# POLITECNICO DI TORINO

Master's Degree in Electronic Engineering


Master's Degree Thesis

# Hierarchical Architecture of a 2D-CNN Accelerator: Leveraging High-Level Design and Embedded Scalable Platform (ESP)

Supervisors

Prof. Mario Roberto CASU

Dr. Luca URBINATI

Candidate

Angelina MARRA

Academic year 2023-2024

# Summary

The rapid expansion of big data applications, while unlocking immense potential for Machine Learning (ML) advancements, has placed overwhelming demands on processing speed and scalability that traditional computing systems can no longer efficiently handle. Conventional von Neumann architectures, which separate data storage from processing units, face significant limitations due to frequent back-and-forth data transfers between the CPU and memory. This inefficiency creates performance bottlenecks and hampers energy efficiency, challenges that are magnified by the vast datasets utilized in Artificial Intelligence (AI) tasks. To overcome these obstacles, new computing platforms tailored specifically for AI have emerged. These domain-specific systems, called hardware accelerators, offer significant power and performance gains by addressing critical issues such as the "memory wall" and "power wall", enabling them to manage the computational load more effectively than general-purpose architectures [1].

In this thesis, starting from the sequential architecture of a 2D-Convolution hardware accelerator realized in a previous work [2], the corresponding hierarchical architecture is implemented leveraging High-Level Design (HLD) and Embedded Scalable Platform (ESP).

- HLD allows the designer to implement loosely-timed or un-timed behavioral descriptions C/C++ of the accelerator, concise and easy to debug, which can be synthesized into RTL using High-Level Synthesis (HLS) tools like Catapult HLS by Siemens [3].
- The ESP platform, developed by the Columbia University, supports HLS-based flows and facilitates the seamless integration of the accelerator's architecture into an FPGA-based prototype of a complex System-on-Chip (SoC), basically composed by a processor tile, a memory tile, an accelerator tile and an auxiliary tile [3].

In particular, in order to obtain the hierarchical architecture, where all the phases performed by the hardware accelerator (configuration, load, compute, store) are implemented inside functions running in parallel, specific coding style and HLS directives are employed. Once the architecture is validated through C++ simulations,

synthesized through Catapult and integrated into a SoC using ESP, the bitstream of the SoC is generated using Xilinx Vivado and uploaded into the FPGA. Together with the hierarchical implementation of the accelerator, the previous sequential implementation [2] is validated and synthesized too, to validate some functionalities that were not tested in the previous work [2] and to obtain a version that can be compared with the hierarchical one. Finally, FPGA deployment is performed involving the use of a bare-metal application that implements a tiling algorithm developed in a previous work [2]. Specifically, for the tests on the hardware, two different layers are utilized, and various tiling strategies are applied. The results demonstrate that the hierarchical architecture improves the performance of the sequential architecture [2] by approximately 33%. They also indicate that the hierarchical accelerator achieves optimal performance when the number of tiles to process is relatively low.

# Acknowledgements

I would like to extend my heartfelt gratitude to my family and friends for their constant support and encouragement throughout the years.

Additionally, I would like to express my sincere thanks to my supervisors, Prof. Mario Roberto Casu and Dr. Luca Urbinati, for giving me the opportunity to work on such an interesting and rewarding project and for their essential guidance and assistance.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Hardware Accelerators

Classical philosophy once described human thought as the mechanical manipulation of symbols, and for centuries, people have aspired to create artificial beings with intelligence and consciousness. This was the foundation of what we now call Artificial Intelligence (AI). In 1950, Alan Turing proposed "the imitation game" (now known as the Turing Test), the mathematical possibility of creating an intelligent machine. The 1956 Dartmouth summer research project on AI officially marked the beginning of AI as a research field. Since then, AI has seen periods of both progress and decline. Recently, with the advent of big data and the growing computing power, AI has surged in importance, attracting widespread attention and investment. Machine Learning (ML) techniques have been successfully applied to numerous challenges in both academia and industry [1].

Originally, since the biological brain is considered as the most complex and efficient “machine”, ML algorithms sought to mimic its behavior. Like biological nervous systems, ML algorithms are composed of neurons and synapses responsible for information processing and feature extraction. Various neuron models, such as the McCulloch–Pitts, Sigmoid, and Rectified Linear Unit (ReLU), are used for nonlinear feature extraction and Neural Network (NN) training [1].

Despite its tremendous potential for ML’s devlopment, the explosive growth in big data applications places significant demands in terms of processing speed and scalability on traditional computing systems. Conventional von Neumann architectures separate components related to processing and data storage, are limited by frequent data transfers between processors and memory, leading to bottlenecks in performance and energy efficiency. These challenges are amplified by the massive volume of data used by AI applications. As a result, computing platforms for AI applications have emerged as domain-specific computing systems designed to overcome these challenges, offering significant power and performance improvements by addressing issues like the "memory wall" and "power wall" [1]. These platforms, specifically designed for AI tasks, often feature highly parallelized

computing and storage units arranged in a two-dimensional way (2D) to facilitate common matrix-vector operations in NN. At the core of their innovation three key pillars lie: biological theory foundation, hardware design, and algorithms (software) [1].

As this thesis centers on accelerating *2D Convolutional Neural Networks* (*2D CNNs*), from now on we will focus on the hardware implementation of this algorithm.

The main component in convolutional layers are Multiply-And-Accumulate (MAC) operations, which lend themselves well to parallelization. Systems often utilize highly-parallel computing paradigms to achieve optimal performance, with both temporal and spatial architectures being common approaches [4].

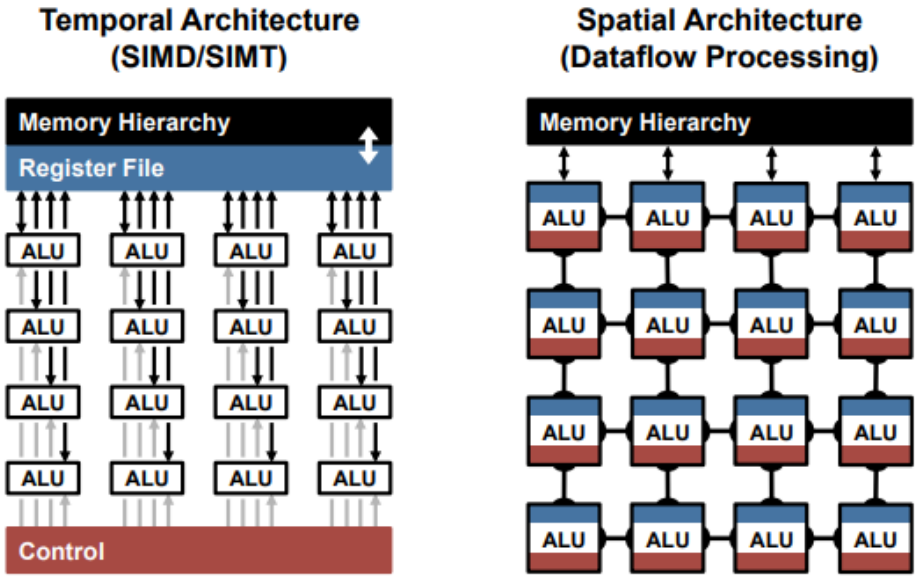


**Figure 1.1:** Highly-parallel compute paradigms: temporal architecture (on the left) and spatial architecture (on the right). Source [4].

**Temporal architectures** are typically found in *CPUs* and *GPUs*. These architectures enhance parallelism relying on centralized control of multiple Arithmetic Logic Units (ALUs), which access data from the memory hierarchy but cannot directly communicate with each other [4].

On the other hand, **spatial architectures** utilize dataflow processing, where ALUs are organized into a processing chain and pass data directly between one another. Each ALU can sometimes be equipped with its own control logic and local memory, such as a scratchpad or register file, effectively turning it into a Processing Engine (or Processing Element) (PE). Spatial architectures are widely used for

the implementation of CNNs, particularly in *Application-Specific Integrated Circuit* (*ASIC*) and *Field-Programmable Gate Array* (*FPGA*) designs, which are also called *hardware accelerators* [4]. The main goal of hardware acceleration is to enhance computational speed by employing custom-designed hardware tailored for specific routines or algorithms [5].

For CNNs, memory access is the primary bottleneck during processing. Each MAC operation requires three memory reads (for the filter weight, feature map activation, and partial sum) and one memory write (for the updated partial sum), as shown in Fig. 1.2. In the worst-case scenario, all of these memory accesses would rely on off-chip DRAM, significantly impacting both throughput and energy efficiency [4]. Given the intensive computations and extensive external data access required by CNN algorithms, CPUs and GPUs implementations struggle to meet real-time demands [6].

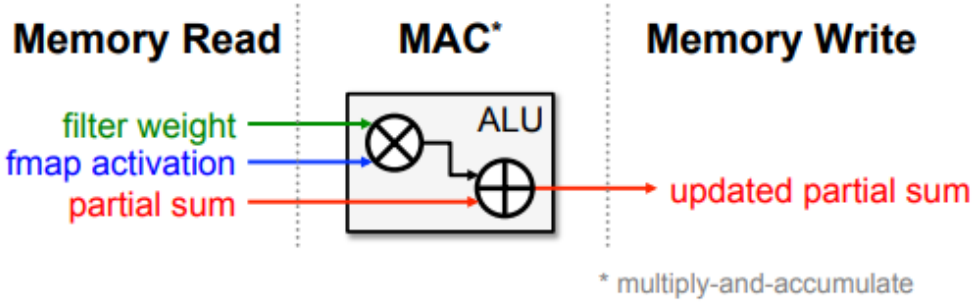


**Figure 1.2:** Read and write access per MAC. Source [4].

Spatial architectures, offer a solution to reduce the energy cost associated with data movement. This is achieved by incorporating multiple levels of local memory hierarchy with varying energy costs, as illustrated in Fig. 1.3. The hierarchy includes a large global buffer of several hundred kilobytes that interfaces with the DRAM, an inter-PE network that allows data to be passed directly between ALUs, and a small Register File (RF) within each Processing Element (PE), typically only a few kilobytes in size. These layers of memory hierarchy improve energy efficiency by enabling low-cost data access. For instance, retrieving data from the RF or neighboring PEs consumes 1 to 2 orders of magnitude less energy than fetching it from the DRAM [4].

The hardware accelerator architecture is illustrated in Fig. 1.4. This setup allows the CPU to handle other tasks while the hardware accelerator takes over the computation for the specific routine. The CPU's involvement is limited to initiating the co-processor and waiting for the results. To achieve performance gains, the overhead of communicating with the co-processor must be lower than

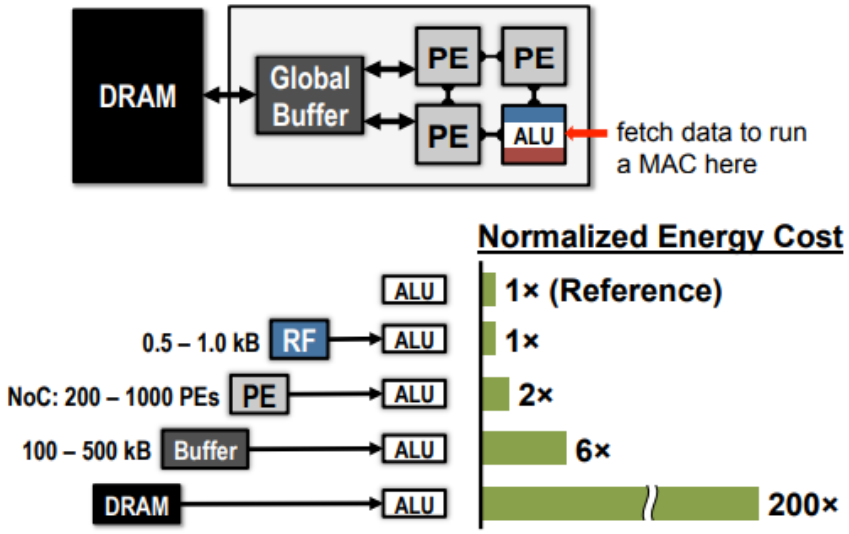


**Figure 1.3:** Memory hierarchy and data movement energy. Source [4].

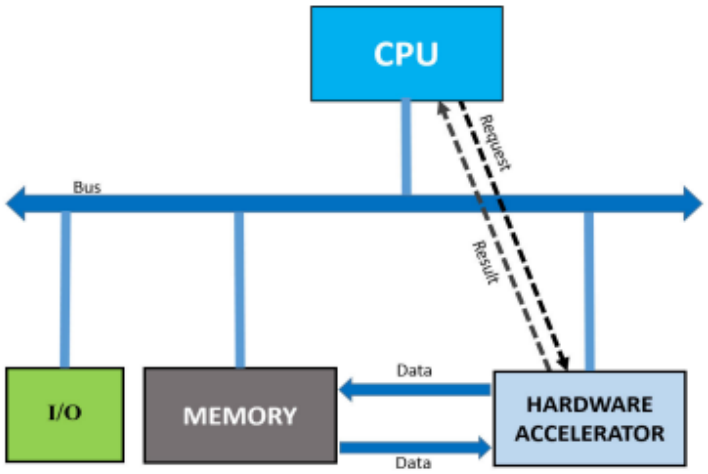


**Figure 1.4:** Hardware Accelerator architecture. Source [5].

the cost of performing the computation itself [5].

Additionally, FPGAs offer deterministic performance and latency, as long as the application permits. Unlike CPUs and GPUs, which depend on complex protocol stacks and are subject to Operating System (OS) resource scheduling, FPGAs can provide precise latency measurements since they operate independently from an OS scheduler. Moreover, FPGAs are highly reliable and require less development

time compared to ASICs. This makes them a superior option for hardware-based acceleration tasks, especially in the early stage of deployment, for small-scale productions and for their reconfiguration capabilities that allow to deploy different AI algorithms over time [5].

# Chapter 2

# Embedded Scalable Platform (ESP)

One of the key effects of the slowdown of Moore's Law and the end of Dennard scaling has been the increasing complexity of chip design. To achieve both performance and energy efficiency, heterogeneous *System-On-Chip* (*SoC*) architectures—combining multicore processors and specialized hardware accelerators—have become the preferred solution across various application domains. The costs of SoC development rise, therefore to reduce them there is a growing need for new methodologies and platforms that emphasize design reuse. *Open-Source Hardware* (*OSH*) offers a valuable solution to support design reuse by fostering innovation and collaboration between industry and academia. The success of the RISC-V open standard Instruction Set Architecture (ISA) has driven to a wave of new SoC designs. With more OSH components available, the open-source community requires *Computer-Aided Design* (*CAD*) methodologies to transform these into domain-specific SoC designs. These methodologies must emphasize flexibility, robustness, and scalability, while supporting the logic design, system-level integration and physical design of these complex systems [7].

A methodology is:

- *flexible* when it allows seamless integration of diverse OSH components across different technologies and tools while meeting performance, power, and area (PPA) requirements [7].
- *robust* if it ensures functional correctness from Register-Transfer Level (RTL) specification to final implementation by detecting design issues early and minimizing errors [7].
- *scalable* when it is able to manage the growing complexity of SoC designs

without exponential increases in computational resources, engineering effort, or design time, ensuring efficiency in large-scale projects [7].

A flexible, robust and scalable methodology for the agile physical design and programming of heterogeneous FPGA-based SoC architectures is build on the *ESP* (*Embedded Scalable Platform*) platform. ESP is the result of more than a decade of research at Columbia University [7].

The following explanation about the ESP Architecture and Methodology is taken from references [7] [3] [8] [9].

## 2.1 The ESP Architecture

The ESP architecture is designed as a heterogeneous grid of tiles, Fig. 2.1. For a specific application domain, the architect determines the structure of the SoC by selecting the number and type of tiles. The tiles are of four types: processor tiles, accelerator tiles, memory tiles for communication with the main memory, and auxiliary tiles for peripherals (such as UART and Ethernet) or system utilities (such as the interrupt controller and timer). To ensure high scalability, ESP tiles are connected through a multiplane *NoC* (*Network on Chip*).

Each tile's content is encapsulated in a *socket*, which connects the tile to the NoC and implements the platform services. This socket-based design approach, which decouples the design of the tile from the design of the rest of the system, is a key element of the agile ESP SoC design process. It simplifies the tile development by handling all the system integration tasks and facilitates the reuse of the blocks. For example, the ESP accelerator socket provides services like *Direct Memory Access* (*DMA*), cache coherence, performance monitoring and distributed interrupt requests.

Unlike other open-source hardware platforms, ESP adopts a system-centric perspective rather than a processor-centric one, where processors and accelerators are equally prioritized within the SoC.

Let us now analyze more in detail the different types of tiles that the ESP tool comprises:

- **Processor Tile:** Each processor tile includes a processor core selected at design time from the available options. Currently, the choices are between the 32-bit SPARC Leon3 core, the 32-bit RISC-V Ibex core, and the 64-bit RISC-V CVA6 (formerly known as Ariane) core. All the cores support Linux and feature their own private L1 caches. Processor integration into the distributed ESP system is seamless, requiring no ESP-specific software patches to boot

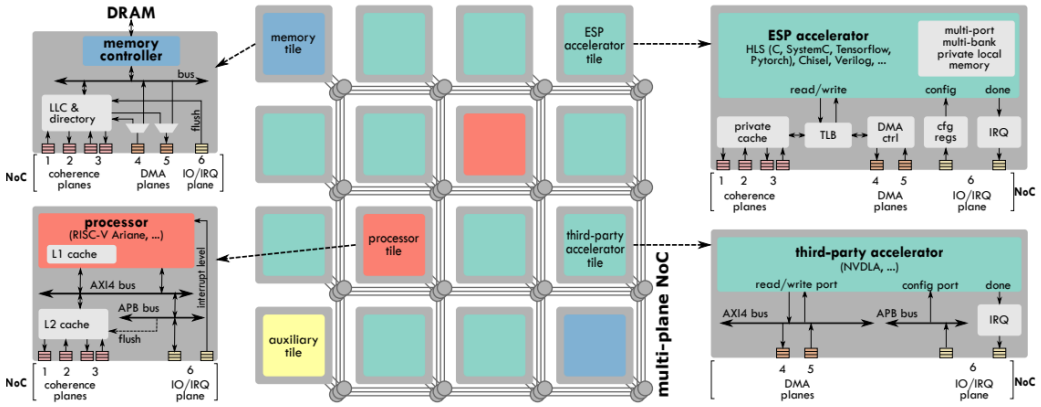


**Figure 2.1:** An instance of an ESP SoC architecture with a 4×4 tile grid. Source [8].

Linux. Each processor operates on a local bus and remains unaware of the broader system. In addition, the socket of this tile augments the processor with a private L2 cache of configurable size. The IO/IRQ NoC plane supports *Input-Output* (*IO*) and *Interrupt Request* (*IRQ*) channels, typically used by processors and accelerators to communicate with each other.

- **Memory Tile:** Each memory tile includes a channel to the external DRAM, and the number of memory tiles can be configured during the design phase. Typically, this number ranges from one to four, depending on the size of the SoC. The necessary hardware logic for partitioning the addressable memory space is automatically generated, and this partitioning is completely transparent to the software. Additionally, when the ESP cache hierarchy is enabled, the memory tile contains the ESP *Last-Level Cache* (*LLC*) that can handle DMA requests directly from accelerators.

- **Accelerator Tile:** This tile contains the specialized hardware of a loosely-coupled accelerator, which executes its tasks independently from the processors while exchanging large datasets with the memory hierarchy. *Private Local Memories* (*PLMs*) are also present inside each accelerator tile to store local data and are generally much smaller than the external memory. To be integrated into the ESP tile, accelerators must follow a simple interface that includes load/store ports for latency-insensitive channels, signals to configure and start the accelerator and an acc_done signal to notify completion and trigger an interrupt to the processors. Newly designed ESP accelerators created by using one of the supported design flows, automatically adhere to this interface. For pre-existing accelerators, ESP provides a third-party integration flow.

In this latter case, the accelerator tile includes only a subset of the proxy components, as standard bus adapters replace configuration registers (DMA for memory access and Translation Lookaside Buffer (TLB) for virtual memory). The platform services provided by the socket, such as address translation, DMA, configuration registers and coherence, relieve designers from the need to reimplement fundamental capabilities, when they want to focus on the optimization of their accelerators.

- **Auxiliary Tile:** The auxiliary tile hosts all shared peripherals in the system, except for memory, including a digital video interface and a monitor module that collects various performance counters. For this reason the auxiliary tile's socket is the most complex, since it must provide a wide range of platform services for the devices it hosts.

## 2.2 The ESP Methodology

ESP allows to quickly create FPGA-based prototypes of complex SoCs, facilitating the seamless integration of third-party OSH components. SoC architects can also integrate accelerators developed using a variety of supported design flows, most of which are automated and supported by commercial CAD tools. As illustrated in Fig. 2.2, the accelerator design flow (on the left) assists in building an Intellectual Property (IP) library, while the SoC flow (on the right) automates the integration of heterogeneous components into a complete SoC.

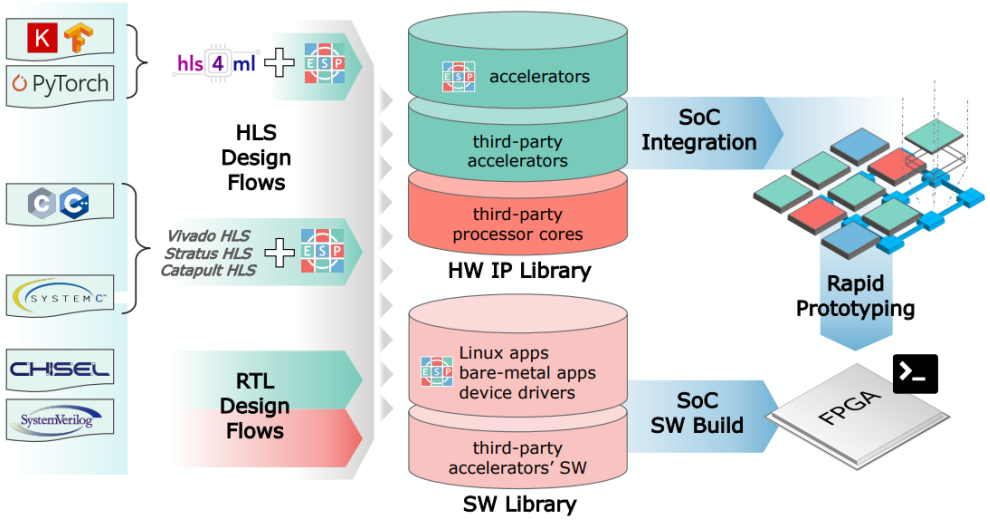


**Figure 2.2:** Agile SoC design and integration flows in ESP. Source [3].

The ultimate goal of this process is to expand the accelerator library with new elements that can be automatically instantiated within the SoC flow. Designers have the flexibility to work at different abstraction levels using various specification languages, including:

- Cycle-accurate RTL descriptions in languages such as VHDL, Verilog, SystemVerilog, or Chisel.
- Loosely-timed or un-timed behavioral descriptions in SystemC or C/C++, which can be synthesized into RTL using *High-Level Synthesis* (*HLS*) tools. ESP currently supports the three major commercial HLS tools: Cadence Stratus HLS, Siemens Catapult and Xilinx Vivado HLS.
- Domain-specific libraries for deep learning, such as Keras TensorFlow, PyTorch and ONNX.

For **HLS-based flows**, ESP simplifies the work of accelerator designers by offering ESP-compatible accelerator templates, HLS-ready skeleton source code, various examples and step-by-step tutorials for each flow. The increasing adoption of HLS from C/C++ code can be attributed to several factors:

- The extensive codebase of algorithms written in these languages.
- The simplified hardware/software co-design (as most embedded software is in C).
- The significantly faster functional verification of the C/C++ source code compared to RTL simulations.

The goal for an ESP accelerator designer is to create a well-structured description that partitions the source code into concurrent functional blocks. The aim is to produce a synthesizable source code that enables exploration of a wide design space by evaluating various micro-architectural and optimization choices. Fig. 2.3 illustrates the relationship between the C/C++/SystemC design space and the RTL design space. HLS tools provide a rich set of configuration knobs, represented by the green arrows, which allow for the generation of multiple RTL implementations, each offering different trade-offs between cost and performance. Designers can also make manual transformations (orange arrows) to explore the design space while maintaining functional integrity. For example, they can expose parallelism, by eliminating false dependencies, or reduce resource usage, by encapsulating code sections with similar behavior into reusable functions.

There are many HLS tools. **Catapult HLS by Siemens** is the one we used in this thesis to design our accelerator. As illustrated in the red box in Fig. 2.4, starting from the design source code in C, C++ or SystemC describing the structure

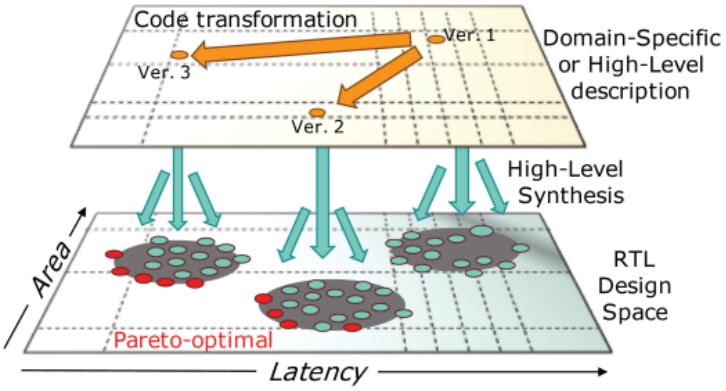


**Figure 2.3:** HLS-based accelerator design in ESP. Source [3].

and behavior of the design, Catapult is able to synthesize interfaces, data structures and loops for a targeted ASIC or FPGA technology, ultimately producing an optimized RTL implementation that is ready for simulation and gate-level synthesis and includes the following files:

- HDL files (VHDL, Verilog).
- RTL simulation and synthesis scripts—Simulation and synthesis scripts for specified downstream tools such as *Synopsys DesignCompiler*.
- Analysis reports.

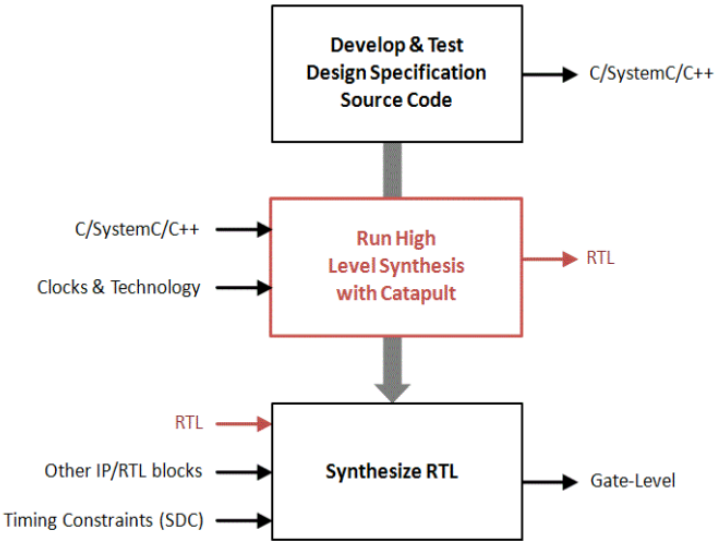


**Figure 2.4:** High Level Flow. Source [9].

Let us now analyze all the steps of the **HLS-based accelerator's design and integration flow in ESP**, Fig. 2.5.

- **Accelerator design flow:**
  - **Generate skeleton:** Designers can automatically generate a fully functional, HLS-ready accelerator skeleton by specifying a small set of parameters. These include a unique name and ID, the desired HLS tool and details about the accelerator's input and output data. The skeleton is provided by ESP with a unit testbench, synthesis and simulation scripts, a bare-metal test application and a Linux device driver accompanied by a test application.
  - **Customize:** Starting from the automatically generated skeleton, designers need to customize the computational part of the accelerator. They are also responsible for adjusting the input-generation and output-validation functions in the unit testbench, as well as in the bare-metal and Linux test applications.
  - **Generate RTL with HLS:** Designers can automatically generate one or more RTL implementations of an accelerator with a simple command that executes the selected HLS tool. The RTL code generated by HLS is automatically added to the ESP library of IP blocks for integration.
  - **Test:** The validation step involves running the accelerator's unit testbench, which simulates the behavior of the accelerator's tile socket.
- **SoC design flow:**
  - **Software build:** ESP automates the creation of both the bare-metal binaries and the Linux image for testing, together with their Make targets.
  - **Soc configuration:** The ESP Graphical User Interface (GUI) assists designers in configuring an SoC by allowing them to choose the number, type, and position of tiles, along with other design options such as processor type or cache size. Based on these configurations, ESP generates the complete RTL implementation of the SoC, including the tile sockets required by each accelerator.
  - **FPGA prototyping:** ESP users can prototype their SoC on any supported FPGA board without requiring prior FPGA experience. The generation of the bitstream file, the programming script and the software deployment are fully automated. FPGA prototyping helps identify and address potential design issues and performance bottlenecks in a timely manner.

– **Full-system simulation:** This simulation encompasses both hardware and software components within a virtual environment. This type of simulation facilitates comprehensive testing and evaluation of the SoC's functionality and performance and the interactions between its various components. By performing a full-system simulation, potential issues or conflicts can be detected and resolved before the physical implementation of the SoC.

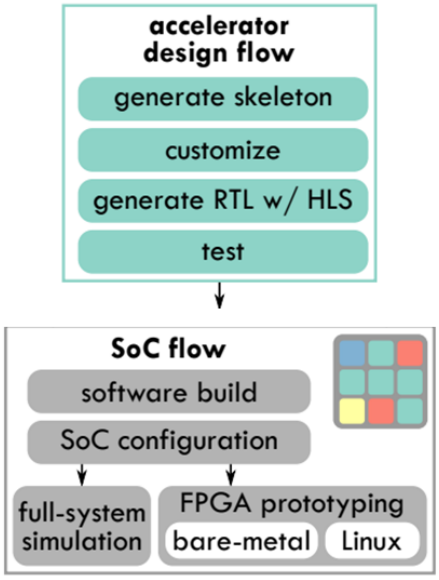


**Figure 2.5:** HLS-based accelerator's design and integration flow in ESP. Source [8].

# Chapter 3

# Summary of previous work

In this chapter, the thesis of Diego Ricardo Bueno Pacheco, "*Efficient Tiling Architecture for Scalable CNN Inference: Leveraging High-Level Design and Embedded Scalable Platform*" [2], is summarized, since it represents the starting point for this actual thesis.

As the demand for deep learning and computer vision applications continues to grow, there is an increasing need for more efficient and scalable solutions to meet the computational requirements of advanced CNN models. Additionally, the push to extend these applications to a wider range of devices without relying on cloud-based solutions has resulted in a shift towards performing computations on edge devices. However, this shift poses a significant challenge due to the limited computational power and memory resources available on such devices. Overcoming this challenge necessitates the use of hardware accelerators and the division of tensors into smaller, manageable tiles that fit within memory constraints.

In this context, the thesis in [2] presents an innovative tiling architecture designed to enable large-scale CNN inference, with a specific focus on leveraging the capabilities of High-Level Design (HLD) and ESP. The tiling algorithm considers several important factors, including the organization and addressing of tensors in the external memory, the maximum number of processing elements available in the accelerator (e.g., MAC units), the required precision for MAC operations (e.g., 16, 8, or 4 bits), and the memory capacities of PLMs within the accelerator. These considerations are carefully incorporated into the tiling algorithm to optimize performance and resource usage. The proposed architecture is thoroughly tested through RTL simulation and FPGA deployment, demonstrating its practicality and effectiveness in real-world applications.

## 3.1 2D-Conv Accelerator Based on the Sum-Together Multiplier

One effective way to balance performance and accuracy when working with Deep Neural Network (DNN) layers is through the use of hardware accelerators that integrate Precision-Scalable MAC units (PSMACs). One approach to PSMACs is based on the **Sum-Together (ST) multiplier** [10], which is described in Fig. 3.1 and Fig. 3.2. The following explanation about it is taken from the references [10] and [11]. Depending on the CONFIG configuration signal, the multiplier can perform either one 16×16, one 16×8 multiplication, two 8×8, two 8×4 or four 4×4 dot products in parallel, utilizing signed operands packed into the 16-bit inputs A and B. Based on the configuration, either a portion or the entire multiplier's 32-bit output P will contain the result of the operation.

The focus on these specific precisions is due to several reasons. In applications demanding high accuracy, 16-bit quantization of activations and weights is a common choice. This is particularly relevant for safety-critical applications such as image segmentation in foggy conditions for autonomous driving. The 8-bit precision, often the default for quantizing DNNs to maintain performance without degradation, is widely used. For scenarios requiring smaller bit-widths, 4-bit quantization techniques provide for most applications a good balance between reducing model size and maintaining performance.

The asymmetric configurations (such as 16×8 and 8×4) are included because they allow for efficient packing of lower-precision operands, like DNN weights, while maintaining the precision of other operands, like DNN activations. This helps reduce the memory footprint of ML models.

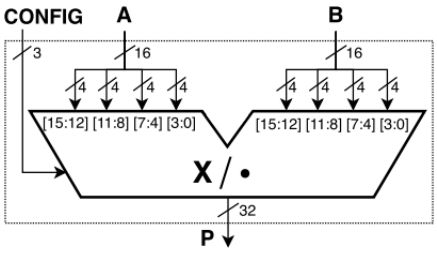


**Figure 3.1:** Generic ST multiplier. Source [11].

| *CONFIG* | ST multiplier's output |
|---|---|
| 16×16 (000b) | $P_{[31:0]} = A_{[15:0]} \times B_{[15:0]}$ |
| 16×8 (100b) | $P_{[31:0]} = A_{[15:0]} \times B_{[7:0]}$ |
| 8×8 (010b) | $P_{[16:0]} = A_{[15:8]} \times B_{[7:0]} + A_{[7:0]} \times B_{[15:8]}$ |
| 8×4 (011b) | $P_{[16:0]} = A_{[15:8]} \times B_{[3:0]} + A_{[7:0]} \times B_{[11:8]}$ |
| 4×4 (001b) | $P_{[9:0]} = A_{[15:12]} \times B_{[3:0]} + A_{[11:8]} \times B_{[7:4]} + A_{[7:4]} \times B_{[11:8]} + A_{[3:0]} \times B_{[15:12]}$ |

**Figure 3.2:** Configurations of a ST multiplier. Source [11].

Let us now analyze the working principle of the **2D-Conv accelerator** implemented that integrates ST multipliers within its MAC units. Fig. 3.3 illustrates the various access patterns (highlighted in red) employed by the accelerator to read data from the activation (blue) and weight (orange) tensors, and how these data are packed into the 16-bit inputs of the ST multipliers. For each filter (orange) with C kernels, a MAC unit in the 2D-Conv accelerator perform the multiplications of the C channels of the input tensor (blue) with the corresponding weight kernels and the channel-wise accumulation of these multiplications. At full precision (N = 1), the ST multiplier in the MAC unit processes the activations and weights from one input channel at a time. However, at lower precision, the ST multiplier is fed with pairs of activation and weight data from either two (N = 2) or four (N = 4) input channels simultaneously. By utilizing the dot-product capability of the ST multiplier, this approach ideally reduces the number of MAC cycles to C/N and lowers the latency, scaling at 1/N.

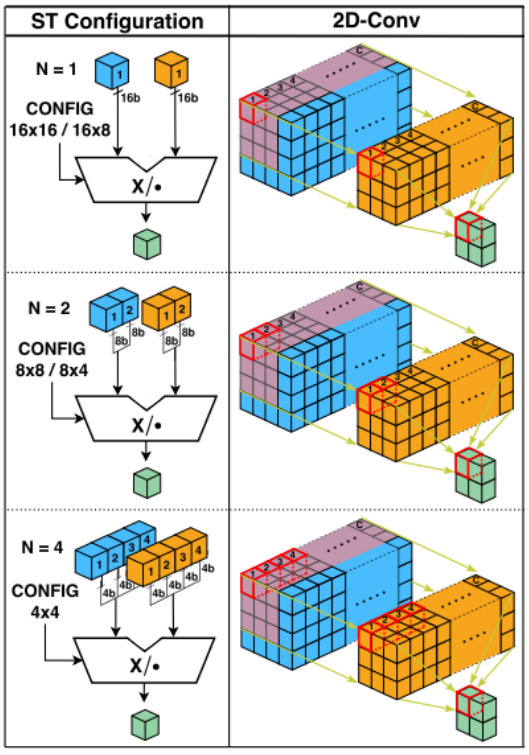


**Figure 3.3:** Working principle of the 2D-Conv accelerator based on the ST multiplier. Source [11].

## 3.2 Architecture of the 2D-Conv NN

In this section the software implementation will be explained, focusing on the tiling algorithm used in [2]. Then, the hardware implementation of the sequential architecture of the accelerator will be also addressed.

### 3.2.1 Software Implementation: Tiling Algorithm

Several techniques have been developed to reduce the memory demands of CNNs and minimize the transfer of partial results between intermediate layers, which are often stored in off-chip memory and then brought back to on-chip memory. This back-and-forth process increases both latency and energy consumption. Tiling methods are a set of strategies that divide CNN operations into smaller chunks, or tiles, helping to strike a balance between computational efficiency and memory constraints, enabling real-time, energy-efficient processing of deep neural networks. However, choosing the right tiling method and optimizing its implementation requires trade-offs between memory usage, processing speed, and model performance, often involving careful experimentation and fine-tuning for specific applications.

The tiling algorithm implemented takes into account several key aspects of the accelerator architecture, as well as the topological characteristics, dimensions of the tensors, typically found in CNN layers. These elements are used as inputs to the tiling algorithm to determine tile sizes and ordering constraints. The main considerations can be summarized as follows:

- Primary Objective: The goal of the tiling algorithm is to fit the input, weight and output data into the accelerator's PLMs. To achieve this, the algorithm must know the various sizes of these PLMs, allowing it to recursively check if a given tile will fit.

- Fixed Loop Bounds: The accelerators' for loops should have fixed upper limits, where each loop works on one dimension of the tensor. This implies that each tensor dimension has a maximum value supported by the accelerators, and these values must be considered when determining tile dimensions.

- MAC Unit Distribution: The accelerators contain multiple MAC units. Each Processing Element (PE) operates on a different set of data. For example, in 2D convolution (2D-Conv), each PE works on a different output channel.

- Precision-Scalable Accelerator Consideration: The tiling architecture assumes the use of a precision-scalable (PS) accelerator based on ST multipliers. As explained in Sec. 3.1, when operating in low-precision mode, multiple input and weight values are fetched for each PS multiplier. For 2D-Conv, this involves

up to 4 channels at the lowest precision. To maximize the benefits of the PS accelerator, the tiling algorithm must consider the different values mentioned in the previous bullet points and split the input or weight tensors according to the precision configuration, allowing for efficient use of low-precision modes.

- Data Replication in Tiling: When tiling the input tensor along the height or width dimensions, sliding process used in convolution requires replicating portions of the data in adjacent tiles to ensure correct processing. Therefore, this algorithm avoids tiling the input tensor in the width dimension and avoids tiling the weight tensor in both the width and height dimensions.

With these requirements in mind, a set of inequalities is derived for each accelerator, taking inspiration from [12]. Specifically, three memory size conditions must be satisfied for each PLM used. The inequalities are:

$$PLM_{IN} \geq H_{IN} * W_{IN} * C_{IN} \tag{3.1}$$

$$PLM_{W} \geq H_{kernel} * W_{kernel} * C_{IN} * C_{OUT} \tag{3.2}$$

$$PLM_{OUT} \geq H_{OUT} * W_{OUT} * C_{OUT} \tag{3.3}$$

Inequality 3.1 evaluates if the input PLM size is is greater or equal to the multiplication of the input tensor dimensions: height, width and channel dimension. Inequality 3.2 evaluates if the weight PLM size is greater or equal to the multiplication of the weight tensor dimensions: kernel height, kernel width, channel input, and channel output dimension. Inequality 3.3 evaluates if the output PLM size is greater or equal to the multiplication of the output tensor dimensions: height, width and channel dimension. These conditions are evaluated each time a tile is processed. If all conditions are met, the tiling algorithm can generate a feasible tile for the accelerator. If not, the memory constraints need to be relaxed.

Tiling Algorithm Steps:

- Tiling the Output Channel Dimension: The process begins by dividing the output channel dimension of both the weight and output tensors. To maximize the accelerator's performance, the number of output channels per tile matches the number of PEs in the accelerator. If the number of PEs exceeds the number of output channels, no tiling is performed. This allows the accelerator to fully utilize its internal resources and process data in parallel. Fig. 3.4 shows a weight tensor with two output channels represented as two cubes.

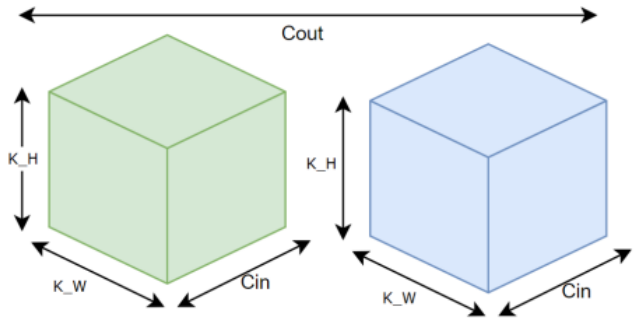


**Figure 3.4:** Tiling of the weight tensor across output channel dimension. Source [2].

- Tiling the Input Channel Dimension: The next step is to tile the input channel dimension of both the input and weight tensors. In this step, the tensors are recursively divided by two until they meet the constraints defined in Inequalities 3.1 and 3.2, or until a minimum input channel value is reached. This minimum value varies based on the precision configuration of the accelerator, which determines how many input channels can be processed simultaneously. For example, in a low precision setting (i.e., 4-bit for activations and weights), the accelerator processes up to 4 different values from distinct input channels, so the minimum input channel value is 4. For medium precision (i.e., 8-bit for activations and weights), the value is 2, and for full precision (i.e., 16-bit for activations and weights), the value is 1, as the accelerator processes one value at a time from the input channels. Fig. 3.5 illustrates a tensor that has been tiled along the input channel dimension, with the tensor divided in half.

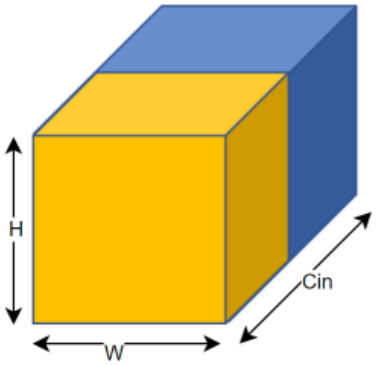


**Figure 3.5:** Tiling of the input/weight tensor across input channel dimension. Source [2].

- Tiling the Height Dimension: If the tiled tensor still does not fit in memory, the next step involves tiling the input and output tensors along the height dimension. To simplify the handling of data replication between adjacent tiles, a fixed-height length is used. This height is set to match the height of the kernel, which avoids the complexity of calculating a compatible tile dimension with the size of the kernel and the stride used in convolution operation. This approach also ensures that no portion of the tensor is unnecessarily included due to tile size mismatches with the kernel or stride. Tiling across the width dimension is avoided to prevent data replication in adjacent tiles, which can have some part of the input data in one and the rest of the data in the next tile. Fig. 3.6 illustrates a tensor that has been tiled along the height dimension, with the tile height matching the kernel height.

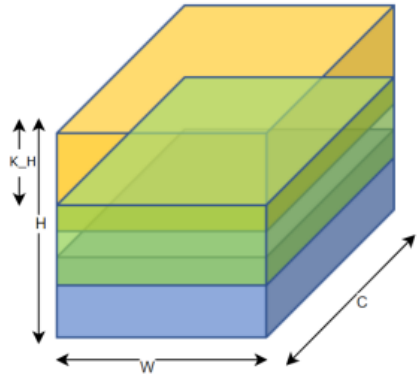


**Figure 3.6:** Tiling of the input tensor across the height dimension. Source [2].

- Additional Tiling of the Output Channel Dimension: If memory constraints still aren't met at this stage, the final step is to tile the output channel dimension once more. In this case, performance is sacrificed in favor of functionality. The tile is divided by two, conditions are rechecked, and this process is repeated until only one output channel remains.

Once the tile sizes are determined, the next step is to decide the order in which the tiles will be processed to optimize system performance and minimize memory transfers when partial results are generated. Additionally, the addressing of tiles must be managed to ensure correct data access and that the output is stored in the appropriate locations. Fig. 3.7 illustrates how tensor data is stored in the external memory for the proposed architecture. The figure represents an example of a tensor with the following dimensions: 4x2x2x2 (Width - Height - Input Channels - Output Channels). The values are packed first considering the width dimension or the rows, then each row is packed considering the height dimension. Next, the input

channel dimension is grouped, placing the tensor Width-Height-Input_Channel consecutively. Finally, the outermost index represents the output channels, where the last packing is done. This is the most complex case, typically involving a weight tensor. For input and output tensors, only the first three dimensions are considered, though the packing order remains the same.

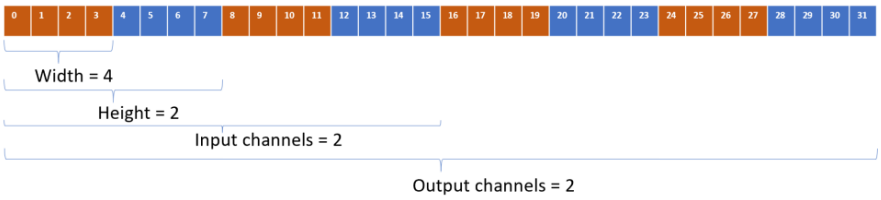


**Figure 3.7:** Tensor data organization in external memory. Source [2].

Fig. 3.8 depicts the external memory organization for the input, weight and output tensors. To address each tensor, three pointers are used, each pointing to the start of its respective memory region. These pointers serve as references when a new tile needs to be addressed. The software responsible for invoking the accelerator to perform the convolution operation must also calculate the pointer offsets to correctly address the corresponding tiles. Consequently, the three pointers are continuously updated with the calculated offsets. Additionally, the tile sizes are verified in case there is an uneven division during the final iteration.

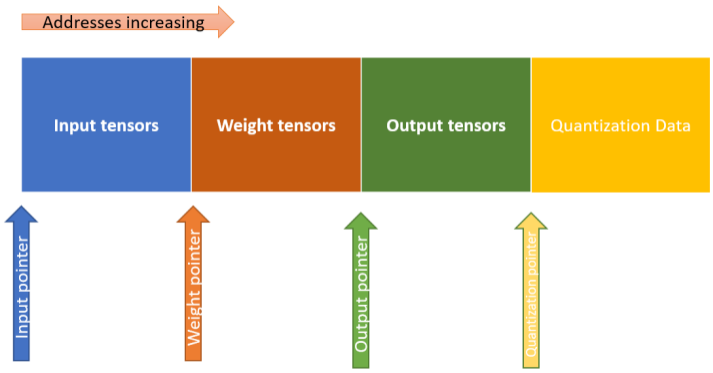


**Figure 3.8:** Organization of the tensors in external memory. Source [2].

The tiling algorithm has been implemented in C and can be executed offline, as the sizes of the accelerator's PLMs are known beforehand. This means the tile sizes can be calculated prior to the inference process, reducing computation time for the processor.

### 3.2.2 Hardware Implementation: Sequential Architecture

This section explains the hardware implementation of the accelerator used in the tiling architecture, obtained referencing to previous works [13] [14]. Fig. 3.9 illustrates the ESP accelerator interface and the signals required to integrate the accelerator into the SoC. The accelerators operate in four main phases: configuration, load, compute and store. The thesis of D. R. Bueno Pacheco [2] considers only the sequential architecture of the accelerator, also called single-block architecture, where all these four phases are executed sequentially and are implemented in a single C++ function [15]. Instead, a hierarchical architecture where the configure, load, compute, store phases are coded as separated C++ functions that can run in parallel [15], can also be implemented and this is the main subject of this work. Fig. 3.10, that has been obtained adapting to our case a figure from [13], shows the sequential architecture of a 2D-Convolution accelerator.

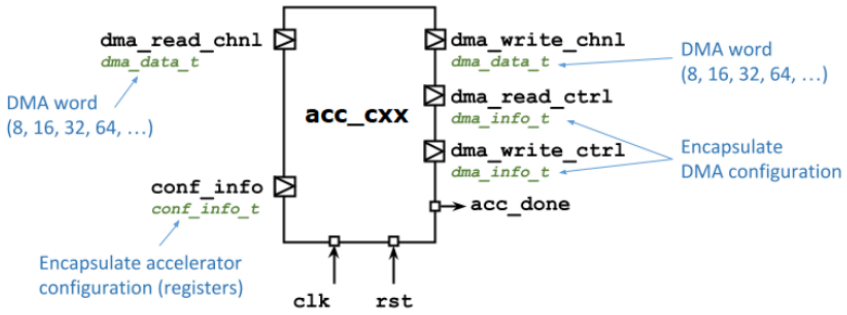


**Figure 3.9:** Interface for a generic ESP accelerator. Source [15].

The accelerator implemented is the PS accelerator proposed in [10]. As mentioned in Sec. 3.2.1, it contains multiple PEs, each working independently on a different output channel. Each PE consists of PSMAC units based on ST multipliers, enabling configurable precision for convolution computations. Additionally, the accelerator supports output data quantization for three different bit-widths: 4, 8 and 16 bits.

**Configuration Phase**: During this phase, the configuration parameters stored in the memory-mapped registers, set by the processor before starting the accelerator, are retrieved by the accelerator via the `conf_info` port. According to ESP accelerator specifications, the maximum number of user-defined registers is 14 [16]. Since the proposed architecture requires at least 20 parameters, some of them have been packed into a single register when written by the processor, and later unpacked using masks and bitwise or and operations when read by the accelerator. Once the parameters are unpacked and read, they are passed to the Load phase.

**Load Phase**: In this phase, the input and weight data of the corresponding

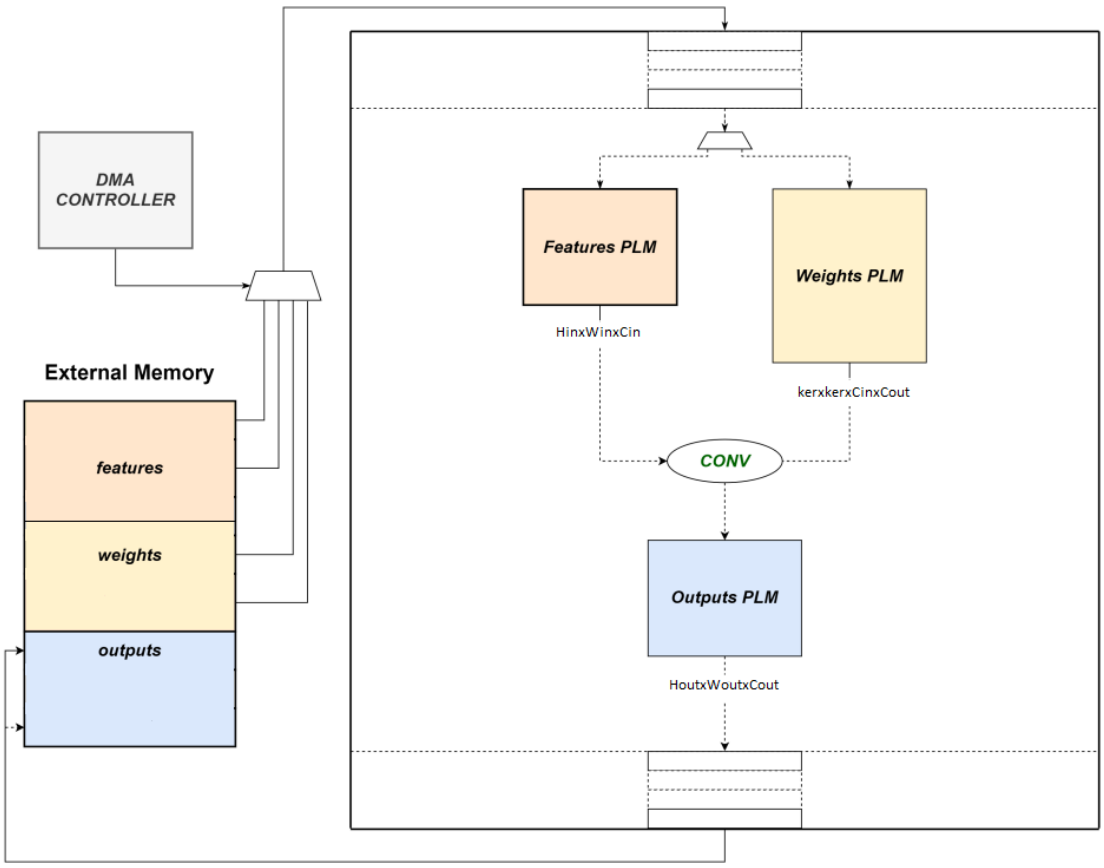


**Figure 3.10:** 2D-convolution Accelerator: sequential architecture. Source [13].

tile are transferred from external memory to the accelerator's PLMs. Then, the parameters required to perform the quantization of the results of the convolution are read and stored in their respective variables and they are ready to be used for the computation phase. These data transfers are handled by the DMA engine already provided by ESP. Once the data is loaded into the PLMs, it must be arranged to meet the requirements of the computation unit. This unit requires the data to be split into four groups, each responsible for handling 4 bits of one MAC operand. Depending on the precision configuration selected for the ST multiplier, the operands can be formed in the following ways:

- One 16-bit value from a single input channel.
- Two 8-bit values from two input channels.
- Four 4-bit values from four input channels.

The packing for the weight data is similar and follows [11].

**Computation Phase**: This phase is essentially the version of the accelerator developed in [10] and [11] which has been adapted to the ESP accelerators guidelines [15]. For a more in-depth understanding of the internal architecture, it is recommended to refer to the original papers.

**Store Phase**: At this stage, all the computed and quantized values are stored in the output PLM and transferred sequentially to the external memory through the DMA.

## 3.3 Results

In [2], the correctness of the tiling algorithm described above has been tested through both C and RTL simulations, followed by further evaluation on an FPGA using the ESP framework. Additionally, the performances of hardware accelerators using the tiling algorithm have been compared with the performances of the RISC-V processor in computing convolution operations. The results show that the accelerator performs better then the RISC-V processor bare-metal code when the number of tiles is low and the tile sizes are large. In such cases, the accelerator better utilizes each DMA transaction, transferring more data in each transaction while spending less time on configuring the accelerator or transferring data with the DMA. The findings suggest that larger PLMs are preferable, as they allow for fewer and larger tiles. Conversely, when the number of tiles increases and tile sizes decrease, the tiling and DMA transaction overhead becomes more significant, negatively impacting the overall performance of both the accelerator and the tiling algorithm.

We will observe how the implementation of a hierarchical design (with pipelined config, load, compute and store phases using dataflow directives) will improve performances.

# Chapter 4

# 2D-Convolution Accelerator: Hierarchical Architecture

In this chapter, the principles underlying the quantization algorithm applied to the output values will be described. Then, the implementation of the hierarchical architecture of the ST based 2D-Convolution accelerator, obtained starting from the sequential architecture implemented in the previous thesis work [2] and reported in Ch. 3, will be explained and analyzed.

## 4.1 Output Quantization

The algorithm that performs the quantization on the output values of the accelerator was already present in the base structure of the accelerator code from which D. R. Bueno Pacheco started his thesis work [2]. In this section the explanation about the quantization technique implemented in the accelerator is reported and it is taken from the reference [11].

Quantization of deep neural networks (DNNs) has become a widely adopted practice that reduces the numerical precision of weight parameters and activation values within neural network layers. This technique minimizes the model's size and lowers memory requirements by allowing multiple low-precision feature maps and weights to be efficiently stored within a single memory word. Consequently, it also cuts down on data transfer costs. Furthermore, quantization can enhance inference speed, throughput and energy efficiency by leveraging dedicated hardware like ST multipliers.

The focus is on *Uniform Integer Quantization* (*UIQ*) [11], although numerous other quantization methods exist [17]. This choice is guided by UIQ's straightforward mathematical formulation and its availability in popular ML frameworks

(e.g., TensorFlow Lite).

The mathematical foundations of UIQ in the context of DNNs are now introduced. It is important to note that, since the targets are ST-based accelerators for DNN inference only, the focus is on UIQ for inference rather than training.

### 4.1.1 Uniform Integer Quantization UIQ

Given a set of real numbers within the *real range* $[\alpha, \beta]$, UIQ maps each $x \in [\alpha, \beta]$ to an integer value $x_q \in [\alpha_q, \beta_q]$ uniformly represented using $b$ bits. The *quantized range* $[\alpha_q, \beta_q]$ depends on the type of integer representation: for asymmetric or symmetric signed integers, it is either $[-2^{b-1}, 2^{b-1} - 1]$ or $[-2^{b-1} - 1, 2^{b-1} - 1]$, respectively, while for unsigned integers, the range is $[0, 2^b - 1]$. The quantization process is described by the following equation:

$$x_q = \text{clip}\Big(\text{round}\big(\frac{1}{s}x + z\big), \alpha_q, \beta_q\Big) \tag{4.1}$$

where:

- $s$ is the *scaling factor*.
- $z$ is the *zero-point* (i.e., the integer value that exactly represents the real number zero).
- *round* is the rounding function (e.g., round-to-nearest).
- *clip* ensures that the output remains within the quantized range by saturating values that fall outside the range.

The scaling factor $s$ and the zero-point $z$ are derived from the real and quantized ranges as follows:

$$s = \frac{\beta - \alpha}{\beta_q - \alpha_q} \tag{4.2}$$

$$z = \text{round}\big(\frac{\beta\alpha_q - \alpha\beta_q}{\beta - \alpha}\big) \tag{4.3}$$

The reverse operation, which maps $x_q$ back to the real range, is defined as:

$$\hat{x} = s(x_q - z) \tag{4.4}$$

where $\hat{x}$ represents the closest real value to the original $x$, though it may not be identical due to the rounding and clipping steps, which can introduce irreversible errors.

The quantization mapping described so far, where the ranges are asymmetric and $z \neq 0$, is known as affine quantization. On the other hand, when both ranges are symmetric (i.e., $z$ becomes zero) the equation simplifies to a pure scale transformation. This form of quantization mapping is commonly referred to as *scale* or *symmetric* quantization.

Furthermore, when the scaling factor $s$ is the same across all channels of a tensor, it is called *per-layer* quantization. However, when $s$ is a one-dimensional vector with a distinct scalar value for each channel, this method is known as *per-channel* quantization.

### 4.1.2 Integer-Only DNN Kernels

The expression of a Fully Connected (FC) layer is the following:

$$Y_k = b_k + \sum_{c=1}^{C} X_c W_{c,k} \qquad \forall\, k \in [1, K] \tag{4.5}$$

where $X \in \mathbb{R}^C$ represents the input vector of neurons, $W \in \mathbb{R}^{K \times C}$ represents the weight matrix, $b \in \mathbb{R}^K$ represents the bias array, $Y \in \mathbb{R}^K$ represents the output array, $C$ and $K$ represent the number of input and output activations processed by the FC layer, respectively.

If the equation 4.4 is applied to each of the four real variables in 4.5, determining their quantized ranges in advance, it is possible to obtain the quantized FC expression valid for the $k$-th output activation:

$$\begin{aligned} Y_{q,k} = \underbrace{z_Y}_{\text{(a)}} + \underbrace{\frac{s_b}{s_Y}(b_{q,k} - z_b)}_{\text{(b)}} + \frac{s_X s_W}{s_Y} \Bigg[ \underbrace{\left( \sum_{c=1}^{C} X_{q,c} W_{q,c,k} \right)}_{\text{(c)}} \\ - \underbrace{\left( z_W \sum_{c=1}^{C} X_{q,c} \right)}_{\text{(d)}} - \underbrace{\left( z_X \sum_{c=1}^{C} W_{q,c,k} \right)}_{\text{(e)}} + \underbrace{C z_X z_W}_{\text{(f)}} \Bigg] \quad \forall\, k \in [1, K] \end{aligned} \tag{4.6}$$

where $X_q$, $W_q$, $b_q$, $Y_q$ represent the integer values; $s_X$, $s_W$, $s_b$, $s_Y$ represent the scaling factors; and $z_X$, $z_W$, $z_b$, $z_Y$ represent the zero-points, associated with $X$, $W$, $b$, $Y$, respectively. Term (c) represents the core of the computation, the integer dot product, while term (d) introduces additional overhead, resulting in a performance penalty. Both must be computed online as they depend on $X_q$, which is only known at runtime. On the other hand, terms (a), (b), (e), and (f) are constant and can be computed offline. When scale quantization is applied to the weights and affine quantization is used for the activations (a common practice in literature), both

$z_W$ and $z_b$ become zero, and so also terms (d) and (f), while (b) simplifies. This assumption holds true for this work as well. Before assigning the result of the expression to $Y_q$, the value is rounded and clipped to ensure it fits within the desired quantized output range for $Y_q$, though this step is not shown here for readability.

The integer-only formulations for 2D-Convolution closely follow the FC layer derivation [11].

### 4.1.3 ST Based 2D-Conv Accelerator Design: UIQ Variables

To meet a hypothetical requirement of achieving zero computational errors in UIQ formulas, it would be necessary to use mathematical operators (such as multipliers and adders) with extremely large bit widths. This is due to the precision propagation through the various operations involved. However, this would lead to impractically large accelerators or even prevent the HLS tool from generating feasible designs. Therefore, to optimize the hardware accelerators, the bitwidth of the C/C++ variables used in the UIQ formulas have been reduced until there is not a performance loss greater than a certain threshold.

The UIQ formula 4.6 for the FC layer, with $z_W = 0$ and $z_b = 0$, is considered as reference. The same logic can be applied to the UIQ formulas for other accelerators. From the decomposition of (4.6) in (4.7)–(4.10) the intermediate results $v1_{q,k}$, $v2_{q,k}$, $v3_{q,k}$ are obtained and their expressions are reported below:

$$v1_{q,k} = \left[ \sum_{c=1}^{C} X_{q,c} W_{q,c,k} - z_X \sum_{c=1}^{C} W_{q,c,k} \right] \tag{4.7}$$

$$v2_{q,k} = s_X s_W \cdot v1_{q,k} \tag{4.8}$$

$$v3_{q,k} = s_b b_{q,k} + v2_{q,k} \tag{4.9}$$

$$Y_{q,k} = clip(round(z_Y + s_Y^{-1}\ v3_{q,k}), \alpha_q, \beta_q) \tag{4.10}$$

where $Y_{q,k}$ represents the $k$-th output element, with $k \in [1, K]$, quantized on INT$y$ bits ($y = 16$, 8, or 4) on the integer quantized range $[\alpha_q, \beta_q] = [-2^{b_y-1} + 1, 2^{b_y-1} - 1]$, and all other variables are those introduced with (4.6) in Sec. 4.1.1. In the ST based 2D-Conv accelerator implemented in this thesis, each of these variables is either a fixed-point or an integer number. In particular, five variables are identified and used in the output quantization algorithm in the accelerator:

- **weights crossproduct**

$$w\_cross = z_X \sum_{c=1}^{C} W_{q,c,k} \tag{4.11}$$

- **scaling factor of inputs and weights**

$$scaling\_factor\_iw = s_X s_W \tag{4.12}$$

- **bias quantized scaled**

$$biasq\_scaled = s_b b_{q,k} \tag{4.13}$$

- **zero point of the output**

$$z\_o = z_Y \tag{4.14}$$

- **scaling factor of the output, inverse**

$$scaling\_factor\_oi = s_Y^{-1} \tag{4.15}$$

## 4.2 Hierarchical Architecture

In this section, that is the core of this thesis work, the high-level description (C++ implementation) of the hierarchical architecture of the accelerator will be explained and analyzed. In particular, the thesis work [2], resumed in Ch. 3, will be used as starting point, since it shows the developing of the sequential architecture of the same accelerator, while the thesis works [13] and [14] will be used as references since the explain from the very basic steps the workflow followed to implement a CNN accelerator in ESP. Also, the ESP documentation [15] will be used as guideline.

### 4.2.1 Accelerator Interface

The ESP accelerator interface and the signals required to integrate the accelerator into the SoC are the one already shown in Fig. 3.9. The picture is reported again below, since the interface is now analyzed more thoroughly.

The interface of an ESP accelerator consists of common ports across all supported HLS flows, with minor syntax variations depending on the tool used.

These ports allow the accelerator to:

- Communicate with the CPU via memory-mapped registers (`conf_info`).
- Program the DMA controller (`dma_read_ctrl` and `dma_write_ctrl`).
- Transfer data to/from the main memory (`dma_read_chnl` and `dma_write_chnl`).
- Notify task completion back to the software application (`acc_done`).

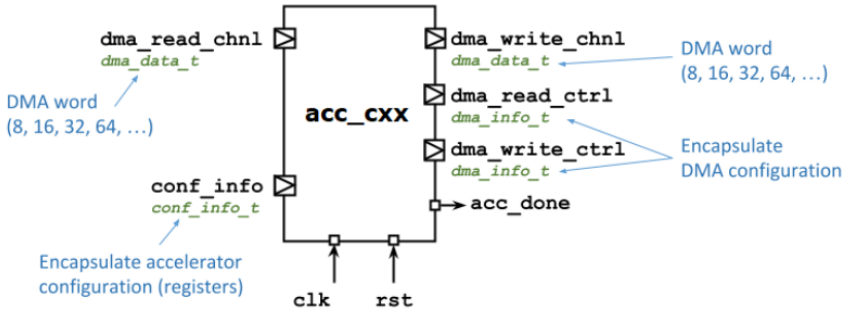


**Figure 4.1:** Interface for a generic ESP accelerator. Source [15].

**Listing 4.1:** `conv2d.cpp` - Accelerator Interface

```
void CCS_BLOCK(conv2d_cxx_catapult)(
    ac_channel<conf_info_t> &conf_info,
    ac_channel<dma_info_t> &dma_read_ctrl,
    ac_channel<dma_info_t> &dma_write_ctrl,
    ac_channel<dma_data_t> &dma_read_chnl,
    ac_channel<dma_data_t> &dma_write_chnl,
    ac_sync &acc_done)
```

The interface uses:

- `ac_channel`: Communication channels to transfer data between the top module and the other components.
- `ac_sync`: Synchronization channel, which Catapult HLS provides for specifying standalone handshaking signals when a designer needs to control synchronization directly.

In addition to the communication and synchronization channels, the accelerator interface in ESP relies on specific data types:

- Configuration Registers: The configuration data is stored in memory-mapped registers:

**Listing 4.2:** `conf_info.hpp` - Struct for configuration parameters

```
struct conf_info_t {
    uint32_t options;
    uint32_t offset_PE;
    uint32_t offset_PE_out;
    uint32_t offset_q_data;
    uint32_t pad_stride_kern;
    uint32_t filt;
    uint32_t offset_read_ci;
    uint32_t n_c;
    uint32_t n_h;
    uint32_t n_w;
    uint32_t in_add;
```

```
13      uint32_t w_add;
14      uint32_t out_add;
15      uint32_t flags;
16  };
```

This `conf_info_t` structure is used to define the configuration registers. As already explained in Sec. 3.2.2, all the 14 user-defined registers are used, inside which 20 parameters are packed. The meaning of these configuration parameters is the following:

- `options`: This configuration register contains two parameters: `CONFIG1` Configuration variable that specifies the precision (4, 8, or 16 bit) set for the multiply and accumulate operations performed by the ST multipliers; `CONFIG2` Configuration variable that specifies the precision (4, 8, or 16 bit) set for the output quantization
- `offset_PE`: Offset value used to read inter-spaced weight values from different output channels.
- `offset_PE_out`: Offset value used to write inter-spaced output values to different output channels.
- `offset_q_data`: Pointer or offset to the output quantization variables.
- `pad_stride_kern`: This configuration register contains four parameters: `pad` Number of pixels to apply the padding operation; `pad_type` Configuration variable that indicates where to apply padding around the tensor; `stride` Number of positions the kernel slides; `kern` Width and Height size of the weight tile (since most of the times a symetric kernel is used, these dimensions are cosidered the same).
- `filt`: Output channel size of the output and weight tile.
- `offset_read_ci`: Offset value used to read inter-spaced input values from different input channels.
- `n_c`: Input channel size of the input and weight tile.
- `n_h`: Height size of the input tile.
- `n_w`: Width size of the input tile.
- `in_add`: Pointer or offset to the input tile.
- `w_add`: Pointer or offset to the weight tile.
- `out_add`: Pointer or offset to the output tile.
- `flags`: This configuration register contains three parameters: `EN_QUANTIZATION` Flag that enables the output quantization operation; `RST_OUT_ACC` Flag that enables accumulation of partial results; `EN_RELU` Flag that enables Relu operation of the output results.

- DMA Configuration: DMA settings are defined by:

**Listing 4.3:** Struct for DMA configuration

```
struct dma_info_t {
    uint32_t index;
    uint32_t length;
    ac_int<3, false> size;
};
```

  - `index`: The memory offset for the DMA transaction.
  - `length`: The length of the DMA transaction.
  - `size`: Specifies the width of the DMA word, encoded as follows: 0 for an 8-bit word, 1 for 16 bits, 2 for 32 bits, etc.

  This `dma_info_t` struct is common to all accelerators and designers can not modify it.

- DMA Word: The DMA word is defined using:

**Listing 4.4:** `conv2d.hpp` - Struct for DMA configuration

```
typedef ac_int<DMA_WIDTH, false> dma_data_t;
```

  `DMA_WIDTH` represents the width of the DMA word, which in our case is 64.

### 4.2.2 Top-Level Function

The accelerator's execution is divided into four key phases: configuration, load, compute and store. As already said, in the sequential architecture implemented in the previous thesis work [2], these phases are executed sequentially and are coded in a single C++ function. Conversely, in a hierarchical-block architecture the ESP-accelerator phases are mapped onto blocks that can run concurrently. This concurrency is achieved by applying HLS constraints (they will be analyzed in Sec. 5.2) along with a recommended coding style.

Key characteristics of the hierarchical-block architecture, shown in Fig. 4.2, include:

- The configure, load, compute, and store phases are implemented as separate C++ functions.
- Private local memories (PLMs) are globally defined and shared across these C++ functions.

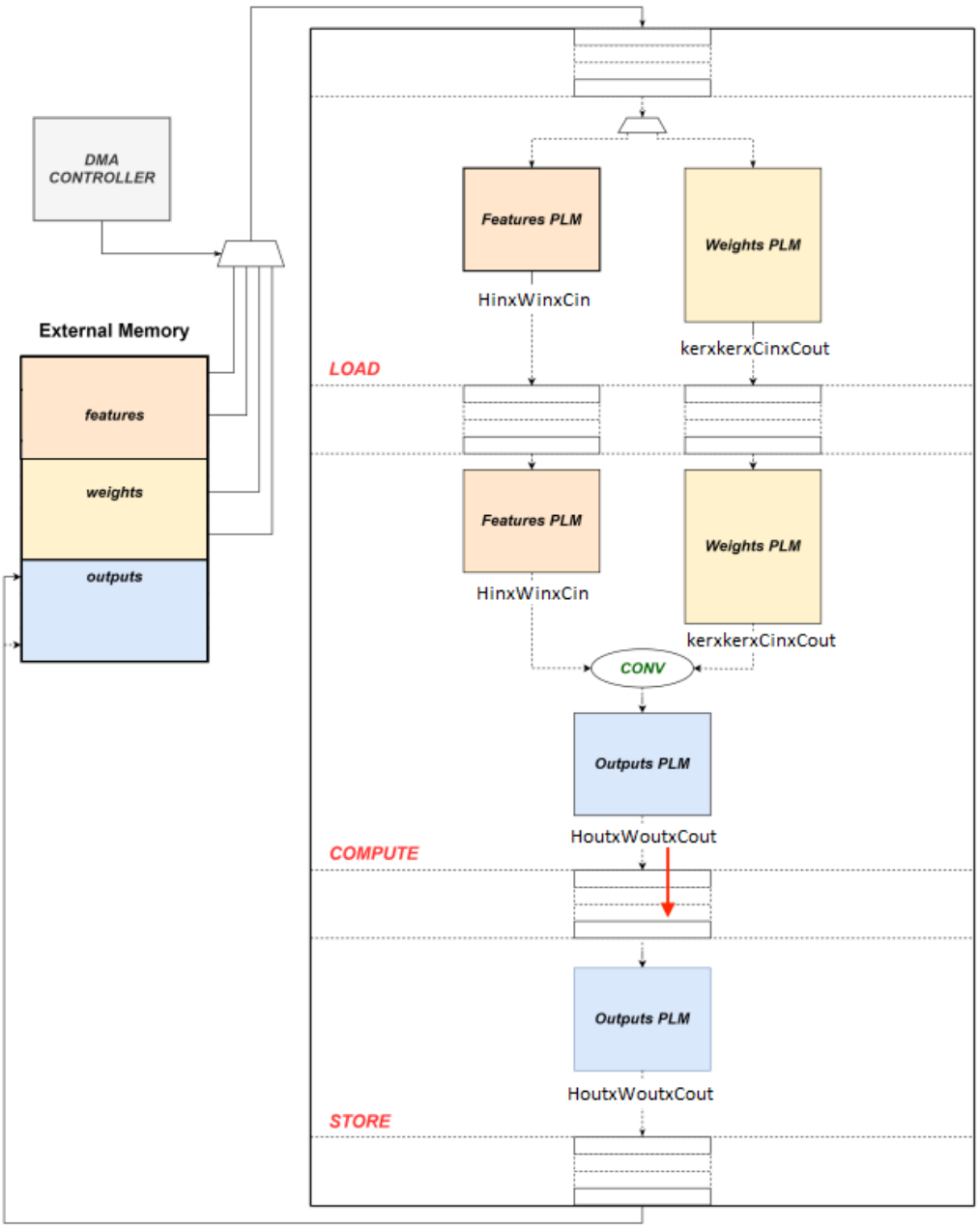


**Figure 4.2:** 2D-convolution Accelerator: hierarchical architecture. Source [13].

In standard C++ programming, functions such as load, compute and store typically exchange data using shared arrays or variables. However, this approach poses challenges for HLS, as synchronization for data exchange between hierarchical blocks must be automatically inserted by the synthesis tool. To address this

complexity, Catapult HLS provides the `ac_channel` modeling construct, enabling users to model data exchange between blocks of the hierarchy more effectively.

In the top-level function of the 2D-Conv accelerator, named `conv2d_cxx_catapult`, Lst. 4.5, it is possible to observe that:

- inputs, weights and outputs PLMs are declared and shared among functions using `ac_channel`.
- Configuration information and output quantization variables are also exchanged via `ac_channel`.
- Synchronization signals are handled using `ac_sync`.

**Listing 4.5:** `conv2d.cpp` - Top-level function

```
#pragma hls_design top
void CCS_BLOCK(conv2d_cxx_catapult)(
    ac_channel<conf_info_t> &conf_info,
    ac_channel<dma_info_t> &dma_read_ctrl,
    ac_channel<dma_info_t> &dma_write_ctrl,
    ac_channel<dma_data_t> &dma_read_chnl,
    ac_channel<dma_data_t> &dma_write_chnl,
    ac_sync &acc_done) {

    static ac_channel<packed_weights_t> packed_weights;
    static ac_channel<packed_inputs_t> packed_inputs;
    static ac_channel<plm_outputs_t> plm_outputs;

    static ac_channel<conf_info_t> plm_conf_load;
    static ac_channel<conf_info_t> plm_conf_compute;
    static ac_channel<conf_info_t> plm_conf_store;

    static ac_channel<parameters_t> parameters;

    static ac_sync config_done;
    static ac_sync load_done;
    static ac_sync compute_done;
    static ac_sync store_done;

    config(conf_info, plm_conf_load, plm_conf_compute, plm_conf_store, config_done
    );
    load(plm_conf_load, packed_inputs, packed_weights, parameters, dma_read_ctrl,
    dma_read_chnl, load_done);
    conv2d_m4_v10_reconf_reducedbitwidth(packed_inputs, packed_weights,
    plm_outputs, plm_conf_compute, parameters, compute_done);
    store(plm_conf_store, plm_outputs, dma_write_ctrl, dma_write_chnl, store_done)
    ;

    config_done.sync_in();
    load_done.sync_in();
    compute_done.sync_in();
    store_done.sync_in();

    acc_done.sync_out();
}
```

When dealing with large arrays, it is common practice to map them to memories during synthesis, as mapping to registers can become too costly in terms of area and power consumption. In Catapult HLS, mapping shared arrays between different blocks in a hierarchical design automatically infers a ping-pong memory structure. This structure typically involves two or more memories, enabling them to be written to and read from in a sequence that allows the blocks to operate concurrently. The templated structure `plm_t` is used to map shared arrays to memories. It also simplifies the declaration of arrays and matrices and defines an array with S elements of a specified T data type. This structure is used for representing the PLMs where input features, weights and outputs are stored, as demonstrated in Lst. 4.6 and Lst. 4.7.

**Listing 4.6:** `conv2d.hpp` - Templated structures for PLMs, definition of maximum dimensions of the 2D-Conv layers supportd by the accelerator, definition of the PLMs types

```
template <class T, unsigned S>
struct plm_t {
public:
   T data[S];
};

#define N_H_IN_MAX 18
#define N_W_IN_MAX N_H_IN_MAX
#define N_C_MAX 32
#define KERN_MAX 7
#define FILT_MAX 8
#define STRIDE_MAX 2
#define PAD_MAX 3
#define N_W_OUT_MAX N_W_IN_MAX
#define N_H_OUT_MAX N_H_IN_MAX

#define FILTERS_SIZE_MAX    KERN_MAX * KERN_MAX * N_C_MAX * FILT_MAX
#define INPUTS_SIZE_MAX     N_W_IN_MAX * N_H_IN_MAX * N_C_MAX
#define OUTPUTS_SIZE_MAX    N_W_OUT_MAX * N_H_OUT_MAX * FILT_MAX

typedef plm_t<FPDATA_OUT, OUTPUTS_SIZE_MAX> plm_outputs_t;
typedef plm_t<FPDATA_packed, INPUTS_SIZE_MAX> packed_inputs_t;
typedef plm_t<FPDATA_packed, FILTERS_SIZE_MAX> packed_weights_t;
typedef plm_t<FPDATA_IN, INPUTS_SIZE_MAX> plm_inputs_t;
typedef plm_t<FPDATA_IN, FILTERS_SIZE_MAX> plm_filters_t;
```

**Listing 4.7:** `fpdata.hpp` - Definition of the inputs, weights and outputs types stored in the PLMs

```
typedef ac_int <OUT_BITWIDTH_16x, true> FPDATA_packed;
typedef ac_int <OUT_BITWIDTH_32x, true> FPDATA_IN;
typedef ac_int <OUT_BITWIDTH_32x, true> FPDATA_OUT;
```

Furthermore, a particular coding style must be adhered in order to prevent unintentional inference of additional memories: a local instance of the templated struct on which memory operations can be performed must be declared inside every

function. This coding style, will be shown and analyzed more in details in Sec. 4.2.4 and Sec. 4.2.6, the sections related to the load and store functions.

The output quantization variables usually consist in shorter arrays, thus they do not need to be mapped to memories and a normal struct, `parameters_t`, shown in Lst. 4.8, is used to encapsulate them. In this way, they are mapped to simple registers during synthesis. The bitwidth of these parameters are defined in `fpdata.hpp` and have been determined by the studies conducted in [11]. Also in this case, the same coding style that was discussed in the previous paragraph must be adhered to prevent unintentional inference of additional registers.

**Listing 4.8:** `conv2d.hpp` - Structure for ouput quantization variables

```
1 struct parameters_t {
2     ac_int<W_CROSS_BITWIDTH, true> WEIGHTS_CROSSPRODUCT[FILT_MAX];
3     ac_fixed<SF_IN_W_TOT_BITWIDTH, SF_IN_W_INT_BITWIDTH, true>
       SCALING_FACTOR_INPUTS_WEIGHTS[FILT_MAX];
4     ac_fixed<BIASQ_SCALED_TOT_BITWIDTH, BIASQ_SCALED_INT_BITWIDTH, true>
       BIASQ_SCALED[FILT_MAX];
5     ac_fixed<SF_OUT_INV_TOT_BITWIDTH, SF_OUT_INV_INT_BITWIDTH, true>
       SCALING_FACTOR_OUT_INVERSE;
6     ac_int<Z_BITWIDTH, true> Z_O2;
7 };
```

### 4.2.3 Configure Function

The configure function is shown in Lst. 4.9. Its interface consists of the following ports:

- `conf_info` port: It allows the function to receive the configuration parameters from the main function.
- `plm_conf_load`, `plm_conf_compute` and `plm_conf_store` ports: Through them the configuration parameters are transmitted to the load, compute and store functions.
- `ac_done` port: Used to notify task completion to the other functions and synchronize with them.

Inside the function, the configuration parameters stored in the memory-mapped registers, set by the processor, are retrieved via the `conf_info` port. The information transmitted through this `ac_channel` is then organized and stored in the appropriate `conf_info_t` local structure, `params`. The content of `params` is then assigned to the `conf_info_t` local structures `conf_info_load_tmp`, `conf_info_compute_tmp` and `conf_info_store_tmp`. Finally, the the data inside these last three strutures is written inside the three output channels, `plm_conf_load`, `plm_conf_compute` and `plm_conf_store`.

**Listing 4.9:** `conv2d.cpp` - Configure function

```
void config(
        ac_channel<conf_info_t> &conf_info,
        ac_channel<conf_info_t> &plm_conf_load,
        ac_channel<conf_info_t> &plm_conf_compute,
        ac_channel<conf_info_t> &plm_conf_store,
        ac_sync &done) {

    struct conf_info_t params;

    // Read accelerator configuration
#ifndef __SYNTHESIS__
    while (!conf_info.available(1)) {}
    // Hardware stalls until data ready
#endif

    params = conf_info.read();

    conf_info_t conf_info_load_tmp;
    conf_info_t conf_info_compute_tmp;
    conf_info_t conf_info_store_tmp;

    conf_info_load_tmp = params;
    conf_info_compute_tmp = params;
    conf_info_store_tmp = params;

    plm_conf_load.write(conf_info_load_tmp);
    plm_conf_compute.write(conf_info_compute_tmp);
    plm_conf_store.write(conf_info_store_tmp);

    done.sync_out();
}
```

### 4.2.4 Load Function

The load function interface, shown in Lst. 4.10, consists of the following ports:

- `conf_info` port: It allows the function to receive the configuration parameters from the configure function.
- `packed_inputs`, `packed_weights` and `parameters` ports: Through them the input features, the weights and the output quantization parameters are transmitted to the compute function.
- `dma_read_ctrl`: Used to program the DMA controller.
- `dma_read_chnl`: Used to receive data from the main memory.
- `done` port: Used to notify task completion to the other functions and synchronize with them.

**Listing 4.10:** `conv2d.cpp` - Load function interface

```
void load(
        ac_channel<conf_info_t> &conf_info,
        ac_channel<packed_inputs_t> &packed_inputs,
        ac_channel<packed_weights_t> &packed_weights,
        ac_channel<parameters_t> &parameters,
        ac_channel<dma_info_t> &dma_read_ctrl,
        ac_channel<dma_data_t> &dma_read_chnl,
        ac_sync &done)
```

In the first part of the function, Lst. 4.11, the configuration parameters stored in the memory-mapped registers, set by the processor, are retrieved via the `conf_info` port. The information transmitted through this `ac_channel` is then organized and stored in the appropriate `conf_info_t` local structure, `params`. Finally, every parameter is stored inside a proper variable, using masks and and operations when multiple parameters are packed inside one single field of the structure.

**Listing 4.11:** `conv2d.cpp` - Load function, reading of the configuration parameters

```
    // Read accelerator configuration
#ifndef __SYNTHESIS__
    while (!conf_info.available(1)) {} // Hardware stalls until data ready
#endif
    params = conf_info.read();

    in_add = params.in_add;
    w_add = params.w_add;
    out_add = params.out_add;
    acc_flag = (params.flags >>1) & 0x00000001;
    q_flag = (params.flags) & 0x00000001;
    relu_flag = (params.flags >>2) & 0x00000001;
    n_w  = params.n_w;
    n_h = params.n_h;
    n_c = params.n_c;
    kern = ((params.pad_stride_kern) & 0x0000F000)>>12;
    filt = params.filt;
    pad = ((params.pad_stride_kern) & 0x0000000F);
    pad_type = ((params.pad_stride_kern) & 0x00000F00)>>8;
    stride = ((params.pad_stride_kern) & 0x000000F0)>>4;
    offset_PE_out = params.offset_PE_out;
    offset_PE = params.offset_PE;
    CONFIG1 = ((params.options) & 0x0000000F);
    CONFIG2 = ((params.options) >> 4;
    offset_q_data = params.offset_q_data;
    offset_read_ci = params.offset_read_ci;
```

After that phase a few auxiliary variables are defined to manage the loop boundaries. The variables `n_w_in` and `n_h_in` represent the width and height dimensions of the input tile after determining the appropriate padding to be applied. The following code snip shows this process.

**Listing 4.12:** `conv2d.cpp` - Load function, definition of the auxiliary variables

```
    // Padded Input Dimensions
    switch (pad_type)
    {
```

```
4       case 0: // no padding
5           n_w_in = n_w ;
6           n_h_in = n_h ;
7           break;
8       case 1: // padding 3 sides
9           n_w_in = n_w + 2 * pad;
10          n_h_in = n_h + pad;
11          break;
12      case 2: // padding 3 sides
13          n_w_in = n_w + 2 * pad;
14          n_h_in = n_h + pad;
15          break;
16      case 3: // padding 2 sides
17          n_w_in = n_w + 2 * pad;
18          n_h_in = n_h;
19          break;
20      case 4: // padding 4 sides
21          n_w_in = n_w + 2 * pad;
22          n_h_in = n_h + 2 * pad;
23          break;
24      default:
25          n_w_in = n_w ;
26          n_h_in = n_h ;
27          break;
28      }
```

As already explained in Sec. 4.2.2, in order to prevent unintentional inference of additional memories or registers to which the shared arrays are mapped, local structs containing the packed arrays must be declared inside every function to handle memory operations. For this reason, when transferring data from the external environment via DMA to the shared register, `parameters`, a local instance of the `parameters_t` struct (`parameters_tmp`) and local variables to store the output quantization parameters (`WEIGHTS_CROSSPRODUCT_tmp`, `SCALING_FACTOR_INPUTS_WEIGHTS_tmp`, `BIASQ_SCALED_tmp`, `SCALING_FACTOR_OUT_INVERSE_tmp` and `Z_O2_tmp`) are declared. In this way, the iteration to read data from the DMA channel is performed over this local instances. Finally, the content of the local instance `parameters_tmp` is written in the shared register `parameters`. The following code snip shows this process.

**Listing 4.13:** `conv2d.cpp` - Load function, transfer from the external memory to the shared register `parameters` of the output quantization parameters

```
1  if (q_flag){
2
3    uint32_t dma_read_q_data_length = filt * 3 + 2;
4    dma_read_q_info = {offset_q_data, dma_read_q_data_length, DMA_SIZE};
5    bool dma_read_ctrl_done3 = false;
6   LOAD_CTRL_LOOP3:
7      do { dma_read_ctrl_done3 = dma_read_ctrl.nb_write(dma_read_q_info); } while (!
       dma_read_ctrl_done3);
8
9      ac_int<W_CROSS_BITWIDTH, true> WEIGHTS_CROSSPRODUCT_tmp[FILT_MAX];
10     ac_fixed<SF_IN_W_TOT_BITWIDTH, SF_IN_W_INT_BITWIDTH, true>
       SCALING_FACTOR_INPUTS_WEIGHTS_tmp[FILT_MAX];
```

```
11      ac_fixed<BIASQ_SCALED_TOT_BITWIDTH, BIASQ_SCALED_INT_BITWIDTH, true>
        BIASQ_SCALED_tmp[FILT_MAX];
12      ac_fixed<SF_OUT_INV_TOT_BITWIDTH, SF_OUT_INV_INT_BITWIDTH, true>
        SCALING_FACTOR_OUT_INVERSE_tmp;
13      ac_int<Z_BITWIDTH, true> Z_O2_tmp;
14      parameters_t parameters_tmp;
15
16      if (dma_read_ctrl_done3) {
17
18      LOAD_Q_LOOP:
19        for (uint16_t i = 0; i < (filt*3 + 2); i++){
20
21          #ifndef __SYNTHESIS__
22          while (!dma_read_chnl.available(1)) {}; // Hardware stalls until data
        ready
23          #endif
24          ac_int<W_CROSS_BITWIDTH, true> data_WC;
25          ac_int<SF_IN_W_TOT_BITWIDTH, true> data_SF;
26          ac_int<BIASQ_SCALED_TOT_BITWIDTH, true> data_BS;
27          ac_int<SF_OUT_INV_TOT_BITWIDTH, true> data_SFI;
28          ac_int<Z_BITWIDTH, true> data_Z;
29
30          if (i < filt){
31
32            data_WC = dma_read_chnl.read().template slc<W_CROSS_BITWIDTH>(0);
33            WEIGHTS_CROSSPRODUCT_tmp[i].set_slc(0, data_WC);
34            parameters_tmp.WEIGHTS_CROSSPRODUCT[i]=WEIGHTS_CROSSPRODUCT_tmp[i];
35
36          } else if (i >= filt && i < (filt*2)){
37
38            data_SF = dma_read_chnl.read().template slc<SF_IN_W_TOT_BITWIDTH>(0);
39            SCALING_FACTOR_INPUTS_WEIGHTS_tmp[i-filt].set_slc(0, data_SF);
40            parameters_tmp.SCALING_FACTOR_INPUTS_WEIGHTS[i-filt]=
        SCALING_FACTOR_INPUTS_WEIGHTS_tmp[i-filt];
41
42          } else if (i >= (filt*2) && i < (filt*3)) {
43
44            data_BS = dma_read_chnl.read().template slc<BIASQ_SCALED_TOT_BITWIDTH
        >(0);
45            BIASQ_SCALED_tmp[i-filt*2].set_slc(0, data_BS);
46            parameters_tmp.BIASQ_SCALED[i-filt*2]=BIASQ_SCALED_tmp[i-filt*2];
47
48          } else if (i == (filt*3)) {
49
50            data_SFI = dma_read_chnl.read().template slc<SF_OUT_INV_TOT_BITWIDTH>(0)
        ;
51            SCALING_FACTOR_OUT_INVERSE_tmp.set_slc(0, data_SFI);
52            parameters_tmp.SCALING_FACTOR_OUT_INVERSE=SCALING_FACTOR_OUT_INVERSE_tmp
        ;
53
54          } else {
55
56            data_Z = dma_read_chnl.read().template slc<Z_BITWIDTH>(0);
57            Z_O2_tmp.set_slc(0, data_Z);
58            parameters_tmp.Z_O2=Z_O2_tmp;
59
60          }
61
62          if (i == dma_read_q_data_length - 1) break;
63        }
64
```

```
65          parameters.write(parameters_tmp);
66      }
67
68  }
```

The same issue of declaring local structs on which perform memory operations concerns the shared memories `packed_inputs` and `packed_weights`. In this case, the data are transferred from the external environment via DMA to the local instances of the `plm_inputs_t` and `plm_filters_t` structs (`plm_in` and `plm_f`). Then, depending on the precision configuration selected for the ST multiplier, the packing operation on inputs and weights is performed, as already explained in Sec. 3.2.2. Finally, the input features and the weights correctly packed are stored in the shared memories `packed_inputs` and `packed_weights`. The code that performs the operations just explained is not reported, since it consists in the code implemented by D. R. Bueno Pacheco in [2] with just some adjustments to obtain the last step to store inside the shared memories the packed inputs and weights.

### 4.2.5 Compute Function

The compute function interface, shown in Lst. 4.14, consists of the following ports:

- `packed_inputs`, `packed_weights` and `parameters` ports: Through them the input features, the weights and the output quantization parameters are received from the load function.
- `OUT` port: Through it the output of the convolution is transmitted to the store function.
- `conf_info` port: It allows the function to receive the configuration parameters from the configure function.
- `done` port: Used to notify task completion to the other functions and synchronize with them.

**Listing 4.14:** `conv2d.cpp` - Compute function interface

```
1  void conv2d_m4_v10_reconf_reducedbitwidth(
2              ac_channel<packed_inputs_t> &packed_inputs,
3              ac_channel<packed_weights_t> &packed_weights,
4              ac_channel<parameters_t> &parameters,
5              ac_channel<plm_outputs_t> &OUT,
6              ac_channel<conf_info_t> &conf_info,
7              ac_sync &done)
```

As already explained for the load function, also in this case, in the same way, in the first part of the function the configuration parameters stored in the memory-mapped registers, set by the processor, are retrieved and the auxiliary variables

are defined to manage the loop boundaries. The only difference, reported in Lst. 4.15, from the previous function consists in the definition of `n_w_out` and `n_h_out`, auxiliary variables used to hold the dimensions of the output tile, taking into account the padding value and the stride.

**Listing 4.15:** `conv2d.cpp` - Compute function, part of the definition of the auxiliary variables

```
uint16_t n_w_out;
    uint16_t n_h_out;
    if (STRIDE == 1){
        n_w_out = (n_w_in - K_SIZE) + 1;
        n_h_out = (n_h_in - K_SIZE) + 1;
    }
    else{
        n_w_out = (n_w_in - K_SIZE)/2 + 1;
        n_h_out = (n_h_in - K_SIZE)/2 + 1;
    }
```

The packed inputs and packed weights, received from the `packed_inputs` and `packed_weights` ports, are stored into the local instances of the `packed_inputs_t` and `packed_weights_t` structs (`packed_inputs_tmp` and `packed_weights_tmp`), Lst. 4.16. Afterward, depending on if the output quantization is enabled or not, the output quantization parameters received from the `parameters` port are stored into the local instance of the `parameters_t` struct (`parameters_tmp`) and then every parameter is stored inside a proper variable, Lst.4.17. In particular, a loop iteration that corresponds to the number of output channels is performed to retrieve the arrays of values stored in the `WEIGHTS_CROSSPRODUCT`, `SCALING_FACTOR_INPUTS_WEIGHTS` and `BIASQ_SCALED` fields of the (`parameters_tmp`) struct. Once all the values from the input ports are retrieved, the compute function can start to manipulate them.

**Listing 4.16:** `conv2d.cpp` - Compute function, retreiving of the packed inputs and weights from the input ports

```
packed_inputs_t packed_inputs_tmp;
packed_weights_t packed_weights_tmp;

packed_inputs_tmp = packed_inputs.read();
packed_weights_tmp = packed_weights.read();
```

**Listing 4.17:** `conv2d.cpp` - Compute function, retreiving of the output quantization parameters from the input ports

```
struct parameters_t parameters_tmp;
if (EN_QUANTIZATION){
    parameters_tmp = parameters.read();

    for (uint16_t i = 0; i < OUT_CH; i++) {
    WEIGHTS_CROSSPRODUCT[i] = parameters_tmp.WEIGHTS_CROSSPRODUCT[i];
    SCALING_FACTOR_INPUTS_WEIGHTS[i] = parameters_tmp.
    SCALING_FACTOR_INPUTS_WEIGHTS[i];
    BIASQ_SCALED[i] = parameters_tmp.BIASQ_SCALED[i];
    }
```

```
10      SCALING_FACTOR_OUT_INVERSE = parameters_tmp.SCALING_FACTOR_OUT_INVERSE;
11      Z_O2 = parameters_tmp.Z_O2;
12  }
```

Before starting the convolution operation, the packed weights contained in the `packed_weights_tmp` struct are stored in an array with the help of four loops iterations, each of them corresponding to one dimension of the weights tensor, Lst. 4.18. The order of the loops corresponds on how the weight data is used during the convolution operation. Hence, the innermost loop iterates through the output channel dimension, the middle loops iterate first through the width dimension and then through the height dimension and finally the outermost loop iterates through the input channel dimension.

Let us understand why this process is implemented. When the high-level synthesis through Catapult HLS is performed, as it will be shown in Sec. 5.2, since the *loop unrolling* directive is applied to the innermost loop of the convolution operation, the *interleave* directive is applied to the resource containing the packed weights that is accessed in this loop. As stated by *Catapult Synthesis User and Reference Manual* [9], the *interleave* directive is not supported on shared memory type resources, like the `packed_weights_t` struct, thus they need to be transformed in typical arrays.

**Listing 4.18:** `conv2d.cpp` - Compute function, storing of the packed weights in an array variable

```
1  first_ci_for:
2  for (uint8 ci = 0; ci < N_C_MAX; ci ++) {
3  first_i_for:
4      for (uint8 i = 0; i < KERN_MAX; i++) {
5  first_j_for:
6          for (uint8 j = 0; j < KERN_MAX; j++) {
7  first_co_for:
8              for (uint16 co = 0; co < FILT_MAX; co++) {
9                  idx = (MAX_OUTPUT_CHANNELS * (MAX_INPUT_CHANNELS * (KERN_MAX * i +
       j) + ci) + co).to_int();
10
11                 b[idx] = packed_weights_tmp.data[idx];
12
13                 if (co == co_limit) break;
14             } // co
15             if (j == K_SIZE - 1) break;
16         } // k_w_for
17         if (i == K_SIZE - 1) break;
18     } // k_h_for
19     if (ci == in_ch_temp -1) break;
20 } // ci
```

The code that implements the convolution operation is mostly the one that was inside the `conv2d_m4_v10_reconf_reducedbitwidth` function of the sequential architecture inherited by [2]. The only relevant difference is in the innermost loop of the convolution, where in the original code the `multiplier_gautschi_noioreg` function (the function that implements the ST multipilier) was invoked to perform

multiplications, while in this case no function is invoked and the operation performed by the `multiplier_gautschi_noioreg` function are implemented directly by the compute function, Lst. 4.19. It is possible to observe how the code reported below implements an ST multiplier.

**Listing 4.19:** `conv2d.cpp` - Compute function, multiplication phase in the convolution operation

```
co_for:
for (uint16 co = 0; co < FILT_MAX; co++) {

    int32 product;

    idx = (MAX_OUTPUT_CHANNELS * (MAX_INPUT_CHANNELS * (KERN_MAX * i + j) + ci) +
    co).to_int();

    b_int = b[idx];

    int32 output;

    if (CONFIG1 == 4) { // 16x8
        output = a * b_int.slc<8>(0);
    } else if (CONFIG1 == 1) { // 4x4
        output = a.slc<4>(12) * b_int.slc<4>(0) + a.slc<4>(8) * b_int.slc<4>(4) + a
    .slc<4>(4) * b_int.slc<4>(8) + a.slc<4>(0) * b_int.slc<4>(12);
    } else if (CONFIG1 == 2) { // 8x8
        output = a.slc<8>(8) * b_int.slc<8>(0) + a.slc<8>(0) * b_int.slc<8>(8);
    } else if (CONFIG1 == 3) { // 8x4
        output = a.slc<8>(8) * b_int.slc<4>(0) + a.slc<8>(0) * b_int.slc<4>(8);
    } else { // 16x16
        output = a * b_int;
    }

    product = output;
    ACC:
    output_acc[co] += product;

    if (co == co_limit) break;
}
```

### 4.2.6 Store Function

The store function interface, shown in Lst. 4.20, consists of the following ports:

- `conf_info` port: It allows the function to receive the configuration parameters from the configure function.
- `plm_outputs`: Through it the output of the convolution is received from the compute function.
- `dma_write_ctrl`: Used to program the DMA controller.
- `dma_write_chnl`: Used to sent data to the main memory.

- `done` port: Used to notify task completion to the other functions and synchronize with them.

**Listing 4.20:** `conv2d.cpp` - Store function interface

```
void store(
        ac_channel<conf_info_t> &conf_info,
        ac_channel<plm_outputs_t> &plm_outputs,
        ac_channel<dma_info_t> &dma_write_ctrl,
        ac_channel<dma_data_t> &dma_write_chnl,
        ac_sync &done)
```

As already explained for the load and compute functions, also in this case, in the first part of the function the configuration parameters stored in the memory-mapped registers, set by the processor, are retrieved and the auxiliary variables (the same as the compute function) are defined to manage the loop boundaries.

As already explained in Sec. 4.2.2, in order to prevent unintentional inference of additional memories or registers to which the shared arrays are mapped, local structs containing the packed arrays must be declared inside every function to handle memory operations. For this reason, when transferring data from the shared memory, `plm_outputs`, to the external environment via DMA, a local instance of the `plm_ouputs_t` struct (`plm_ouputs_tmp`) is declared. In this way, after the content of the shared memory is read in the local struct, the iteration to write data in the DMA channel is performed over this local instances. The following code snip shows this process.

**Listing 4.21:** `conv2d.cpp` - Store function, transfer from the shared memory `plm_outputs` to the external memory of the output values

```
plm_outputs_tmp = plm_outputs.read();
dma_write_data_length = n_w_out * n_h_out;

STORE_CO_LOOP: for (uint16_t co = 0; co < FILT_MAX; co++){
    uint32_t offset_write = out_add + co*offset_PE_out;
    dma_write_info = {offset_write, dma_write_data_length, DMA_SIZE};
    bool dma_write_ctrl_done = false;
    STORE_CTRL_LOOP:
    do {dma_write_ctrl_done = dma_write_ctrl.nb_write(dma_write_info);} while (!
    dma_write_ctrl_done);

    if (dma_write_ctrl_done) {
        STORE_H_LOOP: for (uint16_t h = 0; h < N_H_OUT_MAX ; h++){
            STORE_W_LOOP:    for (uint16_t w = 0; w < N_W_OUT_MAX; w++){

            uint32_t index = MAX_OUTPUT_CHANNELS * (MAX_OUTPUT_WIDTH * h + w) + co
    ;
            FPDATA_OUT data = plm_outputs_tmp.data[index];
            assert(DMA_WIDTH == 64 && "DMA_WIDTH should be 64 (simplicity choice)"
    );
            ac_int<DMA_WIDTH, false> data_ac;
            ac_int<32, false> DEADBEEF = 0xdeadbeef;
            data_ac.set_slc(32, DEADBEEF.template slc<32>(0));
            data_ac.set_slc(0, data.template slc<DATA_WIDTH>(0));
```

```
22
23                dma_write_chnl.write(data_ac);
24
25                if (w == n_w_out -1) break;
26                }
27            if (h == n_h_out -1) break;
28            }
29        }
30  if (co == filt -1 ) break;
31  }
```

# Chapter 5

# C++ Simulations, High-Level Synthesis, FPGA Implementation and Results

In this chapter, the correctness of the hierarchical architecture of the ST based 2D-Convolution accelerator, implemented in Ch. 4, will be tested through C++ simulations. Then, the high-level synthesis through Catapult HLS by Siemens will be performed, obtaining the RTL implementation of the accelerator. In this way, a co-simulation will be carried out where the RTL is compared against the untimed C++ to check if there are no functional errors. Finally, the accelerator, integrated in a SoC exploiting the ESP framework, will be deployed on an FPGA on which a bare-metal test application will run, further testing the architecture. All these experiments will be carried out in parallel with the same ones related to the sequential architecture, and this is due to two main reasons:

- to validate the whole sequential architecture, together with the output quantization part that has not been validated in the previous thesis work [2].
- to perform the high-level synthesis of the sequential architecture, the same directives utilized for the hierarchical architecture, except those related to the dataflow, are applied; in this way, we will obtain a sequential accelerator comparable with the hierarchical one.

In the end, the results obtained from the FPGA deploying will highlight the conditions under which the two architectures perform optimally. Additionally, the performance of the hardware accelerators will be compared with the performance of the RISC-V processor alone, when computing convolution operations.

The essential steps to set up the framework and environment variables required to replicate the mentioned tests are detailed in these thesis works [13], [14] and [2].

These, together with *Catapult Synthesis User and Reference Manual* [9], can be used as reference guides for reproducing the workflow from the ground up. Therefore the following chapter will focus on explaining the differences in the workflow and discussing the considerations made to validate the hierarchical architecture and compare it with the sequential architecture and the RISC-V processor.

## 5.1 C++ Simulation

The first step in validating the correct functionality of the hierarchical architecture of the accelerator is to conduct a C++ simulation, using the push-button SystemC simulation flow integrated in Siemens Catapult HLS, where the accelerator is isolated from the rest of the SoC. In this simulation, the testbench, `main.cpp`, must emulate the operations of the processor. This involves setting the all the configuration parameters and supplying the DMA channels of the accelerator, `conv2d_cxx_catapult`, with inputs, weights and output quantization parameters. The same configuration parameters, inputs, weights and output quantization parameters are passed to the *golden function* `conv2d_tb` inside the testbench. Finally, to validate the correctness of the accelerator, its output is fetched from the accelerator's output memory and compared with the *golden output*.

The first difference from the testbench utilized in [2] to test the sequential architecture consists of the kind of data supplied to the DMA channels of the accelerator. The testbench implemented in [2] supplied to the DMA channels only the input and weight values, which were randomly generated, while in this thesis, realistic stimuli, that are inputs, weights and also the output quantization parameters (as stated before, this actual testbench, differently from the one in [2], allows to validate the output quantization part too), are supplied to these channels. In particular, to validate the architecture ensuring a realistic behavior, a code based on TensorFlow, and developed in previous works, is used to perform the convolution operation, also incorporating quantization. This code generates and writes to text files inputs, weights, outputs and the parameters needed to apply quantization to the final results, of a 2D-convolutional layer with specified dimensions set by the designer. Then, a Python script, developed in [2], is utilized to read the values and parameters from the text files and store them in arrays within header files. Finally, this header files are imported by the testbench and their data is loaded in the channels, as shown in the following code snip.

**Listing 5.1:** `main.cpp` - Main function, supplying the DMA channels of the accelerator with inputs, weights and output quantization parameters

```
1 for (unsigned i = 0; i < inputs_size; i++) {
2     inputs[i] = input[i];
3
4     ac_int<DMA_WIDTH, true> data_ac;
```

```
5       ac_int<DMA_WIDTH/2, true> DEADBEEF = 0xdeadbeef;
6       data_ac.set_slc(DMA_WIDTH/2, DEADBEEF.template slc<DMA_WIDTH/2>(0));
7       data_ac.set_slc(0, inputs[i].template slc<DATA_WIDTH>(0));
8
9       dma_read_chnl.write(data_ac);
10  }
11  for (unsigned i = 0; i < weights_size; i++) {
12      weights[i] = weight[i];
13
14      ac_int<DMA_WIDTH, true> data_ac;
15      ac_int<DMA_WIDTH/2, true> DEADBEEF = 0xdeadbeef;
16      data_ac.set_slc(DMA_WIDTH/2, DEADBEEF.template slc<DMA_WIDTH/2>(0));
17      data_ac.set_slc(0, weights[i].template slc<DATA_WIDTH>(0));
18
19      dma_read_chnl.write(data_ac);
20  }
21
22  for (unsigned i = 0; i < w_cross_size; i++) {
23
24      w_cross_q[i] = w_cross[i];
25
26      ac_int<DMA_WIDTH, true> data_ac;
27      ac_int<DMA_WIDTH/2, true> DEADBEEF = 0xdeadbeef;
28      data_ac.set_slc(DMA_WIDTH/2, DEADBEEF.template slc<DMA_WIDTH/2>(0));
29      data_ac.set_slc(0, w_cross_q[i].template slc<DATA_WIDTH>(0));
30
31      dma_read_chnl.write(data_ac);
32  }
33
34  for (unsigned i = 0; i < scaling_factor_iw_size; i++) {
35
36      scaling_factor_iw_q[i] = scaling_factor_iw[i];
37
38      ac_int<DMA_WIDTH, true> data_ac;
39      ac_int<DMA_WIDTH/2, true> DEADBEEF = 0xdeadbeef;
40      data_ac.set_slc(DMA_WIDTH/2, DEADBEEF.template slc<DMA_WIDTH/2>(0));
41      data_ac.set_slc(0, scaling_factor_iw_q[i].template slc<DATA_WIDTH>(0));
42
43      dma_read_chnl.write(data_ac);
44  }
45
46  for (unsigned i = 0; i < biasq_scaled_size; i++) {
47
48      biasq_scaled_q[i] = biasq_scaled[i];
49
50      ac_int<DMA_WIDTH, true> data_ac;
51      ac_int<DMA_WIDTH/2, true> DEADBEEF = 0xdeadbeef;
52      data_ac.set_slc(DMA_WIDTH/2, DEADBEEF.template slc<DMA_WIDTH/2>(0));
53      data_ac.set_slc(0, biasq_scaled_q[i].template slc<DATA_WIDTH>(0));
54
55      dma_read_chnl.write(data_ac);
56  }
57
58  for (unsigned i = 0; i < scaling_factor_oi_size; i++) {
59
60      scaling_factor_oi_q[i] = scaling_factor_oi[i];
61
62      ac_int<DMA_WIDTH, true> data_ac;
63      ac_int<DMA_WIDTH/2, true> DEADBEEF = 0xdeadbeef;
64      data_ac.set_slc(DMA_WIDTH/2, DEADBEEF.template slc<DMA_WIDTH/2>(0));
65      data_ac.set_slc(0, scaling_factor_oi_q[i].template slc<DATA_WIDTH>(0));
```

```
66
67      dma_read_chnl.write(data_ac);
68  }
69
70  for (unsigned i = 0; i < z_o_size; i++) {
71
72      z_o_q[i] = z_o[i];
73
74      ac_int<DMA_WIDTH, true> data_ac;
75      ac_int<DMA_WIDTH/2, true> DEADBEEF = 0xdeadbeef;
76      data_ac.set_slc(DMA_WIDTH/2, DEADBEEF.template slc<DMA_WIDTH/2>(0));
77      data_ac.set_slc(0, z_o_q[i].template slc<DATA_WIDTH>(0));
78
79      dma_read_chnl.write(data_ac);
80  }
```

In Lst. 5.2, the code that performs the output quantization inside the *golden function*, `conv2d_tb`, is reported. It has been obtained modifying and adapting to our case the code already present in the compute function of the sequential architecture inherited in [2]. This part of the code, that implements the algorithm explained in Sec. 4.1.3, allows to validate that the output quantization is correctly done by the accelerator. It also represents the second difference from the testbench utilized in [2], inside which the code shown in Lst. 5.2 was not present, since, as stated before, the output quantization part was not validated in this previous thesis work.

**Listing 5.2:** `main.cpp` - golden function, `conv2d_tb`, output quantization

```
1  if (EN_QUANTIZATION == 0) {
2
3      wb_for:
4      for (uint8 co = 0; co < filt; co++) {
5          unsigned index3 = n_w_out * n_h_out * co + i * n_w_out + j;
6          output[index3] = output_bfq[co];
7      }
8
9  }  else { // if (EN_QUANTIZATION == 1)
10
11     q_for:
12     for (uint8 co = 0; co < filt; co++) {
13         unsigned index3 = n_w_out * n_h_out * co + i * n_w_out + j;
14
15         final_output[co] = 0;
16         SUB0_conf2:
17         outputq_sub[co] = output_bfq[co] - w_cross[co];
18         MUL0_conf2:
19         outputq_scaled[co] = outputq_sub[co] * scaling_factor_iw[co];
20         ADD0_conf2:
21         outputq_bias[co] = outputq_scaled[co] + biasq_scaled[co];
22
23
24         if (CONFIG2 == 1) { // 4x
25
26             if (EN_RELU == 1 && outputq_bias[co] < 0) {
27                 outputq_quantized_4x[co] = z_o[0];
28             } else {
```

```
29                  outputq_quantized_4x[co] = z_o[0] + outputq_bias[co] *
    scaling_factor_oi[0];
30              }
31          final_output[co] = outputq_quantized_4x[co].to_ac_int();
32
33
34          } else if (CONFIG2 == 2) { // 8x
35
36              if (EN_RELU == 1 && outputq_bias[co] < 0) {
37                  outputq_quantized_8x[co] = z_o[0];
38              } else {
39                  outputq_quantized_8x[co] = z_o[0] + outputq_bias[co] *
    scaling_factor_oi[0];
40              }
41              final_output[co] = outputq_quantized_8x[co].to_ac_int();
42
43
44          } else { // 16x
45
46              if (EN_RELU == 1 && outputq_bias[co] < 0) {
47                  outputq_quantized_16x[co] = z_o[0];
48              } else {
49                  outputq_quantized_16x[co] = z_o[0] + outputq_bias[co] *
    scaling_factor_oi[0];
50              }
51              final_output[co] = outputq_quantized_16x[co].to_ac_int();
52
53          }
54
55          output[index3] =  final_output[co];
56      }
57  }
```

Fig. 5.1 shows the result of the simulation of the accelerator with hierarchical architecture, considering a layer with the following dimensions:

- Width = 8
- Height = 8
- Input channels = 16
- Kernel width and height = 3
- Ouptut channels = 4
- Stride = 1
- Paddding = not apllied

As already remarked, the sequential architecture is tested too, since its output quantization part has not been validated in [2]. The dimensions of the layer used are the same considered for the hierarchical architecture's simulation and Fig. 5.2 shows the result of the test.

```
Simulating design
cd ../..; ./conv2d_cxx_catapult_hier_fx32_dma64/conv2d_cxx_catapult.v1/scverify/orig_cxx_osci/scverify_top
Info: main(): --------------------------------
Info: main(): ESP - Conv2D [Catapult HLS C++]
Info: main():       Hierarchical blocks
Info: main(): --------------------------------
Info: main(): Configuration:
Info: main():   - n_w: 8
Info: main():   - n_h: 8
Info: main():   - n_c: 16
Info: main():   - kern: 3
Info: main():   - filt: 4
Info: main(): Other info:
Info: main():   - DMA width: 64
Info: main():   - DMA size [2 = 32b, 3 = 64b]: 3
Info: main():   - DATA width: 32
Info: main():   - Input size: 1024
Info: main():   - memory in (words): 1024
Info: main():   - memory out (words): 144
Info: main(): --------------------------------
Info: main(): Validation: PASS
Info: main():   - errors 0 / total 144
Info: main(): --------------------------------
```

**Figure 5.1:** C++ Testbench simulation result of the hierarchical architecture.

```
Simulating design
cd ../..; ./conv2d_cxx_catapult_basic_fx32_dma64/conv2d_cxx_catapult.v1/scverify/orig_cxx_osci/scverify_top
Info: main(): --------------------------------
Info: main(): ESP - Conv2D [Catapult HLS C++]
Info: main():       Single block
Info: main(): --------------------------------
Info: main(): Configuration:
Info: main():   - n_w: 8
Info: main():   - n_h: 8
Info: main():   - n_c: 16
Info: main():   - kern: 3
Info: main():   - filt: 4
Info: main(): Other info:
Info: main():   - DMA width: 64
Info: main():   - DMA size [2 = 32b, 3 = 64b]: 3
Info: main():   - DATA width: 32
Info: main():   - Input size: 1024
Info: main():   - memory in (words): 1024
Info: main():   - memory out (words): 144
Info: main(): --------------------------------
Info: main(): Validation: PASS
Info: main():   - errors 0 / total 144
Info: main(): --------------------------------
```

**Figure 5.2:** C++ Testbench simulation result of the sequential architecture.

## 5.2 High-Level Synthesis and Co-Simulation

The next step consists of performing high-level synthesis (HLS) to generate the RTL code of the accelerator and integrating it into an ESP-generated SoC. The HLS tool used for this process is Siemens Catapult HLS. The `build_prj.tcl` script contains all the Catapult HLS directives used to synthesize the C++ code of the accelerator. Many important directives are set such as clock period, design goal, pipeline constraints and unrolling constraints. For what concerns clock period and design goal directives, the synthesis workflow follows a similar approach to previous works, [13], [14] and [2]. The differences concern the directives setting pipeline

and unrolling constraints reported in Lst. 5.3. The first set of directives is applied to synthesize the hierarchical architecture, while the second one to synthesize the sequential one. As stated before, the goal is to synthesize the two architectures applying the same constraints (with the exception of the dataflow directives applied only for the hierarchical architecture; it is explained below), in such a way to obtain two comparable accelerators.

The only difference between the two cases consists in the first set of directives related to the hierarchical architecture (rows 3-13), which apply pipeline to all the functions in the architecture (config, load, compute and store), allowing them to run in parallel. In particular, the functions are pipelined with an Initiation Interval (II) of 1. This configuration allows them to accept new input data on every clock cycle. However, if a function's input data is not available during a specific clock cycle, Catapult will, by default, completely stall the design (iterations without valid data are referred to as bubbles). Using the `PIPELINE_STALL_MODE` directive, the designer can specify how Catapult should handle cases when input data is unavailable for the pipeline. When the pipeline stall mode is set to `flush` the pipeline will not stall due to missing input data. This mode allows previous iterations that already have all the required data to continue processing and flush[1] by injecting bubbles into the pipeline.

The other directives, similar for the two cases, apply pipeline to most of the loops in the compute function/phase. In particular, for both the architectures, an accelerator with multiple PEs must be synthesized. Consequently, a loop unrolling directive for the innermost loop of the convolution operation needs to be added to the synthesis script. Utilizing multiple PEs requires a simultaneous access to multiple data from the weight PLM (resource accessed in the loop unrolled) to fully exploit the unrolled loops and avoid memory bottlenecks. Since the memory blocks have only one read port and one write port, the interleave directive must be employed to divide the original weight PLM into several smaller memory banks. In this way, each unrolled loop has access to its own interleaved weight PLM.

**Listing 5.3:** `build_prj.tcl` - directives setting pipeline and unrolling constraints

```
1 if {$opt(hier)} {
2
3     directive set /$ACCELERATOR/config/core/main -PIPELINE_INIT_INTERVAL 1
4     directive set /$ACCELERATOR/config/core/main -PIPELINE_STALL_MODE flush
5
6     directive set /$ACCELERATOR/load/core/main -PIPELINE_INIT_INTERVAL 1
7     directive set /$ACCELERATOR/load/core/main -PIPELINE_STALL_MODE flush
8
9     directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/main -
    PIPELINE_INIT_INTERVAL 1
```

[1]flush: *Catapult Synthesis User and Reference Manual* [9] uses this term to indicate that the iterations proceed and finish.

```
10      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/main -
        PIPELINE_STALL_MODE flush
11
12      directive set /$ACCELERATOR/store/core/main -PIPELINE_INIT_INTERVAL 1
13      directive set /$ACCELERATOR/store/core/main -PIPELINE_STALL_MODE flush
14
15      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/b:rsc -
        INTERLEAVE $MAX_OUTPUT_CHANNELS
16
17      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/in_h_for
                -PIPELINE_INIT_INTERVAL 1
18      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/in_w_for
                -PIPELINE_INIT_INTERVAL 1
19      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/
        first_co_for  -PIPELINE_INIT_INTERVAL 1
20      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/k_h_for
                -PIPELINE_INIT_INTERVAL 1
21      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/k_w_for
                -PIPELINE_INIT_INTERVAL 1
22      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/ci_for
              -PIPELINE_INIT_INTERVAL 1
23      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/co_for
              -UNROLL yes
24      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/init_for
              -PIPELINE_INIT_INTERVAL 1
25      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/wb_for
              -PIPELINE_INIT_INTERVAL 1
26      directive set /$ACCELERATOR/conv2d_m4_v10_reconf_reducedbitwidth/core/q_for
              -PIPELINE_INIT_INTERVAL 1
27      } else {
28
29      directive set /$ACCELERATOR/core/B_reconf_HH:rsc -INTERLEAVE
        $MAX_OUTPUT_CHANNELS
30      directive set /$ACCELERATOR/core/B_reconf_HL:rsc -INTERLEAVE
        $MAX_OUTPUT_CHANNELS
31      directive set /$ACCELERATOR/core/B_reconf_LH:rsc -INTERLEAVE
        $MAX_OUTPUT_CHANNELS
32      directive set /$ACCELERATOR/core/B_reconf_LL:rsc -INTERLEAVE
        $MAX_OUTPUT_CHANNELS
33
34      directive set /$ACCELERATOR/core/in_h_for          -PIPELINE_INIT_INTERVAL 1
35      directive set /$ACCELERATOR/core/in_w_for          -PIPELINE_INIT_INTERVAL 1
36      directive set /$ACCELERATOR/core/k_h_for           -PIPELINE_INIT_INTERVAL 1
37      directive set /$ACCELERATOR/core/k_w_for           -PIPELINE_INIT_INTERVAL 1
38      directive set /$ACCELERATOR/core/ci_for          -PIPELINE_INIT_INTERVAL 1
39      directive set /$ACCELERATOR/core/co_for          -UNROLL yes
40      directive set /$ACCELERATOR/core/init_for        -PIPELINE_INIT_INTERVAL 1
41      directive set /$ACCELERATOR/core/wb_for          -PIPELINE_INIT_INTERVAL 1
42      directive set /$ACCELERATOR/core/q_for           -PIPELINE_INIT_INTERVAL 1
43      }
```

Fig. 5.3, Fig. 5.4, Fig. 5.5 and Fig. 5.6 show respectively the loops inside the config, load, compute and store functions of the hierarchical architecture. It is possible to observe how the outermost loop, *main*, of each function is pipelined with an II of 1, as stated by the dataflow directives (rows 3-13 of Lst. 5.3), and consequently all the other loops inside those functions are pipelined with the same II. For this reason, the pipeline directives inside the compute function of the hierarchical architecture could be avoided, while the directive that set the unrolling

of the innermost loop of the convolution operation should be still specified. The unrolling generates 8 PEs, as highlighted in Fig. 5.5. Fig. 5.7 presents the list of loops on which the pipeline and loop unrolling directives are applied in the case of the sequential architecture, highlighting the loop where unrolling is applied (`co_for`) to generate 8 PEs.

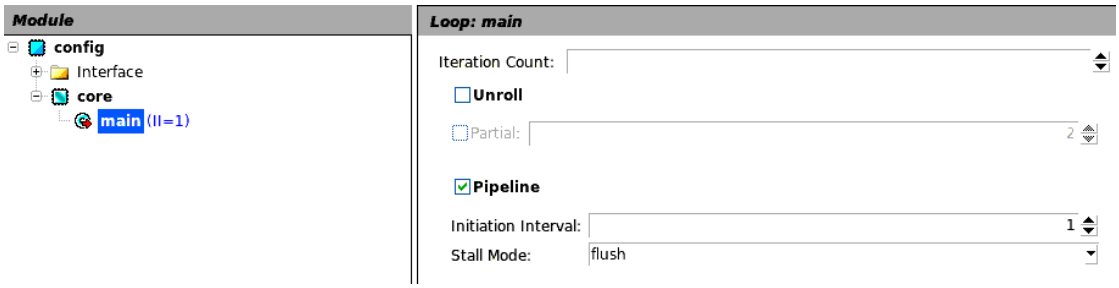


**Figure 5.3:** Loops inside the config function, hierarchical architecture.

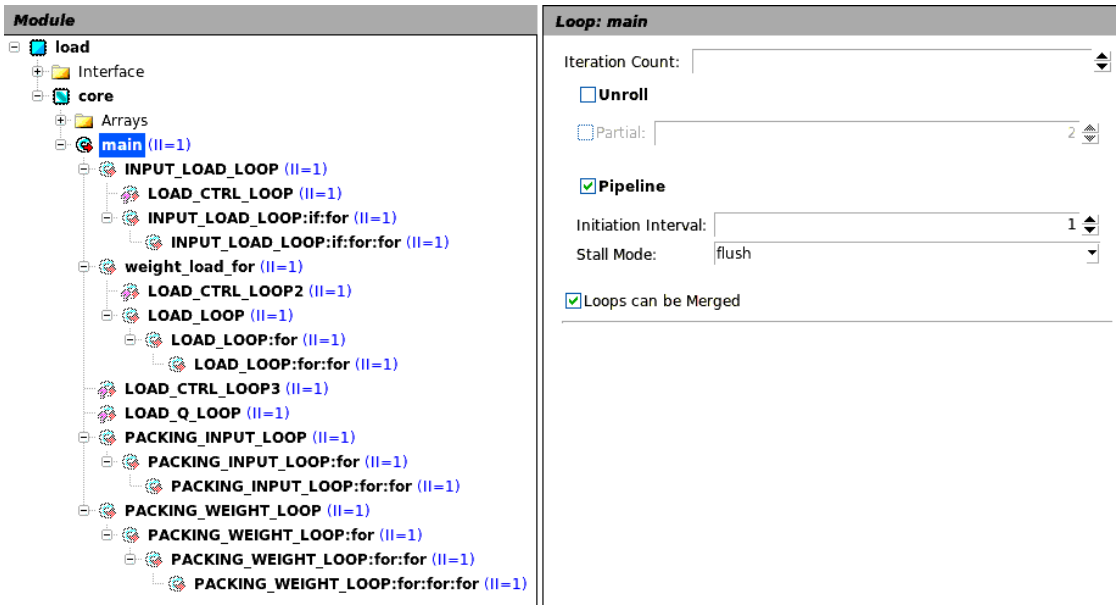


**Figure 5.4:** Loops inside the load function, hierarchical architecture.

Once the high-level synthesis is performed and the RTL implementation of the accelerator is obtained, a co-simulation is performed, both for the hierarchical and sequential architecture, where the RTL is compared against the untimed C++ model to ensure there are no errors. During this phase, QuestaSim is used. The same C++ testbench analyzed in 5.1, setting the same layer's dimensions, is used. Fig. 5.8 and Fig. 5.10 show that, in both cases, no errors are detected (*Simulation*

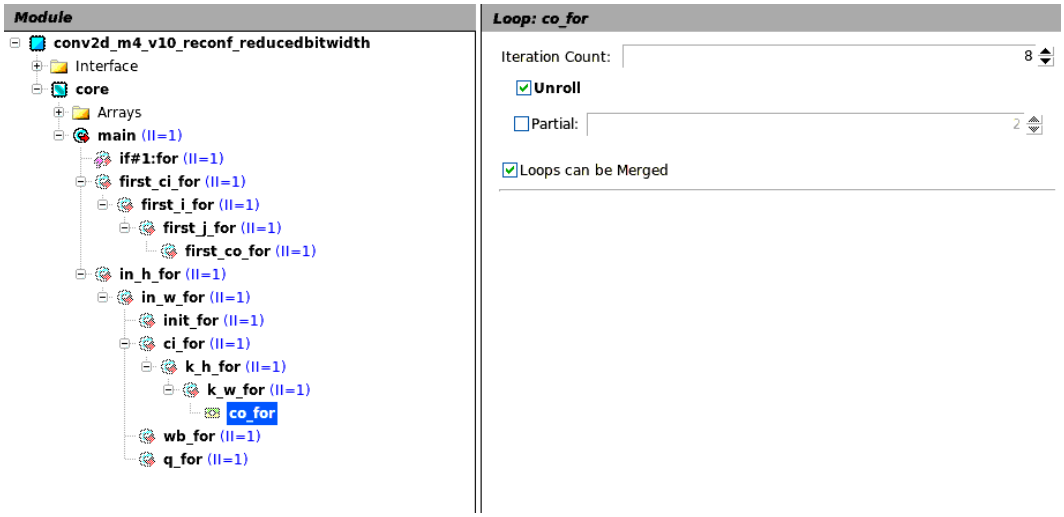


**Figure 5.5:** Loops inside the compute function, hierarchical architecture.

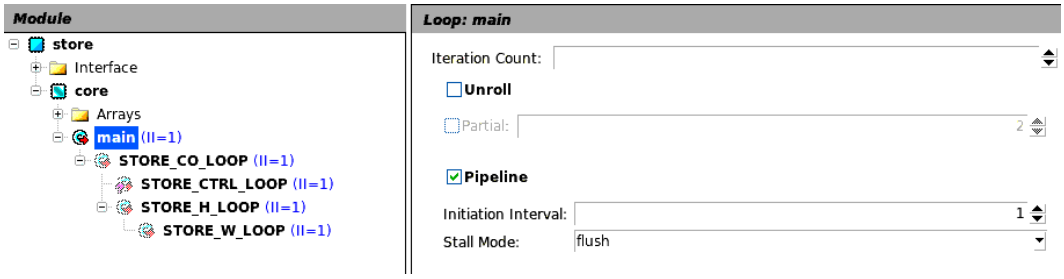


**Figure 5.6:** Loops inside the store function, hierarchical architecture.

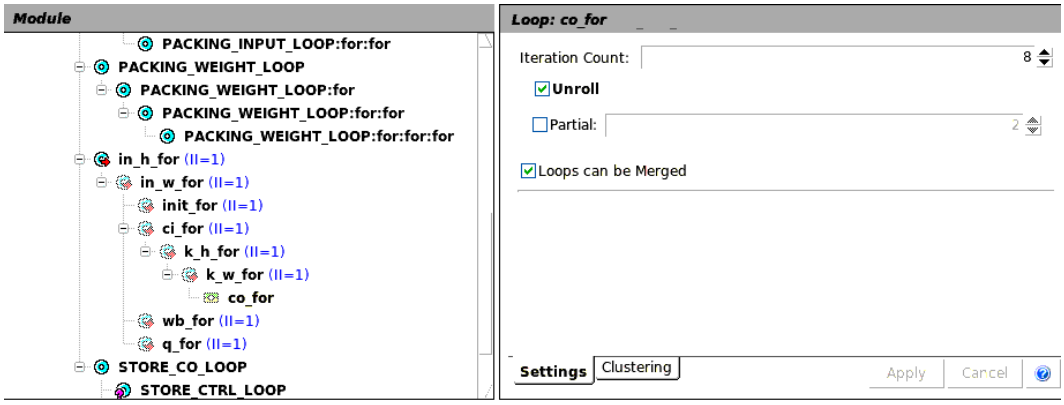


**Figure 5.7:** Focus on the pipelined and unrolled loops, sequential architecture.

*PASSED*). Also, it is possible to observe in Fig. 5.9 and Fig. 5.11, the moment when the signal `acc_done` indicates that the accelerator has finished the computation. In particular, the hierarchical architecture takes 634350 $ns$ from the moment the reset is deactivated to the moment `acc_done` goes high. The sequential architecture takes 1023700 $ns$ instead. Therefore, the hierarchical architecture requires 39% less time than the sequential architecture to perform the same computation.

```
# Info: scverify_top/user_tb: Simulation PASSED @ 634576 ns
# ** Note: (vsim-6574) SystemC simulation stopped by user.
# 1
#
```

**Figure 5.8:** RTL/C++ co-simulation result, hierarchical architecture.

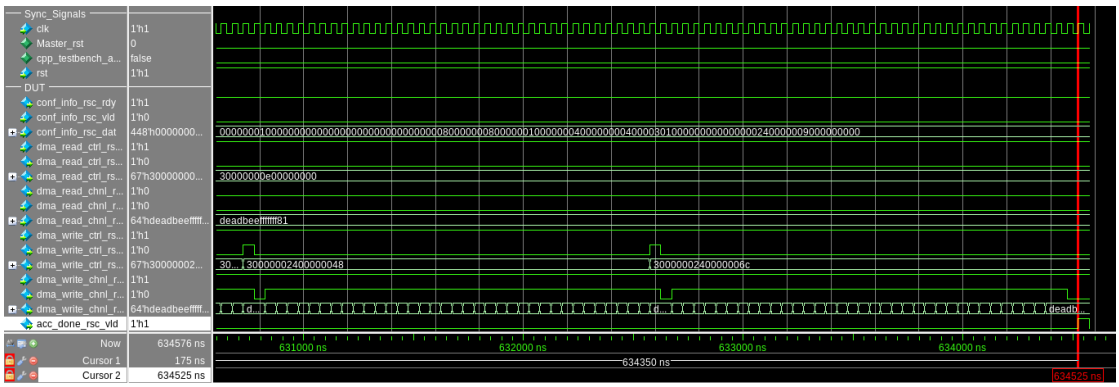


**Figure 5.9:** RTL/C++ co-simulation waveforms, hierarchical architecture.

```
# Info: scverify_top/user_tb: Simulation PASSED @ 1023926 ns
# ** Note: (vsim-6574) SystemC simulation stopped by user.
# 1
#
```

**Figure 5.10:** RTL/C++ co-simulation result, sequential architecture.

## 5.3 FPGA Prototyping and Validation

Finally, the architectures can be tested on the ProFPGA XC7V2000T board. To do this, the accelerators have to be integrated into a heterogeneous SoC first. ESP simplifies this step providing a SoC design GUI that, once opened, displays a default SoC configuration, where the tiles are organized in a 2×2 grid. This

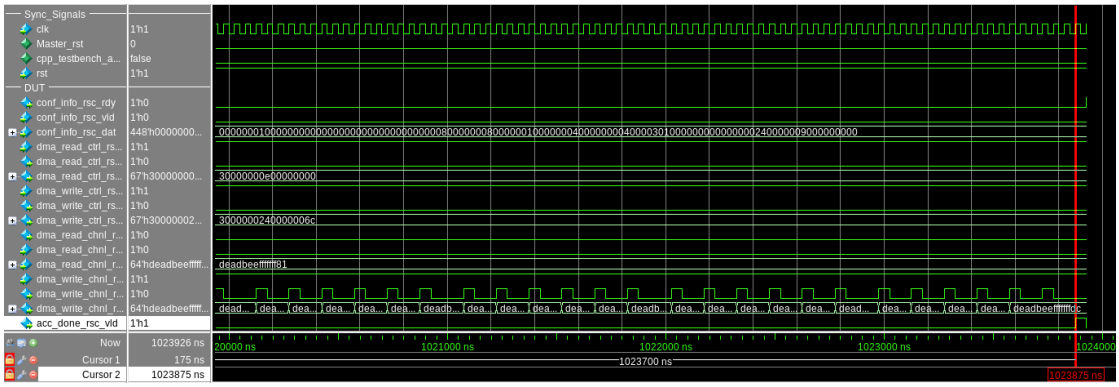


**Figure 5.11:** RTL/C++ co-simulation waveforms, sequential architecture.

grid includes the processor core, the memory controller and memory channel, an auxiliary tile and one empty tile. For our design, the modern Ariane RISC-V processor core is selected and the empty tile is replaced with an accelerator tile containing the RTL implementation of the 2D-convolution accelerator. Both the sequential and hierarchical architecture are selected, one at a time, obtaining two different SoC configurations. The resulting SoC configuration, when the hierarchical architecture of the accelerator is integrated, is shown in Fig. 5.12. Based on the SoC configuration, the corresponding RTL implementation is automatically generated.


**Figure 5.12:** SoC design in the ESP GUI with the hierarchical architecture of the 2D-Conv accelerator.

Finally, the two SoC designs, featuring one the sequential and the other the hierarchical implementation of the accelerator, can be deployed on the FPGA board. The corresponding bitstreams (`top.bit`) are generated using Xilinx Vivado and uploaded into the FPGA. The FPGA is connected via Ethernet to a remote server (host), which can be accessed through SSH tunneling. The UART interface of the SoC can be accessed with Minicom, a serial communication program, as the FPGA is equipped with a UART interface board connected to the host computer via USB.

With a bare-metal test application running on the FPGA, we are able to validate our accelerators at the system level, accounting for the processor and all other peripherals. Its output is displayed on the host's screen via Minicom.

The bare-metal test application has been implemented in the previous thesis work [2] and performs the tiling algorithm described in Sec. 3.2.1. To be more specific, the bare-metal code includes the following functions:

- `get_tiling` It performs the actual tiling of the input and weight tensors.
- `conv2d_sw` It performs the 2D-Convolution in software.
- `conv2d_hw` It supplies to the accelerator all the configuration parameters and invokes it.
- `conv2d_tiling` It calls the `get_tiling` function and, once the tiles are obtained, controls how they are going to be processed by the `conv2d_hw` function calculating the pointer offsets to correctly address the right tiles, as explained in Sec. 3.2.1.
- `conv2d_tilingSW` It calls the `get_tiling` function and, once the tiles are obtained, controls how they are going to be processed by the `conv2d_sw` function calculating the pointer offsets to correctly address the right tiles, as explained in Sec. 3.2.1.
- `main` It fills the external memory with input, weight and output quantization values from the header files analyzed in Sec. 5.1. It then runs the `conv2d_tilingSW` and `conv2d_tiling` functions, measuring the time required to execute them, and so obtaining the number of clock cycles required to perform the 2D convolution in both software and hardware.

Lst. 5.4 shows the part of the bare-metal code where the layer dimensions, the padding type, the stride and the precision desired are set. Then, the PLMs dimensions and the number of PEs wanted are defined. In particular, the PLMs dimensions determine if and how the input and weight tensors are tiled, as explained in Sec. 3.2.1. For example, the values set in the code snip below cause a simulation of the accelerators with no tiling applied, since the PLMs dimensions are larger than the layer ones.

**Listing 5.4:** `conv2d_cxx.c` - setting the values defining the test to be performed

```
const uint16_t Hin = 8;
const uint16_t Win = 8;
const uint16_t Cin = 16;
const uint16_t ker = 3;
const uint16_t Cout = 4;
const uint16_t stride=1;
const uint16_t pad = 0;

const int precision_opt = 0;  // CONFIG1 if 0 just it takes one cin value, if 1 it takes 4 values (4 LSB) in cin , if 2 or 3 it takes 2 values (8 LSB) in cin

const uint8_t q_confg = precision_opt; // CONFIG2

const uint16_t plm_in = 18*18*8; // Hin * Win * Cin
const uint16_t plm_w = 7*7*16*8; // ker * ker * Cin * Cout
const uint16_t plm_out = 18*18*8; // Hout * Wout * Cout
const uint8_t PE = 8;
```

Fig. 5.13 and Fig. 5.14 show the results of the 2D-Convolution accelerators implemented in the FPGA when no tiling is applied to the input and weight tensors.

```
Available local input memory is:        5184    Memory needed is:       1024
Available local weight memory is:       6272    Memory needed is:       576
Available local output memory is:       2592    Memory needed is:       256
Tile dimensions are - Hin x Win x Cin x Cout: 8 8 16 4
Total number of tiles(aprox): 1
        -> Non-coherent DMA
  Tiled Convolution in hardware done
  Elapsed time = 2512690
Available local input memory is:        5184    Memory needed is:       1024
Available local weight memory is:       6272    Memory needed is:       576
Available local output memory is:       2592    Memory needed is:       256
Tile dimensions are - Hin x Win x Cin x Cout: 8 8 16 4
Total number of tiles(aprox): 1
  Tiled Convolution in software done
  Elapsed time = 4080300
Number of errors: 00000000
DONE riscv
```

**Figure 5.13:** 2D-Conv results in FPGA, hierarchical architecture.

## 5.4 Performance Results

The 2D-Convolution accelerators have been tested with two layers of different dimensions reported in the seventh columns of Tab. 5.1 and Tab. 5.2:

- A common layer from [18] (Tab. 5.1): Input Height x Input Width x Input Channels x Output Channel x Kernel Height/Width = 8 x 8 x 16 x 4 x 3.
- The last layer of MobileNet (Tab. 5.2): Input Height x Input Width x Input Channels x Output Channel x Kernel Height/Width = 3 x 3 x 256 x 256 x 1.

```
Available local input memory is:        5184    Memory needed is:       1024
Available local weight memory is:       6272    Memory needed is:       576
Available local output memory is:       2592    Memory needed is:       256
Tile dimensions are - Hin x Win x Cin x Cout: 8 8 16 4
Total number of tiles(aprox): 1
        -> Non-coherent DMA
  Tiled Convolution in hardware done
  Elapsed time = 4070077
Available local input memory is:        5184    Memory needed is:       1024
Available local weight memory is:       6272    Memory needed is:       576
Available local output memory is:       2592    Memory needed is:       256
Tile dimensions are - Hin x Win x Cin x Cout: 8 8 16 4
Total number of tiles(aprox): 1
  Tiled Convolution in software done
  Elapsed time = 4080300
Number of errors: 00000000
DONE riscv
```

**Figure 5.14:** 2D-Conv results in FPGA, sequential architecture.

For both the layers the test has been performed with a stride of 1 and no padding applied. The test also examines different PLMs sizes (in such a way to apply different tiling), two values for the number of PEs (4 and 8, corresponding to the PLMs' output channels Cout) and three precision configurations (16, 8 and 4 bits). All these parameters are reported in the eighth columns of Tab. 5.1 and Tab. 5.2. In particular, the dimensions of the PLMs are represented in the following order:

- INPUT/OUTPUT PLMs: Height (the same for both the PLMs) × Width (the same for both the PLMs) × Input Channels × Output Channels.
- WEIGHT PLM: Kernel Height × Kernel Width) × Input Channels × Output Channels.

The last columns of Tab. 5.1 and Tab. 5.2 show the output of the tiling function `get_tiling` in terms of number of tiles and their dimensions, while the first six columns of both tables report the test results. In particular, the first and second columns display the number of clock cycles taken by the sequential and hierarchical architecture of the accelerator respectively, to compute the layer. The third columns show the time taken by the processor to compute the layer in software. The fourth columns present the speedup achieved by the sequential accelerator (HW) compared to the processor (SW), where a negative value indicates that the accelerator takes less time to perform the computation. The fifth columns compare the speedup of the hierarchical accelerator against the processor. The sixth columns compare the speedup of the hierarchical accelerator against the sequential accelerator.

The results reported in both Tab. 5.1 and Tab. 5.2 show that the sequential accelerator performs better than the processor only when tiling is not applied, while the hierarchical accelerator obtains better results than the processor until 24

tiles are produced by the tiling function. Also, the hierarchical architecture takes on average 33% less time than the sequential architecture.

Therefore , it is possible to state that the accelerator performs at its best when implemented with a hierarchical architecture and the number of tiles is quite low, thus meaning that the tile sizes are large. In these scenarios, the accelerator optimizes the efficiency of each DMA transaction by transferring larger amounts of data per transaction and reducing the time spent configuring both the accelerator and DMA operations. The results indicate that larger PLMs are more advantageous, as they allow for fewer and larger tiles, which enhances performance. Conversely, when the number of tiles increases and their sizes decrease, the overhead from tiling and DMA transactions becomes more significant, negatively impacting the overall performance of the accelerator.

Furthermore, it is possible to observe how the reduction of the precision of ST multiplier used in the MAC units of the accelerator leads to no performance improvements. This is due to the fact that the function where clock cycles reduction occurs is the compute function; however, this function is much faster compared to the others, especially when compared to the load function, which takes most of the computation time. Therefore, the load function determines the final throughput of the accelerator and masks the benefits provided by the ST multiplier. Thus, the load function could be optimized in the future to make it faster.

Moreover, to further improve the performance, the tiling algorithm could be implemented in hardware, integrating it in the structure of the accelerator, in such a way to avoid the configuration of the accelerator for every tile to compute. Also, inside a hierarchical architecture all the functions could be still pipelined and run in parallel, meaning that the tiling algorithm and the other functions (config, load, compute and store) could be performed simultaneously. These improvements are left as future work.

| HW(basic) Time (clock cycles) | HW(hier) Time (clock cycles) | SW Time (clock cycles) | HW(basic) /SW speedup (%) | HW(hier) /SW speedup (%) | HW(hier) / HW(basic) speedup (%) | Layer and dimensions (HxWxCinx-CoutxK) | PLMs IN - OUT dimensions (HxWxCinxCout), PLM WEIGHT dimensions (KxKxCinxCout), Precision | Number of Tiles, Tile dimensions (HxWx-CinxCout), Input (HxWxCin) and Weight (KxKxCinx-Cout) Tiles sizes |
|---|---|---|---|---|---|---|---|---|
| 4070077 | 2512690 | 4080300 | -0.25 | -38.42 | -38.02 | Average Layer, 8x8x16x4x3 | 18x18x16x8, 7x7x16x8, 16 bit | 1, 8x8x16x4, 1024, 576 |
| 4065581 | 2509468 | 4081474 | -0.39 | -38.52 | -37.99 | Average Layer, 8x8x16x4x3 | 18x18x16x8, 7x7x16x8, 8 bit | 1, 8x8x16x4, 1024, 576 |
| 4064795 | 2509278 | 4080568 | -0.39 | -38.51 | -38.01 | Average Layer, 8x8x16x4x3 | 18x18x16x8, 7x7x16x8, 4 bit | 1, 8x8x16x4, 1024, 576 |
| 6879545 | 4246537 | 6460303 | 6.49 | -34.27 | -38 | Average Layer, 8x8x16x4x3 | 7x7x16x8, 7x7x16x8, 16 bit | 2, 8x8x8x4, 512, 288 |
| 6875892 | 4244225 | 6458654 | 6.46 | -34.29 | -37.99 | Average Layer, 8x8x16x4x3 | 7x7x16x8, 7x7x16x8, 8 bit | 2, 8x8x8x4, 512, 288 |
| 6873636 | 4242821 | 6458501 | 6.43 | -34.31 | -37.99 | Average Layer, 8x8x16x4x3 | 7x7x16x8, 7x7x16x8, 4 bit | 2, 8x8x8x4, 512, 288 |
| 14407788 | 8732043 | 11458256 | 25.74 | -23.79 | -35 | Average Layer, 8x8x16x4x3 | 7x7x4x8, 7x7x4x8, 16 bit | 8, 8x8x2x4, 182, 72 |
| 14405764 | 8730851 | 11441161 | 25.91 | -23.69 | -35 | Average Layer, 8x8x16x4x3 | 7x7x4x8, 7x7x4x8, 8 bit | 8, 8x8x2x4, 182, 72 |
| 21431147 | 13107735 | 13808540 | 55.20 | -5.08 | -36.5 | Average Layer, 8x8x16x4x3 | 7x7x4x8, 7x7x4x8, 4 bit | 24, 3x8x4x4, 96, 144 |
| 61577004 | 36544216 | 22056157 | 179.18 | 65.69 | -31.5 | Average Layer, 8x8x16x4x3 | 7x7x4x4, 7x7x4x4, 16 bit | 96, 3x8x1x4, 24, 36 |
| 37072304 | 22400183 | 19369581 | 91.39 | 15.65 | -34.5 | Average Layer, 8x8x16x4x3 | 7x7x4x4, 7x7x4x4, 8 bit | 48, 3x8x2x4, 48, 72 |
| 23635049 | 14455686 | 16012151 | 47.61 | -9.72 | -36.5 | Average Layer, 8x8x16x4x3 | 7x7x4x4, 7x7x4x4, 4 bit | 24, 3x8x4x4, 96, 144 |

**Table 5.1:** HW and SW performance results for an average DNN layer from [18]

| HW(basic) Time (clock cycles) | HW(hier) Time (clock cycles) | SW Time (clock cycles) | HW(basic) /SW speedup (%) | HW(hier) /SW speedup (%) | HW(hier) / HW(basic) speedup (%) | Layer and dimensions (HxWxCinx-CoutxK) | PLMs IN - OUT dimensions (HxWxCinxCout), PLM WEIGHT dimensions (KxKxCinxCout), Precision | Number of Tiles, Tile dimensions (HxWxCinxCout), Input (HxWxCin) and Weight (KxKxCinxCout) Tiles sizes |
|---|---|---|---|---|---|---|---|---|
| 252033329 | 148254917 | 34281627 | 635.18 | 332.46 | -30 | Last layer Mobilenet, 3x3x256x256x1 | 18x18x16x8, 7x7x16x8, 16 bit | 512, 3x3x16x8, 144, 128 |
| 252032897 | 148254645 | 34289202 | 635.02 | 332.37 | -30 | Last layer Mobilenet, 3x3x256x256x1 | 18x18x16x8, 7x7x16x8, 8 bit | 512, 3x3x16x8, 144, 128 |
| 252705022 | 148650012 | 34288558 | 637 | 333.53 | -30 | Last layer Mobilenet, 3x3x256x256x1 | 18x18x16x8, 7x7x16x8, 4 bit | 512, 3x3x16x8, 144, 128 |
| 489703181 | 283065422 | 37040061 | 1222.09 | 664.21 | -27 | Last layer Mobilenet, 3x3x256x256x1 | 18x18x16x4, 7x7x16x4, 16 bit | 1024, 3x3x16x4, 144, 64 |
| 489702988 | 283065311 | 37025544 | 1222.61 | 664.51 | -26.99 | Last layer Mobilenet, 3x3x256x256x1 | 18x18x16x4, 7x7x16x4, 8 bit | 1024, 3x3x16x4, 144, 64 |
| 489702702 | 283065145 | 37022301 | 1222.72 | 664.58 | -27 | Last layer Mobilenet, 3x3x256x256x1 | 18x18x16x4, 7x7x16x4, 4 bit | 1024, 3x3x16x4, 144, 64 |

**Table 5.2:** HW and SW performance results for the last layer of MobileNet

# Chapter 6

# Conclusions

The rapid expansion of big data applications, while offering vast potential for the advancement of machine learning, poses substantial challenges in terms of processing speed and scalability for traditional computing systems. Conventional von Neumann architectures, which separate processing and data storage components, are hindered by frequent data transfers between processors and memory, leading to performance bottlenecks and inefficiencies in energy consumption. These limitations are further exacerbated by the immense data volumes required by AI applications [1].

To address these challenges, domain-specific computing hardware platforms for AI applications, also called hardware accelerators, have emerged, designed to overcome issues such as the "memory wall" and "power wall" by optimizing data flow and minimizing energy-intensive operations [1].

In this thesis work, through High-Level Design and the Embedded Scalable Platform (ESP), starting from the sequential architecture of a 2D-Conv hardware accelerator realized in a previous work [2], the corresponding hierarchical architecture has been implemented. In the hierarchical architecture, the internal phases executed by the hardware accelerator (configure, load, compute and store) are performed in parallel. The architecture has been tested on an FPGA using a bare-metal application that performs a tiling algorithm realized in a previous work [2]. In particular, for the simulations, two different layers have been used and different types of tiling have been applied. The results show that this architecture improves the previous results of the sequential implementation [2] in terms of performance by around 33%. They also highlight that the accelerator achieves the best performance when the number of tiles to be processed is quite low (large PLMs), which implies that the number of DMA transactions get reduced, because more data can be moved per transaction. Moreover, the accelerator spends less time for configuring the DMA, and the processor needs to configure and invoke the accelerator less frequently. Furthermore, no performance improvements are obtained from the reduction of the precision of ST multiplier used in the MAC

units of the accelerator. This is due to the long execution time of the load function that determines the throughput of the accelerator, hiding the advantages provided by ST multipliers in the compute function.

These results lead to several potential directions and suggestions for expanding this thesis work in the near future, like:

- Modifying the tiling algorithm, making it able to adapt to the input layer's dimensions, generating at most a user-defined amount of tiles that allow to obtain the best performance from the hierarchical accelerator.
- Accelerating the loops in the load function using HLS directives.
- Balancing the workload between the load and compute functions to increase the accelerator's throughput by moving some of the operations currently handled by the load into the compute function. Otherwise, smaller (still pipelined) load functions could be created to distribute the workload and increase the accelerator's throughput.
- Implementing the tiling algorithm in hardware, integrating it in the hierarchical architecture of the accelerator, in such a way that it could be performed in parallel with the other functions of the accelerator, which will allow the processor to avoid to configure the accelerator for every tile to compute.

# Bibliography


[1] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang. «A Survey of Accelerator Architectures for Deep Neural Networks». In: *Engineerings* 6 (Mar. 2020), pp. 264–274 (cit. on pp. ii, 1, 2, 65).

[2] D. R. Bueno Pacheco. «Efficient Tiling Architecture for Scalable CNN Inference: Leveraging High-Level Design and Embedded Scalable Platform». MA thesis. Torino, Italia: Politecnico di Torino, Dec. 2023 (cit. on pp. ii, iii, 14, 17, 19–22, 24, 25, 29, 32, 41, 43, 47, 48, 50–52, 59, 65).

[3] P. Mantovani, D. Giri, G. Di Guglielmo, L. Piccolboni, J. Zuckerman, E.G. Cota, M. Petracca, C. Pilato, and L.P. Carloni. «Agile SoC development with open ESP». In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)* (Dec. 2020), pp. 1–9 (cit. on pp. ii, 7, 9, 11).

[4] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer. «Efficient Processing of Deep Neural Networks: A Tutorial and Survey». In: (Aug. 2017). URL: `https://arxiv.org/abs/1703.09039` (cit. on pp. 2–4).

[5] M. Ravi, A. Sewa, S. T.G., and S.S.S. Sanagapati. «FPGA as a Hardware Accelerator for Computation Intensive Maximum Likelihood Expectation Maximization Medical Image Reconstruction Algorithm». In: *IEEE Access* 7 (2019), pp. 111727–111735 (cit. on pp. 3–5).

[6] H. Jia and X. Zou. «An FPGA-Based Resource-Saving Hardware Accelerator for Deep Neural Network». In: *International Journal of Intelligence Science* 11 (Apr. 2021), pp. 57–69 (cit. on p. 3).

[7] M. Cassel Dos Santos et al. «A Scalable Methodology for Agile Chip Development with Open-Source Hardware Components: (Invited Paper)». In: *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)* (2022), pp. 1–9 (cit. on pp. 6, 7).

[8] D. Giri, K.-L. Chiu, G. Eichler, P. Mantovani, and L.P. Carloni. ««Accelerator Integration for Open-Source SoC Design». In: *IEEE Micro 41.4* (2021), pp. 8–14 (cit. on pp. 7, 8, 13).

[9] *Catapult® Synthesis User and Reference Manual*. Mentor, a Siemens Business. 2020 (cit. on pp. 7, 11, 43, 48, 53).

[10] L. Urbinati and M. R. Casu. ««Design-Space Exploration of Mixedprecision DNN Accelerators based on Sum-Together Multipliers». In: *2023 18th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)* (2023), pp. 377–380 (cit. on pp. 15, 22, 24).

[11] L. Urbinati and M. R. Casu. ««High-Level Design of Precision-Scalable DNN Accelerators Based on Sum-Together Multipliers». In: *IEEE Access* 12 (2024), pp. 44163–44189 (cit. on pp. 15, 16, 24, 25, 28, 36).

[12] A. Burrello, A. Garofalo, N. Bruschi, G. Tagliavini, D. Rossi, and F Conti. ««DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs». In: *IEEE Transactions on Computers* 70.8 (2021), pp. 1253–1268. URL: https://doi.org/10.1109%2Ftc.2021.3066883 (cit. on p. 18).

[13] Federico Perenno. «High-Level Design of 2D-Convolution Accelerators for AI Leveraging Embedded Scalable Platform (ESP)». MA thesis. Torino, Italia: Politecnico di Torino, 2022 (cit. on pp. 22, 23, 29, 33, 47, 52).

[14] Riccardo Capodicasa. «High-level design of a Depthwise Convolution accelerator and SoC integration using ESP». MA thesis. Torino, Italia: Politecnico di Torino, 2022 (cit. on pp. 22, 29, 47, 52).

[15] *Guide – How to: design an accelerator in C/C++ (Mentor Catapult HLS)*. URL: https://www.esp.cs.columbia.edu/docs/mentor_cpp_acc/mentor_cpp_acc-guide/ (cit. on pp. 22, 24, 29, 30).

[16] Columbia University - System Level Design Group. *ESP Accelerator Specifications*. Accessed: 2024-10-01. URL: https://www.esp.cs.columbia.edu/docs/specs/esp_accelerator_specification.pdf (cit. on p. 22).

[17] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer. «A Survey of Quantization Methods for Efficient Neural Network Inference». In: *Low-Power Computer Vision: Improve the Efficiency of Artificial Intelligence*. 1st. New York, NY, USA: Chapman and Hall/CRC, 2022. Chap. 1.2.12, pp. 14–17. DOI: 10.1201/9781003162810 (cit. on p. 25).

[18] L. Urbinati and M. R. Casu. ««A Reconfigurable 2D-Convolution Accelerator for DNNs Quantized with Mixed-Precision». In: *Applications in Electronics Pervading Industry, Environment and Society* (2023). Ed. by R. Berta and A De Gloria. Cham: Springer Nature Switzerland ISBN: 978-3-031-30333-3, pp. 210–215 (cit. on pp. 60, 63).