tag{1}$$

$$OH = \frac{IH + 2 \times PADDING - FH}{STRIDE} + 1 \tag{2}$$

Two additional parameters appear in these expressions: **stride** and **padding**. When used together in convolutional layers, they provide fine-grained control over both the spatial size of the output and the ability of the filter to capture contextual information. The following subsections describe these parameters in detail and indicate how they have been integrated into the implementation.

4.2.1 Stride

The **stride** parameter defines the step size with which the kernel moves across the input. During convolution, the stride specifies how many positions the filter shifts at each step. The operation typically begins with the convolution window placed at the top-left corner of the input tensor, and the kernel is then slid across all positions, both horizontally and vertically. While a stride of one corresponds to shifting the window by a single element at a time, in practice larger strides are often employed, either for computational efficiency or to intentionally reduce the spatial resolution by skipping intermediate positions.

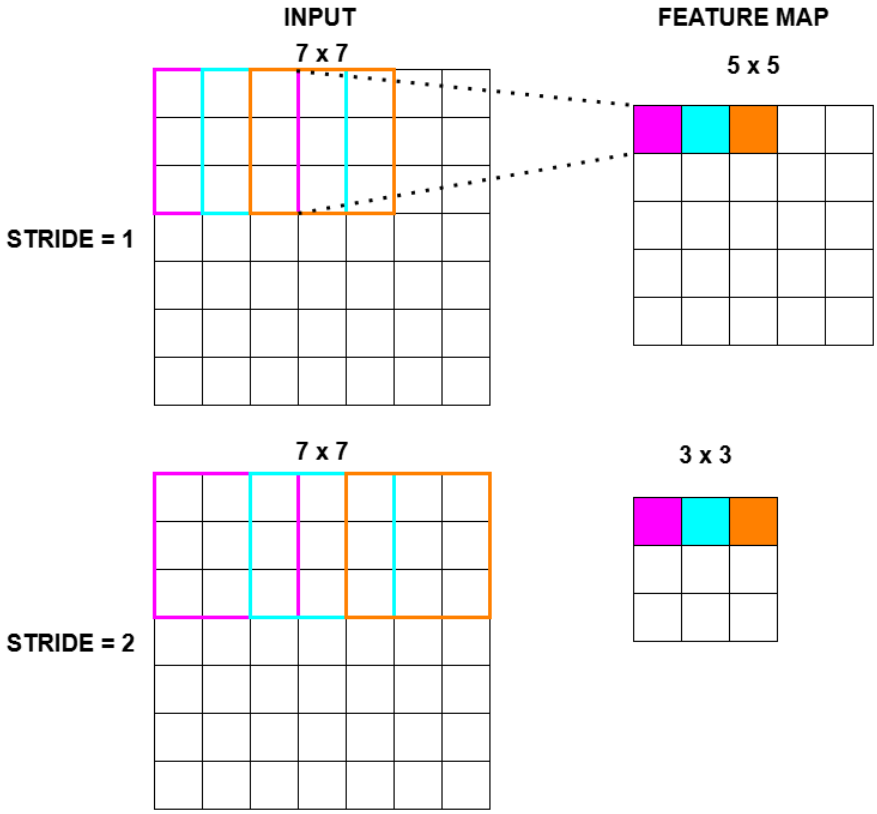


Figure 14: Stride effect on Convolution

Figure 14 shows two examples where stride is set to 1 and 2, respectively, highlighting how the output dimensions are affected. From Equations 1 and 2, it can be observed that there is an inverse relationship between the stride and the spatial size of the output feature map.

The choice of stride impacts several aspects of the convolution. First, as noted, it determines the spatial resolution of the output. In addition, it affects computational efficiency: a stride greater than 1 increases the step size of the kernel, which reduces the number of operations and thus the computational cost. However, this efficiency gain comes at the expense of potential information loss. By skipping positions in the input, the kernel may fail to capture fine-grained features that could be important for accurate representation.

4.2.2 Padding

As can be observed from Equations 1 and 2, the dimensions of the output feature map tend to shrink when padding is not applied. The primary goal of the network is to extract meaningful features from the image through convolutional layers. However, important features may lie near the image borders, where the kernel is applied less frequently, potentially leading to information loss. Padding addresses this issue by extending the input with artificial pixels along its borders, thereby preserving the original dimensions and reducing the risk of losing peripheral information.

Padding can be applied in two common ways:

- **Valid Padding ("No Padding")**: no padding is added to the input feature map, and the resulting output feature map is smaller than the input.
- **Same Padding**: zeros are added around the input feature map so that the spatial dimensions of the output match those of the input.

Figure 15 shows an example where a padding of 1 is applied to a 3×3 input.

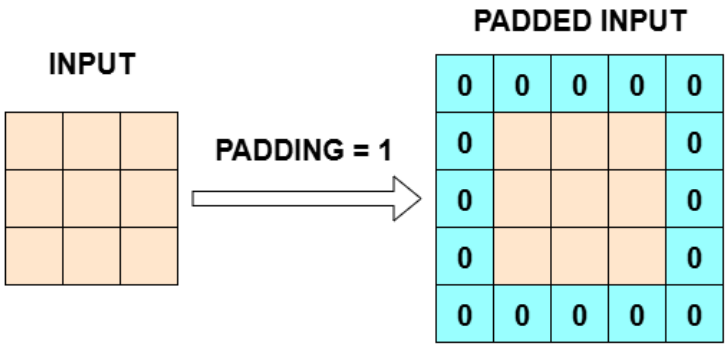


Figure 15: Padding effect on Convolution

4.2.3 Stride and Padding in the HLS Implementation

The combination of stride and padding, previously analyzed, allows balancing computational efficiency with the perceptual depth of the model. It is crucial to implement

these parameters, as they are present in all modern convolutional networks to achieve efficient performance and accurate results. In this thesis, stride and padding were explicitly integrated with the goal of minimizing internal resource utilization.

For instance, regarding padding, the idea of implementing it directly in the *input2conv* function, responsible for producing partitioned memory, was discarded. Storing the entire input together with the padded elements would have significantly increased BRAM usage. Instead, both stride and padding were implemented inside the *mem_conv2stream* function. This function retrieves data from partitioned memory and provides them to the convolution engine with the appropriate unrolling factor. Specifically, it delivers data packets of size $FW \times FH \times ICH\_PAR$, ensuring that at each clock cycle the convolution receives the inputs required for computation. By passing stride and padding as template parameters, the function fetches the correct data from memory, inserts the necessary padding, and streams the result to the convolution operator.

An excerpt of the implementation, available in the GitHub repository in Appendix 7.1, is provided below:

```
1  const int ih_real = s_oh * STRIDE + s_fh - PADDING;
2  const int iw_real = s_ow * STRIDE + s_fw - PADDING;
3  const bool valid = !(ih_real < 0 || ih_real >= IH || iw_real < 0 || iw_real >= IW);
4
5  const int s_mem_i_depth = (iw_real / FW_IN) * (ICH / ICH_PAR_IN) + (ih_real / FH_IN) *
       (ICH / ICH_PAR_IN) * WINDOW_IN + s_ich / ICH_PAR_IN;
6  const int s_mem_i = (iw_real % FW_IN) + (ih_real % FH_IN) * FW_IN + s_ich_par * FW_IN
       * FH_IN;
7  ap_uint<MEM_IN_T_BIT_WIDTH> read_data = 0;
8  if (valid) {
9      // valid: read from memory
10     read_data = memory_in[s_mem_i][s_mem_i_depth];
11 }
12 current_packet((packet_element_idx + 1) * MEM_IN_T_BIT_WIDTH - 1, packet_element_idx *
       MEM_IN_T_BIT_WIDTH) = read_data;
13 packet_element_idx++;
```

In the first lines, the **stride** is applied: *iw_real* and *ih_real* represent the reconstructed x and y coordinates of the input image. These coordinates are computed from the output indices (*s_oh*, *s_ow*), the filter indices (*s_fh*, *s_fw*), and the padding value.

In the following lines, **padding** and **memory indexing** are handled. First, the code checks whether the current position are not lies at the image border; in such a case, the appropriate memory element is fetched using the indices *s_mem_i_depth* and *s_mem_i*, taking into account that the partitioned memory is organized as a two-dimensional array. Otherwise, a zero is written, implementing the padding.

4.2.4 BIAS

In every convolutional operation, the kernel computes a weighted sum of the input pixels. In some cases, a constant term called the ***bias*** is added to this sum, acting as an offset before the activation function is applied. Formally, if a convolutional filter produces an output value y from a local subset of input values x and the corresponding kernel

weights w, the computation is given by:

$$y = \sum_{i=1}^{N} (x_i \cdot w_i) + b \tag{3}$$

In addition to shifting the activation function, biases enable convolutional filters to better adapt to the data by introducing a constant term that can activate or suppress a feature regardless of the input. As a result, they provide greater model flexibility, improve training stability, and contribute to faster convergence.

In this design, the bias values are **stored internally** in the FPGA BRAMs. This choice is motivated by the fact that the number of biases required for this layer is relatively small and does not lead to significant BRAM utilization. However, if resource usage were to become critical, the biases could instead be stored in external DDR memory and accessed through a Direct Memory Access (DMA) implemented in HLS, as is already done for activations, kernels, and outputs.

4.2.5 Template Functions for Convolution

The following section presents **C++ function template** that implements the convolution operator, which represents the core operator within a CNN. The template parameters of the function are first described, followed by an explanation of the input and output interfaces: the partitioned input feature maps, the kernels and biases, and the output feature maps produced by the convolution engine.

```
template<int ICH, int IW, int IH,
    int FW, int FH,
    int OCH, int OW, int OH,
    int ICH_PAR,
    int STRIDE,
    int WINDOW_IN,
    int ICH_PAR_OUT,
    int FW_OUT, int FH_OUT, int OCH_OUT,
    int WINDOW_OUT,
    int RELU>
void conv(hls::stream<conv_packet_t<FW, FH, ICH_PAR>> &conv_data_stream,
    filter_stream_t& filter_stream, //default ap_int<8>
    const ap_int<32> bias[OCH],
    memory_out_conv_t<FW_OUT, FH_OUT, ICH_PAR_OUT, OCH_OUT, WINDOW_OUT>& out_mem)
```

The template parameters represent:

- **ICH**: number of input channels in the tensor.
- **IW, IH**: width and height of the input tensor.
- **FW, FH**: width and height of the kernel.
- **OCH**: number of output channels.
- **OW, OH**: width and height of the output feature map.
- **ICH_PAR**: parallelism on the input channels (i.e., how many channels can be read simultaneously).

- **STRIDE**: applied stride value.
- **WINDOW_IN**: spatial parallelism on the input windows (i.e., how many windows can be processed in parallel).
- **ICH_PAR_OUT**: output channel parallelism, used to generate the partitioned output memory.
- **FW_OUT, FH_OUT**: width and height of the kernel of the subsequent convolution, used to dimension the partitioned output memory.
- **OCH_OUT**: number of output channels of the subsequent convolution.
- **WINDOW_OUT**: spatial parallelism on the output windows of the subsequent convolution, used to properly size the partitioned memory of the next stage.
- **RELU**: Control parameter that specifies whether the ReLU activation is applied internally to the convolution operator.

The following section focuses on the inputs and outputs of the convolution function.

The **first input argument** represents the input tensor **data packet** retrieved from the partitioned memory. This packet is dimensioned to allow the convolution engine to perform the required computations with the proper unrolling factor, indeed, its size is $FW \times FH \times ICH\_PAR$, identical to the vertical dimension of the partitioned memory, which contains the input sub-window and, depending on the value of ICH_PAR, selects the subset of channels involved in the computation.

This type is an **HLS FIFO stream** (`hls::stream`, from the HLS library) is employed, ensuring pipelined communication: while one data packet is being processed, the next can already enter the stream. Notably, a packet is not a single scalar value but a group of elements defined as follows: $ap\_uint < FW \times FH \times ICH\_PAR \times 8 >$. Where, each packet contains a block of unsigned 8-bit integers (uint_8). The definition is fully parameterized so that modifying the data type or size only requires changing this single line of code, thereby providing flexibility and scalability.

The **second parameter** of the function is an FIFO stream of 8-bit elements representing the **kernel coefficients**. Specifically, *filter_stream_t* is defined as follows $hls :: stream < ap\_int < 8 >$. Here, the data type is an 8-bit signed integer, since kernel weights can take negative values.

The **third parameter** is a vector that contains the **bias values**. In the implemented layer, the biases were represented as 32-bit integers, as specified by the ONNX model. These values are incorporated into the convolution according to Equation 3.

The **last parameter** of the function corresponds to the **convolution output**. It is implemented as a partitioned memory, as described in Section 3.3.3. The data type *memory_out_conv_t* is a two-dimensional memory whose dimensions are identical to those analyzed previously, but they refer to the next convolutional stage. If no subsequent stage is present, simply set the parameters `FW_OUT` and `FH_OUT` to 1 to obtain a single (1-D) vector. The only difference is in the data width: after convolution, results are stored in 32-bit integers to avoid overflow or saturation during accumulation.

4.2.6 Hardware Architecture

This section illustrates, from a hardware perspective, what is expressed at a high level in HLS. As previously discussed, the inputs to the convolution function are delivered via an HLS stream packet. A dedicated routine fetches data from the partitioned memory, applies padding and stride when required, and provides exactly the data needed for a single clock tick of computation. Consider, for example, a $\mathbf{3 \times 3}$ **kernel** and $\boldsymbol{ICH\_PAR = 2}$, which determines the degree of channel-level parallelism. The vertical dimension of the memory is therefore $\mathbf{3 \times 3 \times 2 = 18}$, meaning that the data required each cycle are distributed across 18 distinct memory banks. Assume stride and padding euqal to 1. Figure 16 shows how the data stream is populated and forwarded to the convolution.

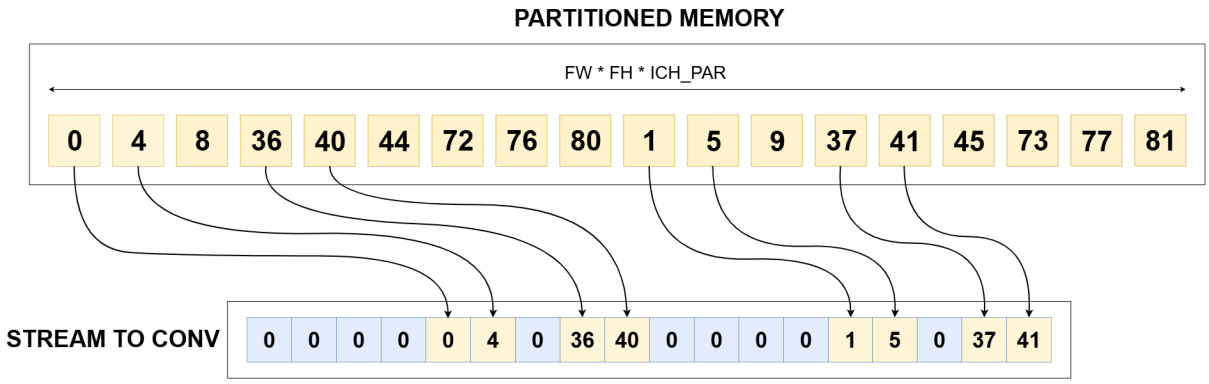


Figure 16: Hardware Architecture of Function before the Convolution

It can be observed that the partitioned memory comprises 18 distinct banks; for brevity, only the first entries are shown. The subsequent block depicts the data packet sent to the convolution: the blue zeros correspond to padding, while the yellow values are read directly from the partitioned memory.

The convolution function is then examined from a hardware standpoint. In practice, convolution performs a large number of **MAC** operations: input samples are multiplied by the corresponding weights, and the products are accumulated across all channels. Hardware-wise, this entails allocating multipliers and accumulators (with the degree of parallelism set by ICH_PAR), and reducing the partial sums to produce each output sample.

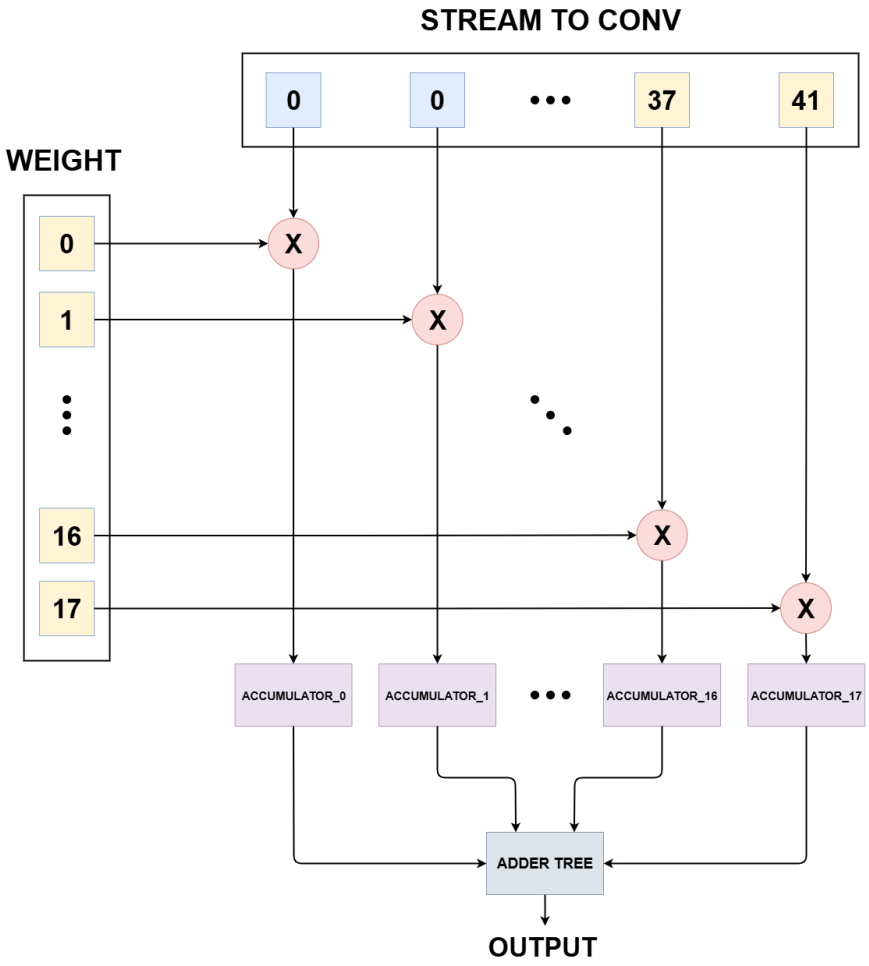


Figure 17: Hardware Architecture of Convolution

In Figure 17, the two inputs are shown: the incoming data packet and the weights used for computation. As indicated, there are **18 accumulators**, reflecting that all operations run in parallel according to the chosen ICH_PAR value. Each multiplier reads its operand from a different memory bank, hence the emphasis in this work on partitioned-memory design to ensure data reside on distinct banks. This organization enables parallelization, thus consuming more hardware resources, while achieving a higher throughput.

The advantage of using HLS lies in the fact that, unlike RTL design, where each hardware block composing the datapath must be instantiated manually, here the algorithm can be described directly in C++, such as: $sum+ = local\_mem[...][...] * filter\_mem[...][...]$;
The focus therefore shifts to properly organizing the loops and ensuring the absence of data dependencies. By applying pragmas, such as those introduced in the previous chapter (e.g., pipelining and unrolling), the Vitis HLS tool can accurately translate the C++ description into RTL, instantiating the correct number of multipliers and accumulators. Otherwise, the instruction `sum += a * b` would be synthesized with only a single multiplier and accumulator, forcing the products to be executed sequentially and thereby drastically reducing throughput.

4.3 ReLU

The **Rectified Linear Unit (ReLU)** is one of the most widely used activation functions in neural networks. An activation function determines whether a neuron should be active or not, introducing non-linearity into artificial neural networks. This non-linearity is crucial for modeling complex relationships within data and significantly improves the representational power of the network. ReLU is formally defined as:

$$\boldsymbol{ReLU(x) = max(0, x)} \tag{4}$$

In essence, this function returns the input itself if it is positive, and zero otherwise. In addition to being extremely efficient to compute, ReLU alleviates the vanishing gradient problem and, by producing many zero outputs, also reduces the computational load in subsequent operations.

4.3.1 ReLU in the HLS Implementation

In most networks, the ReLU activation is placed immediately after the convolution to **break the linearity** of consecutive convolutions; otherwise, stacking multiple convolutions without activation would reduce the representational capacity of the model. In ResNet-18, however, as observed in the ONNX model, ReLU is not always present after every convolution. For this reason, a template parameter was added to the convolution function to indicate whether ReLU should be applied or not.

From an implementation perspective, ReLU is computationally straightforward and has been integrated directly into the convolution function as follows:

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

Before writing the result to the partitioned output memory, the value of the template parameter `RELU` is checked. If it is enabled, the accumulated sum is clamped to zero when negative; otherwise, the value is left unchanged.

Integrating ReLU within the convolution avoids a separate processing stage and reduces memory transactions between the convolution and activation functions. Furthermore, the implementation relies on conditional statements (`if`), which the HLS tool maps to multiplexers and a comparator. As a result, the iteration interval of the convolution function remains equal to one and is not affected by the added logic.

4.4 Pooling Layer

The **pooling layer**, ubiquitous in many CNNs, reduces the spatial dimensions of the input feature maps while preserving the most salient information. By shrinking the feature

maps, it lowers the computational complexity of the network by reducing the number of operations. Pooling slides a two-dimensional window over each channel of a feature map and aggregates the features within the windowed region. The variants used in ResNet-18 and LeNet are:

- **Max Pooling**: selects the maximum element within the windowed region of the feature map, yielding an output that retains the most prominent features from the previous layer.
- **Average Pooling**: computes the mean of the elements within the windowed region, producing the average feature value over that area.
- **Global Average Pooling**: takes each channel of the feature map and averages over all spatial positions $H \times W$, resulting in a single scalar per channel.

Figure 18 provides a graphical depiction of these three cases.

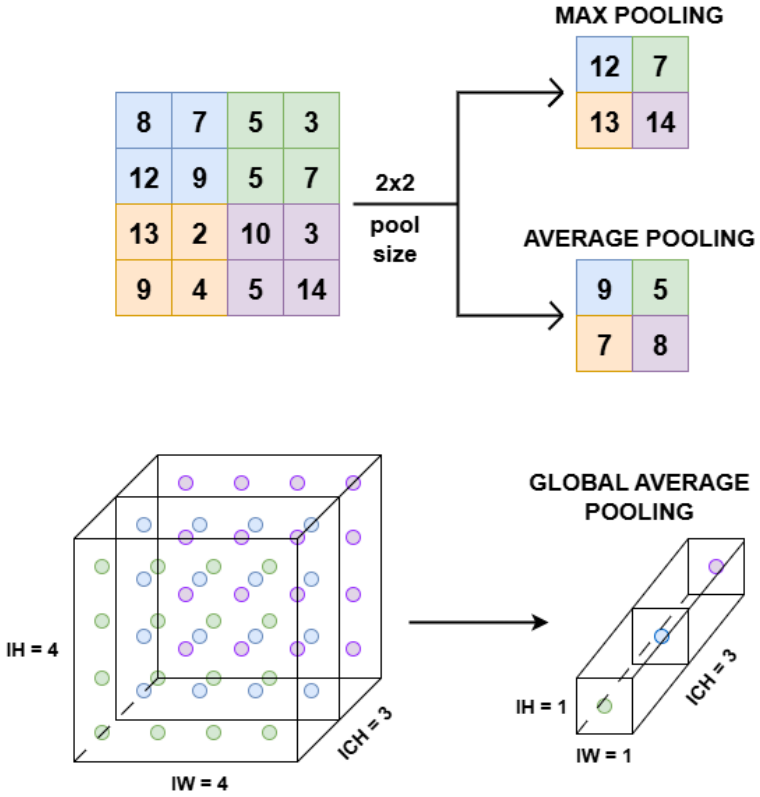


Figure 18: Pooling methods

4.5 Quantization

Nowadays, with the rapid advancement of artificial intelligence, neural networks are becoming increasingly complex and demand substantial computational resources for both training and deployment. **Quantization** is the process of reducing the precision of a digital signal, typically from a higher-precision format to a lower-precision format. This technique is widely used in various fields, including signal processing, data compression

and machine learning. [20]

In the context of AI, quantization aims to accelerate models by converting their weights and activations from high-precision **floating-point formats**, such as 32-bit floats (FP32), to low-precision data types, such as **8-bit integers** (INT8). Reducing the data representation inevitably decreases precision, introducing what is referred to as quantization error. Nevertheless, quantization significantly reduces model size and inference latency, enabling deployment on hardware with limited resources, such as FPGAs.Performing quantization from FP32 to INT8 is nontrivial. Only 256 distinct values can be represented in INT8, whereas FP32 can cover a much broader dynamic range. From a mathematical perspective, and considering uniform quantization, the mapping can be expressed as:

$$Q(r) = Int\left(\frac{r}{S}\right) - Z \tag{5}$$

where Q is the quantization operator, r is a real-valued input (activation or weight), S is a real-valued scaling factor, and Z is an integer zero-point. This function maps real-valued numbers to their corresponding integer representations.

In addition to improving inference speed, quantization also achieves a significant reduction in memory footprint. For instance, converting from FP32 to INT8 leads to an approximate 4× reduction in storage requirements. From a computational perspective, low-precision arithmetic can be executed with simpler hardware units (e.g., ALUs and multipliers), resulting in a considerable decrease in energy consumption, an essential advantage for embedded applications.

As discussed earlier, quantization entails divisions that often involve floating-point numbers. At the hardware level, this can increase the utilization of DSP slices, which are available in limited quantity on FPGAs. For this reason, quantization is frequently implemented using **shift operations**: scales and weights are chosen as powers of two, so multiplications and divisions reduce to shifts that do not consume DSP resources. This approach may introduce a modest loss in accuracy; however, well-established techniques such as QAT and bias correction can recover much of the lost precision.

Given the scope and complexity of this thesis, the quantization function was developed by another student and integrated into the design. The same student is also implementing the pooling layer so that it can be incorporated into the experiments conducted thus far.

4.6 Structure of the Top_Wrapper

To proceed with synthesis, Vitis HLS requires specifying a top-level function. In this work, it is represented by the function defined in **top_wrapper.cpp**, which invokes the various routines needed to implement the neural network. The interface is presented below, detailing precisely what the function takes as input and what it produces as output, in order to implement one layer of ResNet-18, including convolution, ReLU and quantization. The input/output values are read from and written to a DDR memory external to the FPGA. Consequently, in Vitis HLS we instantiated as many DMA engines as there

are parameters of the function.

```
void top_wrapper(hls::stream<mem_in_t>  &memory_in_stream,
                 filter_stream_t        &filter_val_stream,
                 hls::stream<mem_out_t> &memory_out_stream)
```

For the implementation of a single layer, the top function exposes the following three parameters:

- **memory_in_stream**: a FIFO stream where *mem_in_t* denotes the data type, corresponding to $ap\_uint < 8 >$.
- **filter_val_stream**: a FIFO stream where $filter\_stream\_t$ is an $ap\_int < 8 >$, identical to the input data type of the convolution.
- **memory_out_stream**: a FIFO stream where *mem_out_t* denotes the data type, corresponding to $ap\_axiu < 8, 0, 0, 0 >$. This is an HLS structure for a 1-byte AXI4-Stream, a point-to-point protocol for transferring data without addresses. In addition to the payload (TDATA), which in this case is an $ap\_uint < 8 >$, it includes auxiliary fields such as TKEEP and TSTRB, each one bit wide and asserted when the byte is valid. It also includes TLAST, a bit asserted on the last byte to mark the end of the packet. In practice, this format is used when one wants HLS to automatically generate the AXI4-Stream signals and handshaking.

4.6.1 Algorithm Block Diagram

The previously analyzed top level encapsulates the functions required to implement the selected layer. Figure 19 depicts, on the left, an ONNX excerpt of a ResNet-18 model: quantized data weights, biases, and inputs, feed the convolution, followed by operations such as ReLU and quantization. On the right, the workflow executed within `top_wrapper` is shown. In particular, it highlights the principal functions that constitute ResNet-18 (e.g., convolution and quantization), as reflected in the ONNX graph. Several auxiliary routines are also included to organize the data and route it correctly to downstream stages.

In the conducted experiments, in addition to the layer under consideration, the subsequent convolution (shown in white) was also included. This further increased the design complexity and enabled a more detailed assessment of resource utilization and performance.

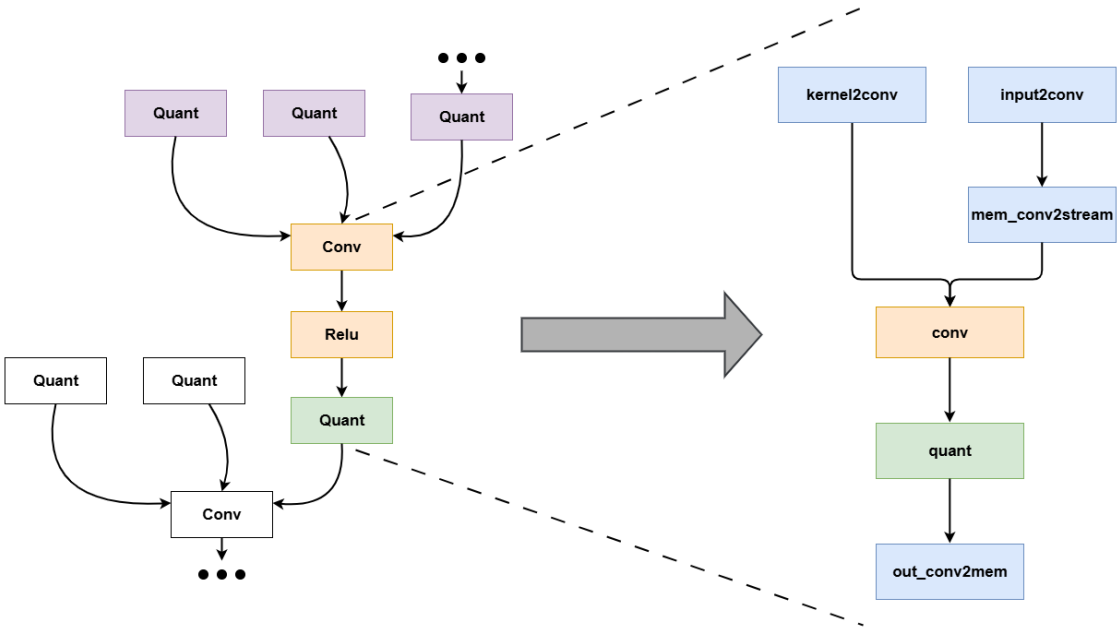


Figure 19: Block diagram of the top_wrapper function flow

In particular:

- **kernel2conv**: this function acts as a buffer between the data arriving from external memory. Specifically, the data are the filter coefficients, which are read from an input stream and forwarded to an output stream at one element per clock cycle. The output stream then feeds the convolution.
- **input2conv**: as introduced in Section 3.4, this function retrieves the input tensor via a stream and places it into a partitioned on-chip memory within the FPGA.
- **mem_conv2stream**: this function reads data from the partitioned memory, adapts them by adding padding and stride, if necessary, and finally transmits a data packet via stream to the convolution. The packet is shaped according to the chosen unrolling factor, in order to parallelize the computations and increase throughput.
- **conv**: as described in Section 4.2.5, this function performs the convolution between the input packet and the kernel. The loops are structured to exploit parallelism according to the input parameters (ICH_PAR).
- **quant**: this function reads the memory produced by the convolution and quantizes the data, converting them from 32-bit integers to 8-bit integers. In this way, they are ready to be consumed by another convolutional layer or to constitute the final output of the network.
- **out_conv2mem**: this function takes the partitioned memory produced by quantization (which remains a two-dimensional partitioned memory) and serializes it into an AXI4-Stream. It essentially writes to external DDR a packet that, in addition to the data payload, also carries fields such as TKEEP, which indicates which bytes are valid, and asserts TLAST on the last element.

4.7 Tcl Script for Project in Vitis HLS

To accelerate simulation and debugging, Vitis HLS was not used via the Graphical User Interface (GUI); instead, the flow was driven from the command line by invoking a Tool Command Language (Tcl) script that explicitly encodes the same commands one would issue in the GUI. An example is shown below: it creates a new project (one_layer), sets the top-level function, and adds the source files required for synthesis and simulation. It also specifies the target FPGA device and the clock period. Finally, it launches, in order, C/C++ simulation, synthesis, RTL co-simulation, and exports the resulting IP core for the subsequent steps in Vivado.

Listing 3: Tcl script for launching the Vitis HLS project

```
# Create a project
set impl_sel "solution_0"
# Open the new project
open_project one_layer
open_solution ${impl_sel}
# Set the top-level function
set_top top_wrapper
# Add design files for synthesis (remove -tb here)
add_files /home/lorenzor/circtvm1/workspace/convolution/top_wrapper.cpp
add_files /home/lorenzor/circtvm1/workspace/convolution/top_wrapper.h
add_files /home/lorenzor/circtvm1/workspace/convolution/conv.h
add_files /home/lorenzor/circtvm1/workspace/convolution/parameter.h
# Add testbench file (only for simulation, keep -tb here)
add_files -tb /home/lorenzor/circtvm1/workspace/convolution/conv_tb_lenet.cpp
# Define technology and clock rate
set_part {xczu9eg-ffvb1156-2-e}
create_clock -period 5
# Run simulation (csim)
csim_design
# Run synthesis (csynth)
csynth_design
# Run simulation (co-sim) + export waveform
cosim_design -trace_level all
# Export IP
export_design -flow syn
exit
```

5 FPGA Design and Synthesis with Vivado

This chapter presents the **Vivado-based synthesis flow** and the FPGA implementation of the project. Vitis HLS enables C/C++ functional simulation (csim), high-level synthesis to RTL, and RTL co-simulation (cosim). It also provides resource estimates for the selected FPGA target. The final step is implementation on the specific hardware platform, in this case an FPGA, using Vivado, which performs logic synthesis and Place & Route on the target device and, finally, generates the bitstream, that is, the configuration file used to program the FPGA.

Subsequently, the Vivado **Block Design** is introduced. It offers a graphical representation of the hardware system and facilitates the connection and configuration of IP cores with the custom HDL module. To interface with the FPGA, a Python script based on AMD's open-source **PYNQ** framework was developed, enabling bitstream programming and data exchange with the programmable logic.

The specific hardware platform used in this thesis is also presented, onto which the bitstream was loaded and the results were verified. In particular, the AMD **ZCU102** board available in the laboratory was employed. This platform integrates ARM CPUs (PS) and PL within the same SoC, enabling control to be executed on the PS while hardware accelerators run on the PL. Finally, we describe how the **dataflow** among the functions in the top-level function was orchestrated using a ping–pong buffer, and the resulting waveforms that validate the implementation were presented from Vivado.

5.1 Vivado

Vivado, introduced in 2012 by **Xilinx**, is a software suite for the synthesis and analysis of HDL designs and for hardware development on FPGAs and SoCs. The design flow starts from a VHDL/Verilog description or pre-packaged IP and culminates in the generation of the bitstream, the configuration file that programs the device. Vivado also enables the specification of clock and timing constraints and performs logic synthesis and implementation. The tool automatically manages run data, enabling repeated build attempts with different RTL source revisions, target devices, synthesis and implementation options, and physical or timing constraints.

With the Vivado Design Suite, you can accelerate design implementation with place and route tools that analytically optimize for multiple and concurrent design metrics, such as timing, congestion, total wire length, utilization and power. The Vivado Design Suite provides you with design analysis capabilities at each design stage. This allows for design and tool setting modifications earlier in the design processes where they have less overall schedule impact, thus reducing design iterations and accelerating productivity. [21]

After generating the RTL with Vitis HLS, the design is exported to create an IP core that can be integrated into the project. The first step in Vivado is to create the Block

Design, where the exported IP is connected to other pre-existing, configurable IP, such as the Zynq Processing System, AXI interconnect/controllers, and peripherals. At this point, synthesis can proceed. Vivado allows you to define design constraints in Xilinx Design Constraintss (XDCs) files. Both physical constraints (e.g., pin assignments and the placement/floorplanning of BRAMs, LUTs, and flip-flops) and timing constraints (clock definitions and target operating frequencies) can be specified. The tool also lets you select from built, in implementation strategies or define custom ones to optimize for area, performance, power, or runtime.

After synthesis, the flow proceeds to implementation, which includes all steps required to place and route the netlist onto the device resources while honoring the project's logical, physical, and timing constraints. During implementation, Vivado performs physical optimizations such as retiming, register replication, and buffer tree insertion. For this reason, the resource figures reported by Vitis HLS should be considered estimates: utilization, especially of LUTs and flip-flops, often changes after physical optimization. In many cases the resource count decreases, but it may also increase, for example when register replication or additional buffering is required to meet the target frequency.

Finally, Vivado generates the bitstream, the configuration file used to program the FPGA. The bitstream specifies LUT functions, internal routing, BRAM/DSP configuration, clocking, and I/O interfaces. The tool produces a ".bit" file that is then loaded onto the FPGA.

Figure 20 presents the complete workflow carried out in this work, with particular emphasis on the steps executed in Vivado: RTL integration, synthesis, implementation and finally programming and debugging on an hardware device.

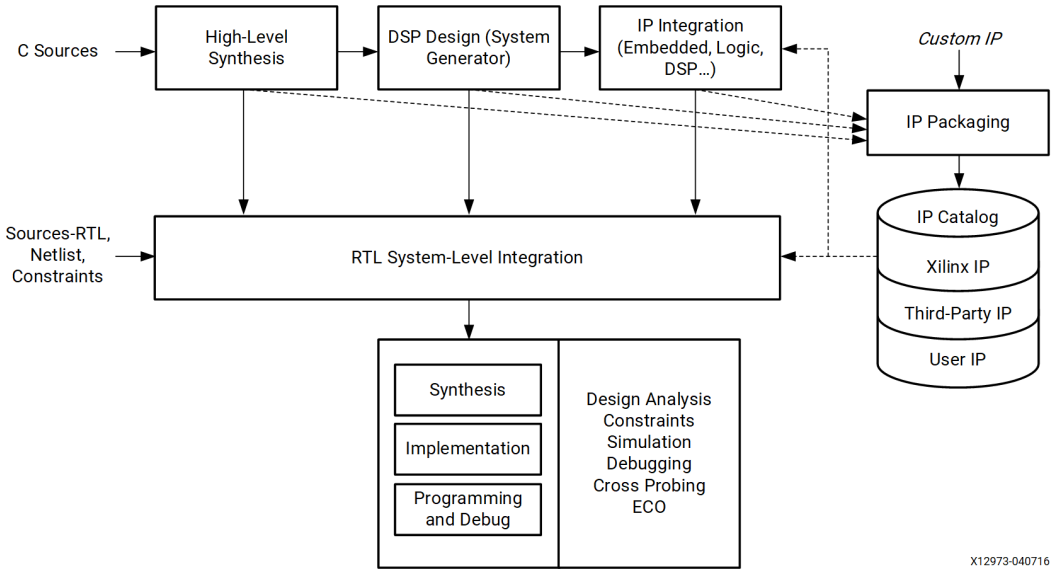


Figure 20: Vivado Design Suite High-Level Design Flow [22]

5.2 Vivado Block Design

As discussed earlier, the first step in Vivado is to implement the **Block Design**, in which the IP generated with Vitis HLS is connected to other IP blocks to form a com-

plete hardware system. Vivado recognizes standard interfaces such as AXI and supports automatic connections for clock and reset, as well as the connections between the various IPs. Figure 21 shows the Block Design developed to implement a ResNet-18 layer (that include convolution, ReLU and quantization); and then the role of each individual block is detailed.

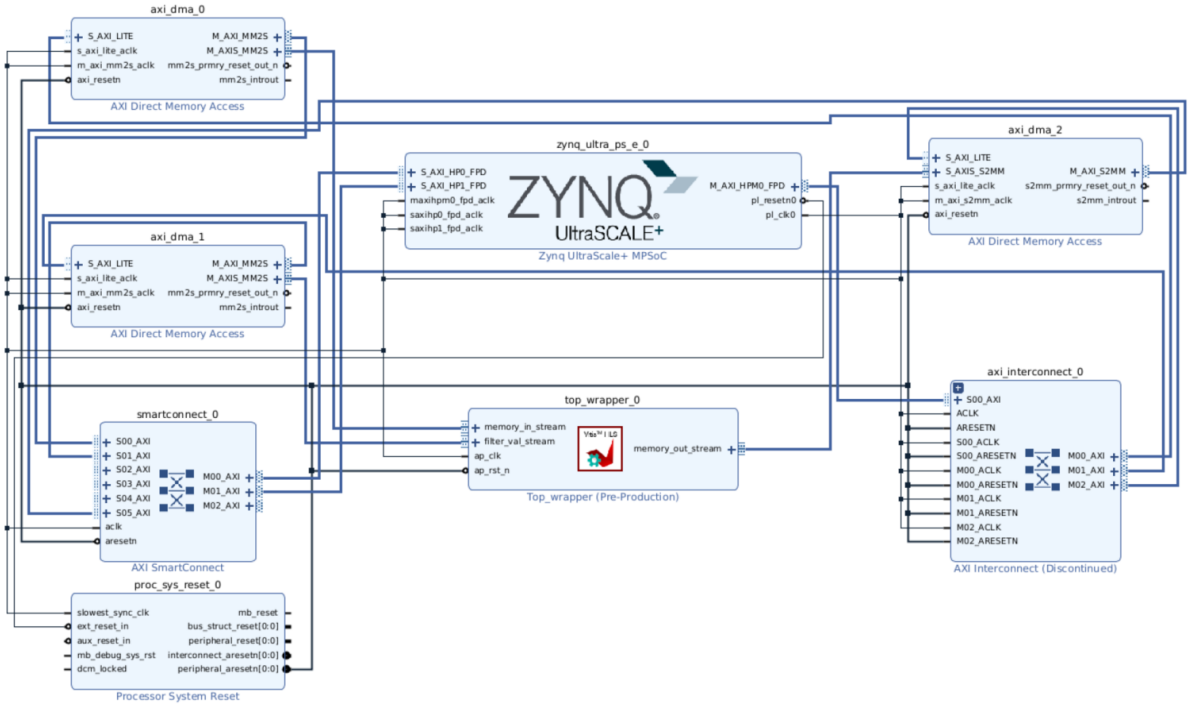


Figure 21: Vivado Block Design

In particular, note the following:

- **Top_wrapper**: located at the center of the Block Design, this is the project IP exported from Vitis HLS. It corresponds to the accelerator developed in C++ and converted to RTL by the tool. Its interface exposes the input signals described in Section 4.6, with the addition of the clock and reset signals.

- **Zynq UltraScale+ MPSoC**: constitutes the FPGA PS; it provides clock and reset to the PL. It exposes an *M_AXI_HPM0_FPD* master port (PS → PL), used by software to configure peripherals over AXI-Lite, and *S_AXI_HP*_FPD* slave ports through which PL masters (the various DMAs) can read from or write to the PS DDR.

- **AXI Direct Memory Access (axi_dma_0 and axi_dma_1)**: both supply the top-level inputs activations and kernels, respectively, via their *M_AXIS_MM2S* stream interfaces. The *M_AXI_MM2S* ports fetch buffers from DDR, while the *S_AXI_LITE* interfaces expose control registers (address, length) programmed by the PS. They are configured in read-only (MM2S) mode, since they read from DDR and drive the input streams. Figure 22 illustrates the configuration of the DMA for the activations:

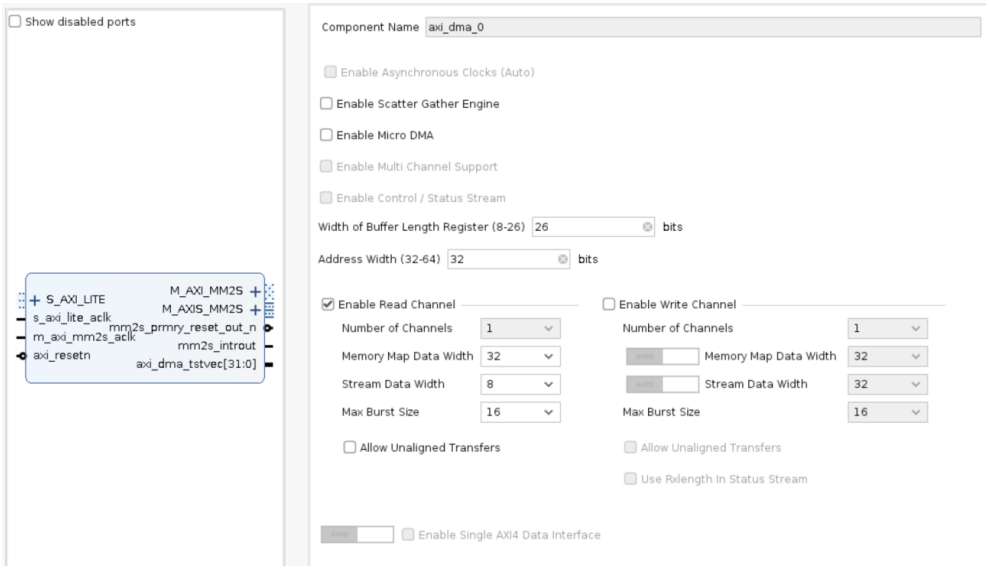


Figure 22: AXI Direct Memory Access input configuration

It can be observed that, in addition to enabling only the read channel, the Width of Buffer Length Register was set to 26; consequently, the maximum length per single transfer is $2^{26} - 1 = 67.108.863$ (≈ 64 MiB). This is sufficient to load the entire input tensor, even when the batch size is greater than one. The Memory Map Data Width is set to 32, indicating 32-bit words on the AXI memory-mapped master. The Stream Data Width is set to 8 bits, since the input data are 8-bit. The Max Burst Size is 16 beats, in this way the DMA will not issue bursts longer than 16 beats on the memory side. Finally, the Enable Scatter Gather Engine option is disabled, meaning the core operates in Simple Mode: the DMA reads blocks from memory and streams them out on an 8-bit AXI4-Stream interface. Stream continuity is managed in software using a ping–pong buffer (see Section 5.5.1), preventing the stream from stalling unless the input data are exhausted.

- **AXI Direct Memory Access (axi_dma_2)**: this DMA, placed on the right of the Block Design, receives the results from the top. It is configured in write-only (S2MM) mode, it accepts a stream on *S_AXIS_S2MM* (the `memory_out_stream` from the top) and writes to memory via *M_AXI_S2MM*. The *S_AXI_LITE* interface manages the control registers. Figure 23 illustrates the configuration of the DMA for the output:

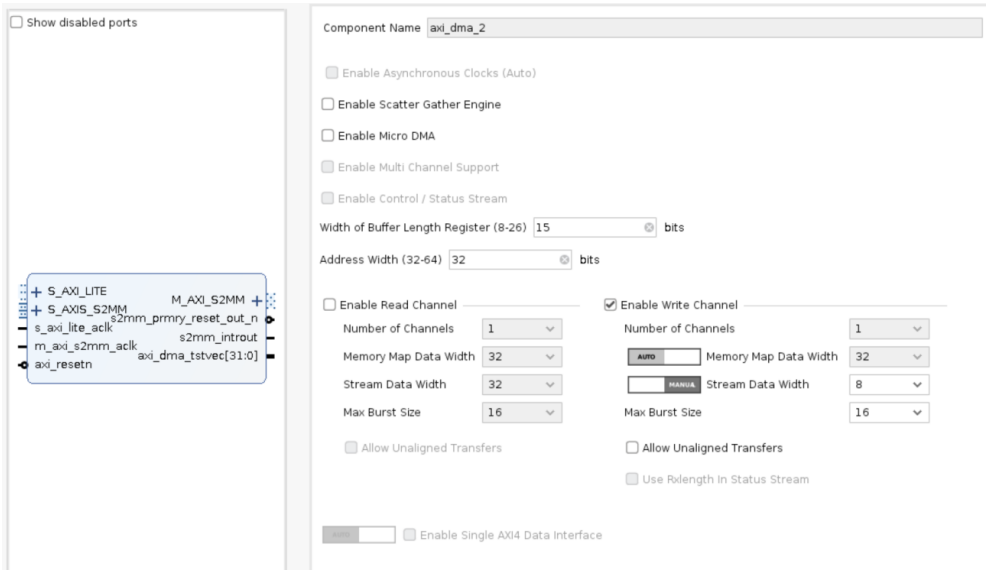


Figure 23: AXI Direct Memory Access output configuration

In this case, the key difference is that only the write channel (S2MM) is enabled. Scatter Gather is disabled, so the DMA operates in Simple Mode. The Width of Buffer Length Register is set to 15. For the implemented ResNet-18 layer, the output size is $512 \times 7 \times 7 = 25\,088$ elements; with 8-bit outputs (1 byte/element), the entire output feature map fits within a single transfer, ensuring correct end-to-end writing.

- **AXI SmartConnect**: manages the interconnections for the DMAs' memory-mapped channels. It is termed "Smart" because it automatically inserts data-width/protocol conversions (and CDC bridges when enabled) and arbitrates transfers among multiple IP.
- **AXI Interconnect**: handles the control-path interconnections; it receives AXI-Lite transactions from the PS and forwards them to the DMAs' *S_AXI_LITE* ports, thereby programming buffer addresses and transfer lengths.
- **Processor System Reset**: receives a reset input and the PL clock, and generates reset signals synchronized to the PL clock domain. This improves robustness by asserting releases only when the clock is stable.

5.3 PYNQ Overlay

PYNQ is an open-source AMD project that streamlines application development on Xilinx/AMD Zynq and Zynq UltraScale+ SoCs by leveraging both the PL and the PS. It enables embedded designers to exploit Adaptive Computing platforms without resorting to ASIC, oriented design tools for programmable logic development. In practice, PYNQ provides facilities to interface with the FPGA, create the drivers required to supply input data, and load the bitstream that configures the device. PYNQ also supports Python

for programming both the embedded processors and the overlays, reusable, configurable hardware libraries implemented in the PL.

After creating the Block Design, running synthesis and implementation, and generating the bitstream, the FPGA deployment was carried out via a Python script using PYNQ. The board is first provisioned with the **.bit** file (Vivado bitstream) and the corresponding **.hwh** file, which provides an object model of the hardware, including IP hierarchies, AXI addresses, and interconnections. Selected excerpts of the Python script (*inference.py*) are presented and explained below.

As mentioned, PYNQ supports Python programming and allows the use of libraries such as Overlay (to load and manage an overlay: .bit + .hwh), allocate (to allocate memory buffers), Clocks (to configure PS clock frequencies), and PL (to query the state of the Programmable Logic).

```
import pynq
from pynq import Overlay
from pynq import allocate
from pynq import Clocks
from pynq import PL
```

Subsequently, the programmable logic is reset to avoid conflicts with previously loaded overlays. The batch size, the number of images to be processed, is then configured. Finally, the bitstream and the associated .hwh file are loaded from the overlay directory.

```
PL.reset() #resetto la PL per evitare conflitti con overlay precedenti

NR_IMG = 1 #numero di immagini da processare

#caricamento overlay (bitstream + hwh) FPGA
print("Loading overlay", flush=True)
BITFILE = '/root/lorenzor/overlay/design_1_wrapper.bit'
overlay = Overlay(BITFILE)
print("Loaded overlay", flush=True)
```

After loading the bitstream and the .hwh file, the **DMA** peripherals are mapped. Care must be taken to ensure that the instance names match those used in the Vivado Block Design. Three DMAs are employed: two inputs, handling activations and kernels, respectively, and a third for the output.

```
# Mappo le periferiche DMA
dma_input = overlay.axi_dma_0
dma_kernel1 = overlay.axi_dma_1
dma_output = overlay.axi_dma_2
```

In the following code, the dimensions of the input tensor, the kernel, and the output are configured, and the corresponding buffers are allocated. Note that, for both input and output, two types of buffer are created: one that spans the entire input/output tensor and another that holds only the single image processed at a time. Naturally, if only one image is to be processed, these two buffers coincide.

```
W, H, C = 7, 7, 512  # Dimensioni delle immagini
PIXELS_PER_IMAGE = W * H * C
N_PIXELS = PIXELS_PER_IMAGE * NR_IMG  # Numero totale di pixel per batch
input_buffer = allocate(shape=(N_PIXELS,), dtype=np.uint8)  # Buffer per input
    immagini
single_image_buffer = allocate(shape=(PIXELS_PER_IMAGE,), dtype=np.uint8) # Buffer per
     una singola immagine
```

```
6
7  # FILTER 1 buffer: shape (OCH, ICH, KH, KW)
8  f1_size = 512 * 512 * 3 * 3
9  f1_buf = allocate(shape=(f1_size,), dtype=np.int8)
10
11 out_size_per_image = 512 * 7 * 7
12 out_size = out_size_per_image * NR_IMG
13 single_output_buffer = allocate(shape=(out_size_per_image,), dtype=np.uint8)  # Buffer
       per un singolo output
14 out_buf = allocate(shape=(out_size,), dtype=np.uint8)
```

After defining and populating the input and kernel buffers, the procedure proceeds with DMA transfers and output collection. A loop iterates over the feature maps to be processed, and the receive-side DMA for the output is started. Using the *allocate* library, the methods `.sendchannel` (to transmit data via a DMA) and `.recvchannel` (to receive data via a DMA) are employed.

In the case of multiple images, the portion of the batched input buffer corresponding to the image to be processed is extracted and transmitted through the input DMA. The kernels are then sent, and the system waits for the output to be received on the output DMA. Finally, the collected outputs are concatenated into an array so they can be compared against the ONNX model outputs to verify correctness.

```
1  for i in range(NR_IMG):
2      print(f"\n--- Processing Image {i+1}/{NR_IMG} ---", flush=True)
3
4      # Avvio il DMA in ricezione per l'output
5      print("Starting DMA for output.", flush=True)
6      dma_output.recvchannel.transfer(single_output_buffer)
7      print("DMA for output started.", flush=True)
8
9      start_idx = i * PIXELS_PER_IMAGE
10     end_idx = (i + 1) * PIXELS_PER_IMAGE
11     single_image_buffer[:] = input_buffer[start_idx:end_idx]
12     # Invio dati input al DMA
13     dma_input.sendchannel.transfer(single_image_buffer)
14     # Aspetta che i dati siano stati inviati
15     dma_input.sendchannel.wait()
16     print("Input data sent.", flush=True)
17
18     # Invio i filtri ai DMA dei kernel
19     dma_kernel1.sendchannel.transfer(f1_buf)
20     print("Filter data sent to DMA 1.", flush=True)
21     # Aspetta che i filtri siano stati inviati
22     dma_kernel1.sendchannel.wait()
23     print("Filter data sent to both DMAs.", flush=True)
24
25     # Avvio il primo kernel
26     dma_output.recvchannel.wait()
27     print("Output data received.", flush=True)
28     all_outputs.append(np.copy(single_output_buffer))  # Copia il buffer di output per
       ogni immagine
29
30 final_result_array = np.array(all_outputs)
```

This Python script is transferred to the FPGA's PS together with the bitstream and the .hwh file, and is then executed with the following command **python inference.py**.

5.4 FPGA Device Used

The FPGA selected for this thesis is AMD's **ZCU102**, widely used in domains such as surveillance, advanced driver-assistance systems (ADAS), computer vision, augmented reality (AR), unmanned aerial vehicles, and medical imaging. In the context of neural-network deployment, FPGAs are achieving substantial success due to their flexibility and ability to be tailored to specific tasks. They enable low latency operation with low power consumption, features that are especially important for embedded applications.

Figure 24 shows the FPGA platform used in this work; its availability in the laboratory enabled the hardware implementation of the proposed design.

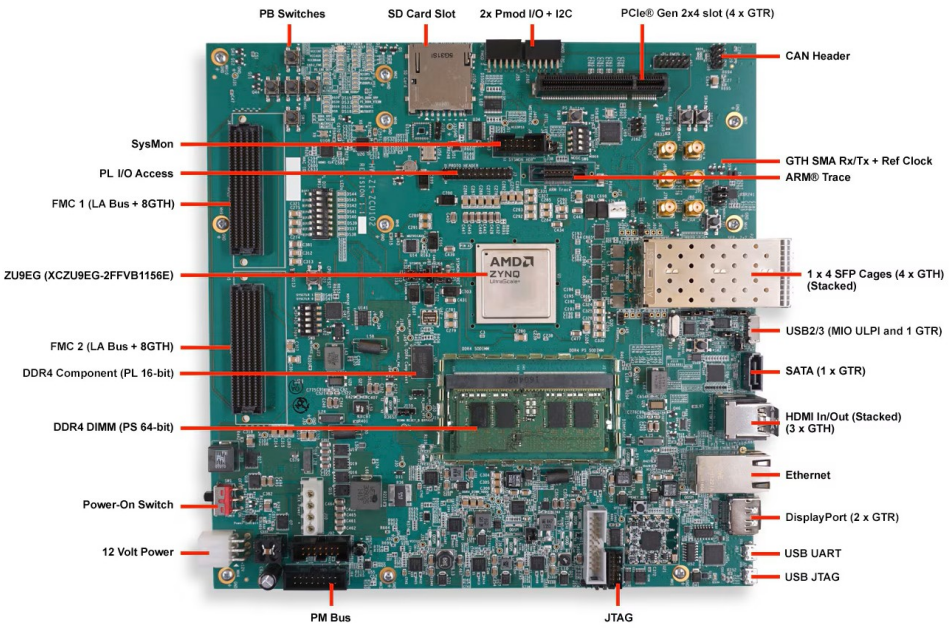


Figure 24: ZCU102 board [23]

At the center of the board is the Zynq UltraScale+ MPSoC, which integrates the Processing System (PS) and the Programmable Logic (PL). Table 1 summarizes the resources available on the ZCU102. Having a clear view of the available resources is crucial because they are inherently limited; this focuses attention on the HLS strategies, such as the pragmas discussed in previous chapters, that boost performance, often at the cost of additional hardware.

Table 1: Device resources and I/O limits [23]

System logic cells (K)	600
DSP units	2520
Memory (Mb)	32.1
Max I/O pins	328

5.5 Dataflow

In this section, a topic that is central to HLS, **dataflow**, is addressed. This optimization enables the various tasks (functions) within an algorithm to execute concurrently. Rather than waiting for a sequential turn, each task starts as soon as its input data are available and its output can accept additional data. Figure 26 shows an example of a function that contains three subfunctions.

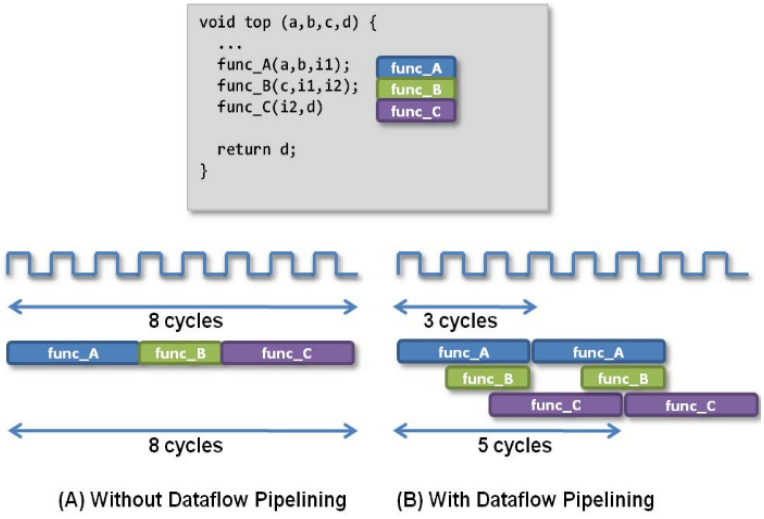


Figure 25: Ping-Pong buffer [24]

On the left, the data path without dataflow is shown: functions A, B, and C execute strictly in sequence for a total of 8 clock cycles. On the right, the same computation is organized with dataflow. In this case, the functions do not start one after another; each begins as soon as its input data are available and its output can accept new data. The first three operations complete in 5 clock cycles, compared with 8 in the sequential case. It follows that dataflow reduces latency, while one stage computes, the others can read or write data, and increases throughput: after the pipeline warm-up, a new result can be produced every clock cycle (assuming loop initiation interval II=1). Overall, this yields a substantial performance improvement.

A crucial note is not to conflate pipelining with dataflow. Pipelining accelerates a loop or a single function by overlapping successive iterations of the same computation, whereas dataflow runs different blocks concurrently, connected via streams or FIFOs.

In HLS, the *<#pragma HLS dataflow>* directive is used to implement dataflow execution across multiple tasks (functions), allowing them to run concurrently as soon as their input data are available. "When the DATAFLOW pragma is specified, the HLS tool analyzes the dataflow between sequential functions or loops and creates channels (based on ping pong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL. If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, the HLS tool attempts to minimize the initiation interval and start

operation as soon as data is available." [25]

5.5.1 Ping-Pong Buffer

Within the HLS code, dataflow is realized by connecting tasks through **FIFOs** implemented as **hls::stream** channels. This decouples the tasks: a producer writes data and stalls only when the FIFO is full, while a consumer reads when data are available and otherwise waits. If the FIFOs are undersized, a deadlock can occur due to a cycle of dependencies, specifically, when processes inside a DATAFLOW region contend for the same channels in a way that prevents further reads or writes, causing all involved processes to block. Deadlocks can be detected during C/RTL co-simulation, where the tool reports an error in the log. They can also manifest on hardware: for example, the system may hang when attempting to receive data on the output DMA channel because the expected stream never arrives.

In addition to using FIFO-based hls::stream channels, which enable the compiler to realize dataflow between functions, a **ping–pong** buffering scheme (also known as double buffering) was adopted. The approach alternates between two on-chip buffers: while one buffer (ping) is being filled from memory (e.g., via DMA), the other (pong) is simultaneously consumed by the compute kernel (e.g., a convolution). When processing completes, the roles swap. This overlapping of data movement and computation hides memory latency, prevents stalls, and sustains high throughput within the DATAFLOW region. Figure 26 illustrates the ping–pong buffering scheme.

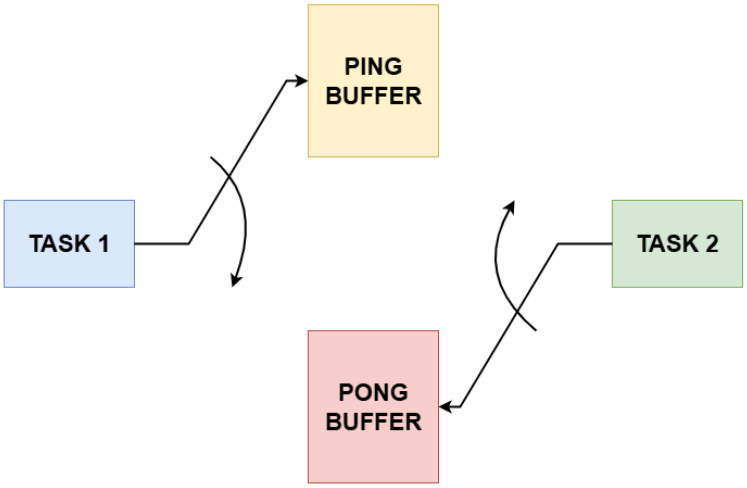


Figure 26: Ping-Pong buffer [26]

Two modules and two memory blocks are employed; particular attention should be paid to the direction of the arrows. While the first task fields one buffer (ping) with new data, the second task can read the data processed by the other buffer (pong) . This arrangement increases throughput, improves overall system performance, and prevents bottlenecks.

An important note is that, in HLS, the source code does not explicitly define two

distinct memory banks across the different functions. Considering the functions convolution and quantization in Figure 19, the convolution stage produces an output array, `out_mem`, which is subsequently read by the quantization stage. This results in an implicit ping–pong buffer: although only a single memory is declared in the code, Vitis HLS automatically realizes a ping–pong scheme by creating two banks. This behavior is evident in the waveform signals shown in Figure 27, where the two ports *q0* and *q1* of `out_mem` are highlighted. In practice, when generating the RTL, Vitis HLS splits the array into two banks to decouple producer and consumer, thereby implementing the ping–pong buffering mechanism.

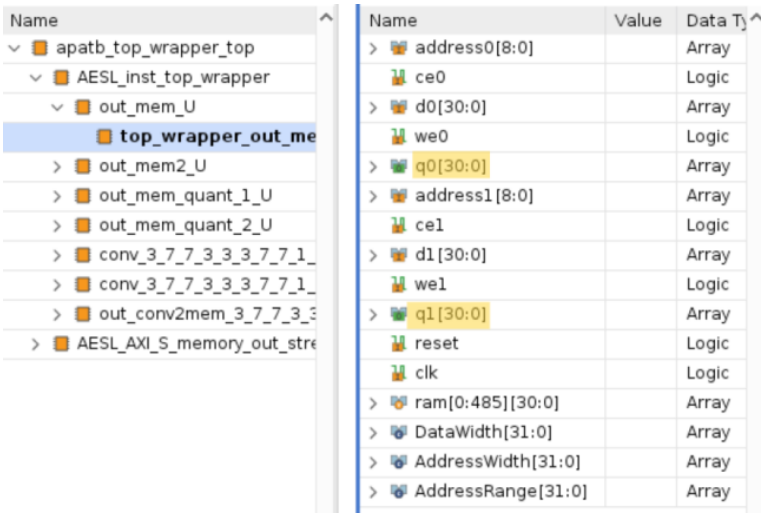


Figure 27: Two bank memory in the waveform signal

5.5.2 HLS Top-Level Control Protocols

In HLS, three control protocols define how a Vitis kernel executes, and they influence dataflow behavior:

- **ap_ctrl_hs** (handshake): exposes the control signals ap_start, ap_done, ap_idle, and ap_ready. The kernel processes one transaction per ap_start assertion and signals completion with ap_done. This mode is convenient when the kernel is orchestrated by software.
- **ap_ctrl_chain** (daisy chain): uses the same handshake signals as above, with the addition of ap_continue (and ap_ready used for chaining) to automatically trigger the next kernel in a sequence without a central controller. Suitable for sequential hardware pipelines.
- **ap_ctrl_none** (free-running): no handshake signals. The kernel remains active continuously; execution is governed by AXI4-Stream handshaking (TVALID/TREADY/TLAST) and internal FIFOs. This can maximize dataflow when producers/consumers are well-balanced.

They are instantiated with the interface pragma, typically on the function return (top level), for example *pragma HLS INTERFACE ap_ctrl_hs*

In the developed code, the free-running control protocol was adopted using: < **pragma HLS INTERFACE ap_ctrl_none** >. This choice reflects the fact that data are exchanged between functions via streams; thus, task execution is naturally governed by the AXI4-Stream handshaking signals (TVALID, TREADY, TLAST) rather than explicit kernel start/stop controls. Keeping the kernel always active facilitates DATAFLOW: the inter-stage pipeline never stalls, there is no per-transaction or per-batch re-arming overhead, and throughput is maximized.

5.5.3 Waveforms

To verify that dataflow is correctly applied, the **waveforms** were inspected in Vivado. Figure 28 shows the signals for two cascaded convolution stages processing five batches of images. The first convolution is shown in green, whereas the second is shown in blue.

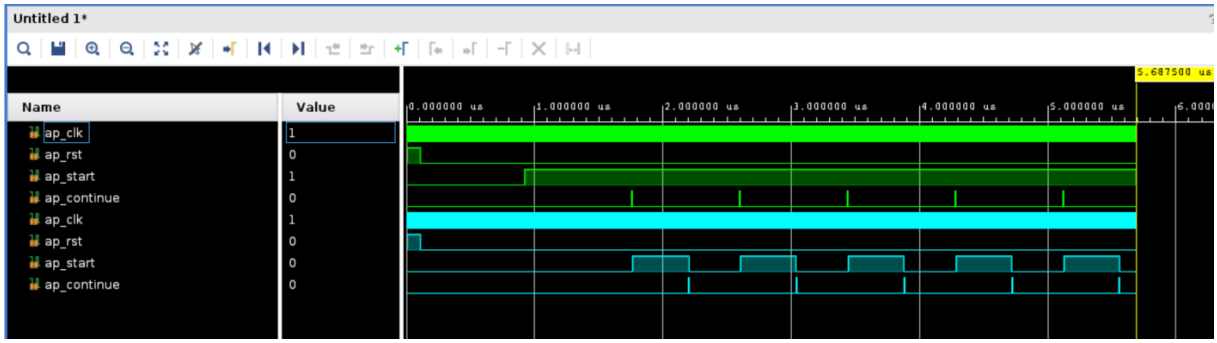
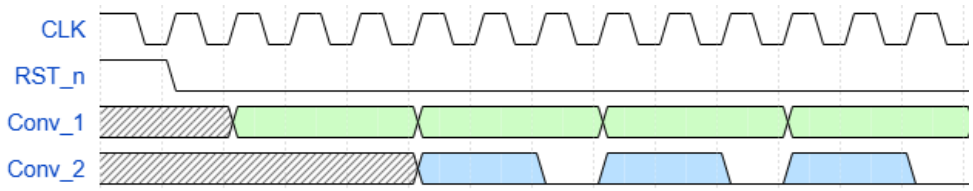


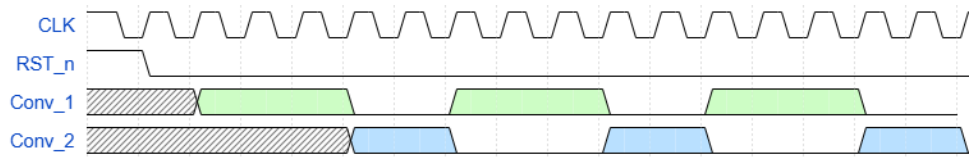
Figure 28: Dataflow

First, observe that per-transaction handshake lines are absent, no ap_done/ap_idle, and only system/control signals such as ap_rst (and, in this view, ap_start/ap_continue) are present, consistent with the free-running control choice discussed earlier. Moreover, as soon as the first convolution begins producing output, the second convolution starts consuming those data while the first convolution immediately proceeds with the next batch. The shorter execution time of the second convolution reflects an imbalance in stage throughput; throughput equalization was not targeted here, since the goal of this experiment was solely to verify that the dataflow mechanism is correctly implemented.

In Figure 29, two timing diagrams are shown to highlight the difference between executions with and without dataflow. In particular, dataflow enables the overlap of work between the two tasks, in this case, the two convolution, allowing them to operate concurrently. This naturally leads to improved performance.



(a) Execution with dataflow



(b) Execution without dataflow

Figure 29: Comparison between executions with and without dataflow

In particular, Figure 29a illustrates the execution where the dataflow pragma is applied. This behavior closely resembles the waveform captured from Vivado, shown in Figure 29a. It can be observed that the second convolution processes the first image, while the first convolution simultaneously begins processing the next image.

In contrast, Figure 29b shows the execution in which the dataflow pragma is not implemented. In this case, the first convolution must wait for the second to complete before starting the next image. It is also visually evident that the second case requires more clock cycles to complete the operations, resulting in a longer execution time and, consequently, reduced performance.

6 Results on ResNet-18

This chapter reports the results obtained after executing the full workflow, from the Vitis HLS project through bitstream generation in Vivado. Results are presented in terms of **resource utilization** (LUT, FF, DSP, BRAM), **power consumption**, and **performance** (latency and throughput). All results refer to a specific FPGA target, the **AMD ZCU102**, whose characteristics are summarized in Section 5.4.

Today's AI workloads demand increasingly complex CNNs. Consequently, it is essential to measure and optimize the above metrics to achieve higher performance. Compared with CPUs, FPGAs offer reconfigurable resources that enable greater parallelism; however, there is a trade-off: pushing parallelism too far can make place & route difficult, increase interconnect delays, and ultimately reduce the achievable clock frequency. It is therefore important to keep resource utilization under control, especially when the target frequency is a key design constraint.

The reported results are related to the implementation of a two specific layers of **ResNet-18**, one of the most widely used networks in contemporary computer vision and object detection. In particular:

- **Single layer** comprising convolution, ReLU, and quantization, as analyzed in Section 4.6.1.
- **Two-layers** sequence comprising the same functions with an additional convolution and quantization pair.

Results are reported for **ICH_PAR=1** and **ICH_PAR=4** to highlight differences in performance and resource utilization. All experiments were conducted at a clock frequency of **200 MHz**, which achieves timing closure with a positive worst negative slack (WNS). The input and output data used in the testbench were extracted directly from the ONNX model and exported in HWC format via a Python script. This enabled fine-grained, function-level debugging by capturing intermediate results.

6.1 One Layer

In this section, the results for a single layer comprising convolution, ReLU, and quantization are presented, as illustrated in Figure 19. A comparison across two ICH_PAR settings highlights the impact of channel-level parallelism on resource utilization and performance, underscoring the importance of parallelizing the computations.

6.1.1 Resource Utilization

The first set of results concerns **resource utilization**, in terms of **LUT, Flip-Flop, DSP and BRAM**. The reported values are taken from Vivado post-implementation,

where the tool applies various optimizations according to the selected strategy. In this case, the chosen strategy is Performance, which tunes placement and routing to improve timing closure. Table 2 reports the results extracted from Vivado. In particular, it compares two settings of **ICH_PAR**, the parameter that controls the parallelization of computations across the input channels.

ICH_PAR	**kLUT**	**kFF**	**DSP**	**BRAM**
1	60.5 (22.2%)	87.6 (16%)	9 (0.4%)	115.5 (12.7%)
4	66.6 (24.3%)	163.4 (29.8%)	36 (1.4%)	127.5 (14%)

Table 2: Resource utilization ResNet-18 of a single layer

A salient observation is that implementing the single layer with a higher degree of parallelism (ICH_PAR) increases the utilization of all resources. This outcome was expected: the objective was to improve latency and throughput while accepting a corresponding rise in resource usage. Particular attention is paid to DSP consumption. With ICH_PAR = 4, the number of DSPs increases by approximately **4×**. This, too, was anticipated, since DSPs are used primarily in the *mem_conv2stream* and *conv* functions; consequently, as the number of concurrent operations grows, DSP utilization scales accordingly.

6.1.2 Performace: Latency

For the single-layer case, performance is evaluated in terms of **latency**. Latency denotes the end-to-end execution time of an inference on a given input, that is, the interval from when the system accepts the input to when the complete output is produced. It is reported in milliseconds (ms). Measurement is performed **on-board** within the PYNQ script using Python's time module.

Metric	**ICH_PAR = 1**	**ICH_PAR = 4**
Latency [ms]	360.7	90.8

Table 3: Latency comparison for different ICH_PAR values of one single layer

Table 3 reports the end-to-end latency for two settings of ICH_PAR. The experimental results show that increasing the degree of parallelism by a factor of four reduces the execution latency by approximately the same factor (**≈ 3.97×**). This indicates near-linear scaling and demonstrates that the system effectively exploits the available parallelism.

6.1.3 Power Consumption

Regarding power consumption, Table 4 reports the execution time, average power, and energy as a function of the degree of parallelism.

Metric	ICH_PAR = 1	ICH_PAR = 4
Total time [s]	0.4	0.1
Mean power [W]	2.9	3.2
Total energy [J]	1.1	0.3

Table 4: Power and energy comparison for different ICH_PAR values of one single layer

The results show that increasing the degree of parallelism from 1 to 4 reduces both the execution time by $\approx \mathbf{4}\times$ (from 0.4 to 0.1) and the total energy consumption (from 1.1 J to 0.3 J, a $\approx \mathbf{3.7}\times$ reduction). By contrast, the average power increases by $\approx \mathbf{1.1}\times$ as parallelism is raised. This behavior stems from the higher resource utilization, which makes leakage power (transistor leakage current) more significant. Nevertheless, the total energy decreases with greater parallelism because the shorter execution time delivers a clear gain in energy efficiency.

6.2 Two Layers

This section reports the results for a two-layers implementation, that is, the previously analyzed layer plus the subsequent convolutional layer (shown in white in Figure 19). Beyond enabling assessment of a more complex topology, this setup allows evaluation of throughput, defined as the number of frames processed per unit time (e.g., fps). To highlight throughput differences, experiments consider not only two degrees of channel parallelism (ICH_PAR = 1 and 4), but also multiimage runs: specifically, **100 images processed in batches of 20** (i.e. 20 images per batch).

6.2.1 Resorse Utilaztion

Table 5 reports the resource utilization results for the two-layers implementation.

ICH_PAR	kLUT	kFF	DSP	BRAM
1	125.1 (45.6%)	238.2 (43.5%)	18 (0.7%)	238.5 (26.2%)
4	133.4 (48.7%)	325.3 (59.3%)	72 (2.9%)	312 (34.21%)

Table 5: Resource utilization ResNet-18 of a two layers

The results show a clear increase in resource utilization compared with the single-layer case reported in Table 2. As in the previous experiments, increasing the degree of parallelism (ICH_PAR) raises all resource counts. In particular, the number of DSPs scales proportionally with parallelism (from 18 to 72), i.e., $\mathbf{4}\times$ increase. This behavior is fully consistent with the earlier analysis: adding a second convolutional layer preserves the $\mathbf{4}\times$ DSP scaling with respect to ICH_PAR.

6.2.2 Performace: Latency & Throughput

Consistent with the previous discussion, this section evaluates not only **latency** but also **throughput** as a function of the degree of parallelism. The study considers both single-image batches and batches of **20 images**, for a total of 100 images. A sufficiently large number of batches is used so that the measured throughput approaches the ideal steady-state value achieved under a fully pipelined regime.

ICH_PAR	**NR_IMG**	**Batch_size**	**Latency (end-to-end)** [ms]	**Throughput** [fps]
1	1	1	720.9	1.4
1	100	20	7555.3	2.7
4	1	1	168.7	5.9
4	100	20	1877.7	10.7

Table 6: Latency results vs ICH_PAR and number of images of two layers

Table 6 reports end-to-end latency and throughput as functions of both the internal parallelism (ICH_PAR) and the batch size. These metrics are computed **on board** via a PYNQ script using Python's time module, which starts and stops timers at the relevant code sections. In particular, the batch latency is computed as:

$$t_{e2e,batch} = \frac{t_{out\_done} - t_{in\_start}}{10^6} \tag{6}$$

The value reported in the table is the average over batches. Throughput is computed as the ratio between the total number of frames processed and the sum of the batch times:

$$fps = \frac{batches \times batch\_size}{\sum t_{e2e,batch}} \tag{7}$$

Analyzing the results shows that increasing the degree of parallelism yields an $\approx \mathbf{4\times}$ reduction in latency, matching the $4\times$ increase in parallelism and consistent with the single-layer case. As the batch size grows, the per-batch latency increases sublinearly thanks to dataflow execution (overlap of transfers and computation). For example, with ICH_PAR=1, the observed increase is $\approx \mathbf{10.5\times}$, not $20\times$ (i.e., not proportional to the batch size of 20).

Regarding throughput, this metric is reported specifically to make the effect of dataflow explicit. In both configurations, processing a single image yields a lower frames-per-second (fps) rate than processing 20 images concurrently. This is expected: throughput measures how many images per second the system processes, and with dataflow correctly implemented, the downstream stage can consume the results of the previous stage while the upstream stage already processes the next image, exactly as illustrated in Figure 28. Batching effectively amortizes pipeline fill/drain overhead and better exploits stage overlap, so the measured throughput with larger batches approaches the steady-state (dataflow) rate, whereas single-image runs remain penalized by startup and teardown effects.

6.2.3 Power Consumption

Table 7 highlights the impact of parallelism and batch size on energy consumption and average power.

ICH_PAR	**NR_IMG**	**Batch_size**	**Total energy** [J]	**Total time** [s]	**Mean power** [W]
1	1	1	2.6	0.7	3.6
1	100	20	140.7	37.8	3.7
4	1	1	0.7	0.2	3.9
4	100	20	38.7	9.4	4.1

Table 7: Power and Energy results vs ICH_PAR and number of images of two layers

Energy consumption is measured directly on the board by leveraging the on-board current and voltage sensors (e.g., INA monitors). The reported total energy is obtained by summing the energies measured for each individual batch, while the average power is computed as the ratio of total energy to total execution time.

The measurements show that energy per image decreases significantly both when the degree of parallelism is increased and when larger batch sizes are employed. In particular, increasing ICH_PAR from 1 to 4 reduces the energy per frame by $\approx \mathbf{3.7}\times$. Moreover, thanks to the use of dataflow, the overall execution time decreases while the average power remains essentially unchanged; consequently, there is a substantial reduction in total energy when moving from processing a single image to processing 20 images per batch, as in this case.

7 Conclusions

This thesis, conducted in collaboration with CEA Saclay, presents the design and realization of a hardware accelerator that supports multiple CNN layers on FPGAs using High-Level Synthesis. Template functions were implemented that currently support two neural networks, LeNet and ResNet-18, and are intended for integration into CEA Saclay's AIdge framework. Using Vitis HLS, the C++ descriptions were synthesized to RTL; subsequently, in Vivado, a Block Design was created and taken through synthesis, implementation, and bitstream generation. The resulting bitstream was programmed onto an AMD **ZCU102** FPGA available in the laboratory, where functional correctness was verified and results were collected in terms of resource utilization (LUTs, flip-flops, BRAMs, DSPs), performance (latency and throughput), and energy consumption.

The adopted strategy is filter reuse (i.e., weight reuse): the entire input tensor is buffered on chip, and one kernel at a time is loaded to perform the computation. The C++ implementation includes the functions required to realize the target layers, specifically convolution, ReLU, and quantization. Particular emphasis was placed early on selecting an appropriate memory-partitioning scheme. This choice not only enables the full input tensor to reside in FPGA on-chip memory, but also supplies the convolution with greater data bandwidth. To exploit channel-level parallelism, a parameter **ICH_PAR** was introduced to process multiple input channels concurrently. While this increases resource utilization, it delivers better performance (lower latency and higher throughput) and reduces energy consumption thanks to the shorter overall runtime. These effects are quantified in Section 6, which evaluates several layers of ResNet-18. The thesis also integrates the work of a colleague who developed the quantization module and is currently implementing the pooling layer. This integration allows additional layers to be composed, with the aim, where FPGA resources permit, of implementing an entire neural network on hardware.

On the other hand, this strategy is limited by the need to store the entire input in on-chip memory. It is therefore suitable for networks whose dimensions are compatible with the target FPGA, or it can be used in a hybrid approach toward the later stages of the network, where the input shrinks while the number of kernels becomes increasingly significant. The principal bottleneck is the kernels: with the full input and a single kernel resident on chip, each new kernel must be fetched from external DDR, which inevitably introduces additional latency. Moreover, convolution biases were stored entirely on the FPGA, which increases BRAM utilization.

7.1 Future Work

This work can be improved in several respects:

- First, an additional DMA can be added to the interface to supply the biases externally; this would reduce on-chip resource utilization.

- Given the benefits obtained by parallelizing computations across input channels, the same approach can be applied to the output channels (`OCH`), so that multiple output channels (i.e. multiple filters) are computed in parallel. This naturally requires storing not just a single kernel in memory, but as many kernels as specified by `OCH_PAR`. In this configuration, several filters are processed concurrently while reusing the same input-activation stream.

A Source code repositories

- Repository implementation of one single layer of ResNet-18
  github.com/lorenzorizzo01/conv_one_layer.git
- Repository implementation of two layers of ResNet-18
  github.com/lorenzorizzo01/conv_two_layer.git

Bibliography

[1] Columbia University. *Artificial Intelligence (AI) vs. Machine Learning*. 2025. URL: https://ai.engineering.columbia.edu/ai-vs-machine-learning/.

[2] Qlik. *Machine Learning vs AI*. 2025. URL: https://www.qlik.com/us/augmented-analytics/machine-learning-vs-ai.

[3] B.J. Copeland. *Artificial Intelligence*. 2025. URL: https://www.britannica.com/technology/artificial-intelligence.

[4] A. M. Turing. *Computing machinery and intelligence*. 1950. URL: https://www.jstor.org/stable/2251299.

[5] Geeksforgeeks. *What is Machine Learning?* 2025. URL: https://www.geeksforgeeks.org/machine-learning/ml-machine-learning/.

[6] Ryan Nash Keiron O'Shea. *Introduction to Convolutional Neural Networks*. 2015. URL: https://arxiv.org/abs/1511.08458.

[7] *ANN vs CNN*. URL: https://how.dev/answers/ann-vs-cnn.

[8] *Convolution*. 2025. URL: https://en.wikipedia.org/wiki/Convolution.

[9] *What are convolutional neural networks?* 2025. URL: https://www.ibm.com/think/topics/convolutional-neural-networks#:~:text=Neural%20networks%20are%20a%20subset,layers%2C%20and%20an%20output%20layer..

[10] Luke Miller. *High Level Synthesis – It's for Real*. 2013. URL: https://semiwiki.com/fpga/xilinx/2222-high-level-synthesis-its-for-real/.

[11] Anna University. *FPGA (Field Programmable Gate Arrays)*. URL: https://eee.poriyaan.in/topic/fpga--field-programmable-gate-arrays--11689/.

[12] Zhuoyuan Liu Yunxiang Hu Yuhao Liu. *A Survey on Convolutional Neural Network Accelerators: GPU, FPGA and ASIC*. 2022. URL: https://ieeexplore.ieee.org/abstract/document/9730377/figures.

[13] *NEUROKIT2E*. 2025. URL: https://neurokit2e.eu/.

[14] CEA LIST. *Aidge*. 2024. URL: https://eclipse-aidge.readthedocs.io/en/latest/.

[15] Sandrine Varenne. *Aidge: Independent Deep Learning framework for embedded AI*. 2025. URL: https://list.cea.fr/en/aidge/.

[16] *Concepts of Pipelining*. 2021. URL: https://witscad.com/course/computer-architecture/chapter/concepts-of-pipelining.

[17] AMD. *pragma HLS array$_p$artition*. 2025. URL: https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-array_partition.

[18] *What Is ResNet-18?* URL: https://blog.roboflow.com/resnet-18/.

[19] Sampa Misra. *Multi-Channel Transfer Learning of Chest X-ray Images for Screening of COVID-19*. 2020. URL: https://www.researchgate.net/publication/343957273_Multi-Channel_Transfer_Learning_of_Chest_X-ray_Images_for_Screening_of_COVID-19.

[20] Bryan Clark. *What is quantization?* 2024. URL: https://www.ibm.com/think/topics/quantization.

[21] AMD. *What is the Vivado Design Suite?* 2025. URL: https://docs.amd.com/r/en-US/ug910-vivado-getting-started/What-is-the-Vivado-Design-Suite.

[22] AMD. *SDC and XDC Constraint Support*. 2025. URL: https://docs.amd.com/r/en-US/ug904-vivado-implementation/SDC-and-XDC-Constraint-Support.

[23] AMD. *AMD Zynq UltraScale+™ MPSoC ZCU102 Evaluation Kit*. 2025. URL: https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/ek-u1-zcu102-g.html.

[24] AMD. *Optimizing with Dataflow*. 2022. URL: https://xilinx.github.io/Vitis-Tutorials/2020-1/docs/build/html/docs/Hardware_Accelerators/Design_Tutorials/01-convolution-tutorial/dataflow.html.

[25] AMD. *pragma HLS dataflow*. 2025. URL: https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-cache.

[26] F. Morgado-Dias Darío Baptista Leonel Sousa. *Raising the Abstraction Level of a Deep Learning Design on FPGAs*. 2020. URL: https://www.researchgate.net/publication/347478182_Raising_the_Abstraction_Level_of_a_Deep_Learning_Design_on_FPGAs.