POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

High-Level Design of 2D-Convolution Accelerators for AI Leveraging Embedded Scalable Platform (ESP)

Supervisors

Prof. Mario Roberto CASU

Dr. Luca URBINATI

Candidate

Federico PERENNO

Academic Year 2021-2022

Summary

Today, Artificial Intelligence (AI) is everywhere and it has made many technological applications more efficient and reliable. Convolutional Neural Networks (CNNs) are at the foundation of the vast majority of AI applications. These types of networks can be extremely accurate, but precision comes at a high computational cost due to the many mutliply-and-accumulate operations between input and weight tensors. Loosely-coupled hardware accelerators supported by general-purpose processors are an effective way to speed up the computation of CNNs.

Thus, the goal of this thesis is to design accelerators that perform a specific type of convolution known as 2D convolution. The design process leverages High-Level Synthesis (HLS) and the Embedded Scalable Platform (ESP) tool, which simplifies accelerators design and their integration into heterogeneous System-On-Chips (SoCs).

This manuscript describes how neural networks work, paying particular attention to CNNs. In the first section it explains the different layers that usually compose these types of neural networks focusing on convolution since it is the most common operation in CNNs, as well as the most computationally demanding. The most successful ConvNet models for image classification are also taken into consideration and their architectures and performances are quickly compared.

It continues showing how hardware accelerators can be used to perform specific functions that can be executed in parallel with other operations performed by the processor core. It also discusses on how accelerators can be easily designed and synthesized leveraging high-level synthesis tools, such as Catapult HLS.

Then, it illustrates the open source Embedded Scalable Platform (ESP) developed at Columbia University. ESP combines a flexible design methodology with a scalable architecture. This platform accommodates various Computer-Aided Design (CAD) tools and design flows. In addition, it facilitates the integration of different hardware blocks in an SoC which otherwise would be a very challenging and longer task.

After this preliminary introduction, the thesis addresses the design of the 2D-convolution accelerators. Each accelerator is integrated into an SoC which is prototyped on a proFPGA XC7V2000T FPGA with a single external DDR3 memory module. Different implementations of the same accelerator are proposed.

The first implementation is a standard 2D convolution with a 18×18×32 input feature map tensor and a 7×7×32×32 weight tensor and it allows to get a latency of 145.8 ms and an estimated area of 83,268.3 μm^2. For this architecture, the data reading phase, the convolution operation and the data write back phase are performed in a sequential way. Nonetheless, 2D convolution is computed 95.65 % faster by our accelerator than it is by the SoC's processor core.

The second implementation exploits HLS directives to perform these three different phases in a pipelined way using a hierarchical design. However, these phases are performed in parallel only when dealing with multiple independent batches of data. When processing 16 batches, latency is reduced by 0.91 % at the cost of a RAM block usage increase of 11.49 % with respect to the standard approach.

The third implementation employs a sliding window architecture which uses line buffers to perform the accelerator phases in parallel when processing a single input batch. This optimized architecture allows to obtain a final latency reduction of 0.90 % while, regarding the FPGA resource usage, block RAM usage is decreased by 13.62 % at the cost of only 2.41 % additional slice logic. Thanks to this final implementation the read and write operations are now transparent to the actual 2D-convolution kernel. However, the accelerator is heavily compute bound. Thus, to fully appreciate the memory optimizations introduced in this thesis, future work should take care

of parallelizing the datapath to achieve greater latency reduction against the first and second implementations.

This thesis contributes to expand the documentation about the Catapult design flow of ESP with a 2D-convolution accelerator and can also be used as a detailed tutorial by future hardware designers for designing new accelerators using Catapult HLS and ESP.

Acknowledgements

I wish to express my gratitude to my family and friends for their constant support throughout the years.
On top of that, I would like to thank my supervisors, Dr. Luca Urbinati and Prof. Mario Roberto Casu for their precious advice and guidance.

Table of Contents

List of Tables

List of Figures

Listings

Chapter 1

Neural Networks

1.1 Deep Neural Networks

Artificial Intelligence (*AI*) has taken the world by storm by making countless technological applications more efficient and reliable and is regarded by some experts as the *new electricity* because of the great impact it is already having upon industries [1].

Deep Neural Networks (*DNNs*) are one of the branches of AI which is evolving and advancing more rapidly. DNNs are everywhere: from communication to transportation, from robotics to healthcare. *DNNs* have been particularly impactful in the last decade because of the availability of high computational power and huge amounts of data to train models. It comes to no surprise that most of their applications are related to images and videos, such as object detection, image classification and segmentation, but also audio. Indeed, speech recognition has significantly improved in accuracy over the last years [2].

The name "*neural networks*" suggests a close relationship with the brain, but, in reality, the main organ of the nervous system has only served as an inspiration rather than something to emulate. It is made of *neurons*, connected by *synapses*, whose key characteristic is to scale the transmitted signals by certain factors called *weights*. The human brain is believed to learn by meticulously adjusting these scaling factors in response to learning stimuli and, in a similar way, neural networks are formed by neurons receiving weighted signals. In addition to this, neurons do not simply output the

incoming signals, but perform a non-linear function on them, so that the neural network is capable of modelling complex mathematical functions.

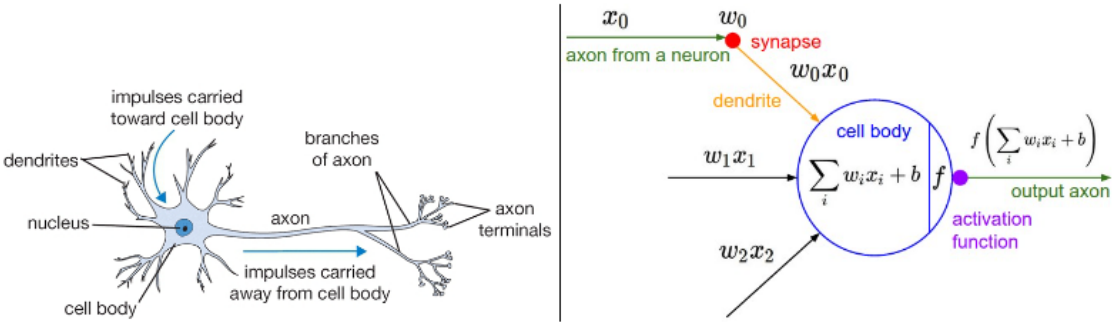

Figure 1.1: Intuitive comparison between a biological neuron and its mathematical model. Source: [3].

Neural networks are organized in a multi-layered structure: for each *layer* there are multiple nodes (neurons) connected to other nodes in different layers. Input data are organized in the first layer (*input layer*) and processed throughout some other layers (*hidden layers*) until the final result is computed by the last layer (*output layer*), as shown in Fig. 1.2. Neural networks are called "deep" when they have multiple hidden layers (more than 3-4) and can provide superior accuracy compared to *shallow neural networks*, which only dispose of a single hidden layer, at the cost of higher computational complexity.

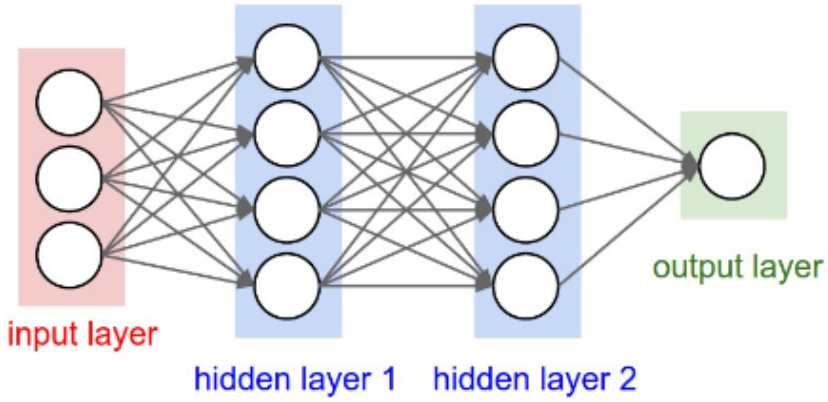

Figure 1.2: Example of a Neural Network. Source: [3].

The output of each neuron is referred to as an *activation*, while each connection (synapses) is a weight, whose value is determined during a learning time called *training*, by exploiting *ad-hoc* algorithms such as the *gradient descent*.

It is best practice to initialize weights to random values, resulting in wrong neural network's outputs at the beginning of the training phase. However, weights are adjusted by the gradient descent algorithm, which tries to continuously reduce the difference, referred to as *cost* or *loss function* (L), between the actual outputs and the expected theoretical results. This algorithm takes the gradient of the loss function over the weight, multiplies it by the *learning rate* (α), and subtracts it to the original weight value (w):

$$w = w - \alpha \cdot \frac{\partial L}{\partial w} \tag{1.1}$$

This process is iterated until the loss converges to its absolute minimum, as shown in Fig. 1.3.
When the loss is at the minimum, all weights are set to their optimal values.

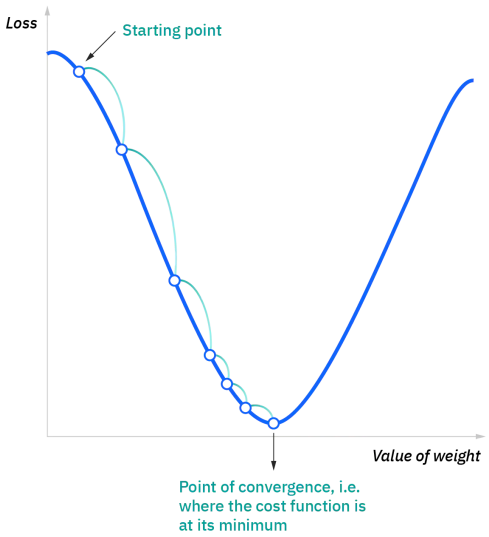

Figure 1.3: Visual representation of the gradient descent algorithm.
Source: [4].

The training phase is crucial to calibrate the network's weights in the optimal way, but it is also very lengthy, since it requires many forward and

backward passes of a hefty portion of the samples (*training samples*) through the network. After the training process, the network is ready to be used for predicting the outputs of new unseen data. This process is called *inference* and it only includes the forward passes.

The most used *activation functions* are *sigmoid*, *softmax*, and *rectified linear unit* (*ReLU*), the *de-facto* standard for (deep) neural networks.

Given the input x, the sigmoid function $\sigma(x)$ converts each real input into a value between 0 and 1, converging, for the most part, to the two extremities:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1.2}$$

The softmax activation function serves a similar, but wider purpose. It converts the real elements of the vector it receives as an input into a smooth probability distribution. Softmax is often used as last layer in a *neural network* to normalize the results:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{1.3}$$

The ReLU function can be applied to the output of a linear transformation to make it nonlinear. However, since it is a piece-wise linear function, it preserves many of the properties that make linear models easy to optimize. Given the input z:

$$g(z) = max\{0, z\} \tag{1.4}$$

A visual comparison between the sigmoid and ReLU functions is shown in Fig. 1.4.

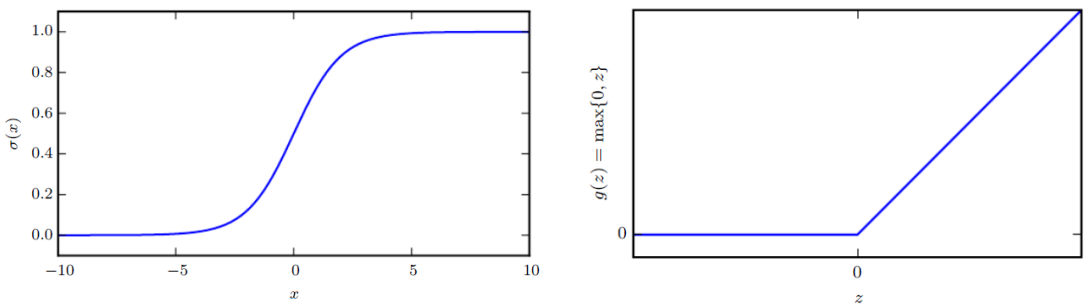

Figure 1.4: Comparison between sigmoid (left) and ReLU (right) activation functions. Source: [5].

1.2 Convolutional Neural Networks

Often abbreviated as *CNNs* or *ConvNets*, *Convolutional Neural Networks* use a different approach for processing data.

In *fully-connected* layers each neuron applies a linear transformation to the vector it receives as an input through a matrix of weights. Then a non-linear transformation is applied to all the elements, exploiting activation functions.

When neural networks which are solely composed of fully-connected layers have to process a big input vector, the number of weights to be trained can grow very large. This results in a huge amount of time spent training the network and it affects its performance.

In order to reduce the total number of parameters in the neural network, a different way of treating the input set has been adopted. Instead of applying weights directly to all neurons, *convolutions* are performed at each layer to generate progressively higher-level abstractions of input data, called *feature maps*.

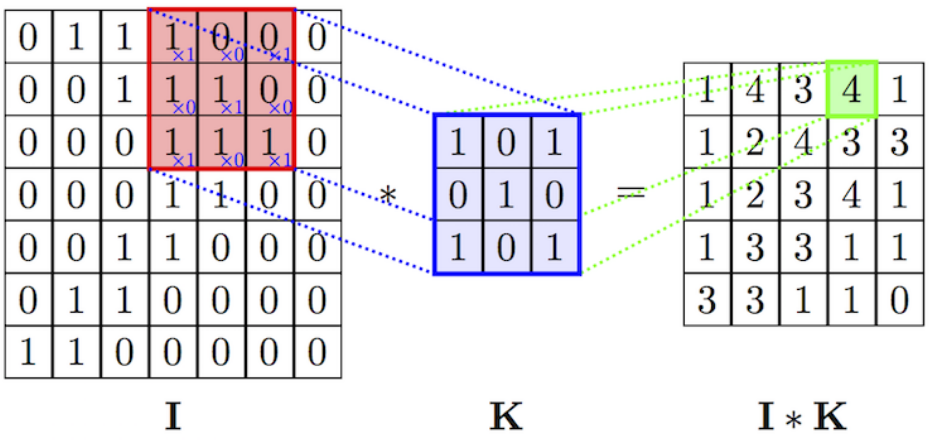

Figure 1.5: Convolution between two matrices. Source: [6].

The mathematical operation known as convolution is performed between two *tensors* (generalizations of vectors and matrices that can be seen as multidimensional arrays). The first tensor either corresponds to the CNN's input data or to a set of feature maps. The second one is a trainable set of

parameters known as a *filter*. In general, the input set is split over multiple *channels*, while the filter is composed of multiple distinct *kernels*. *Multiply-and-accumulate* operations are performed between the two tensors until a third one is generated as a result. Fig. 1.5 shows the convolution between an input channel and a single kernel. These two example matrices can be seen as unidimensional tensors.

Using multiple filters at each layer results in having multiple feature maps. Their name is due to the fact that each filter is specifically trained to detect a different feature. Fig. 1.6 shows how various unique features can be obtained from the same image and how these can be used to get progressively higher-level abstractions of the input data.

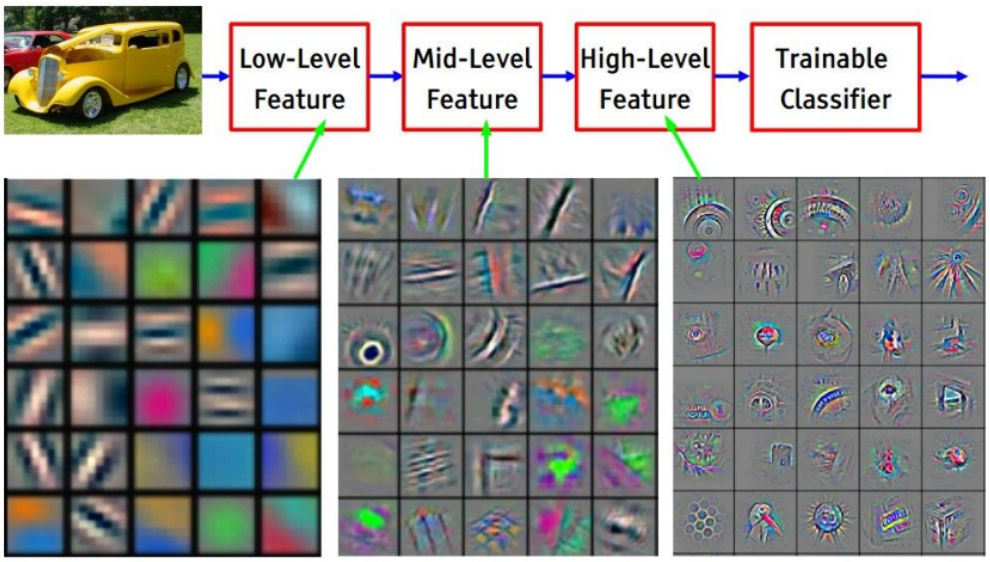

Figure 1.6: Different features obtained from the same image. Source: [7].

The advantages of adopting convolutions to build a neural network rather than solely using fully-connected layers are mainly [8]:

- **Parameter sharing**, which implies that a feature detector used in a given part of the neural network might be useful elsewhere. Hence, the same kernel can be re-used with no need to duplicate it and to re-train it each time;
- **Sparsity of connections**, given by the fact that for each layer, the output depends only on a small number of inputs.

Furthermore, ConvNets generally guarantee *translation invariance*, meaning that when shifting, for example, an image by a few pixels, the neural network returns the correct result, with no need to repeat the learning process. However, gradient descent is still used to train the parameters, which are now the elements of the filters.

In ConvNets, when the convolution between two tensors is performed, the output generally has smaller vertical and horizontal dimensions than the input one. This means that, after multiple consecutive convolutional layers, the output is inevitably shrunk to a very small size, at a rate depending on the kernels' dimensions. Moreover, not all the elements of the input tensor have the same relevance: the values in the corners affect the output less than those at the center because they are involved in less convolutions. One way to overcome these issues is *padding*. This technique is used to increase the vertical and horizontal dimensions of the input by increasing the border of the input by a certain amount (p) of *zero-filled layers*, depending on the kernel size (k):

$$p = \frac{k-1}{2} \tag{1.5}$$

Padding allows to get an output the same size of the input one, as exemplified in Fig. 1.7.

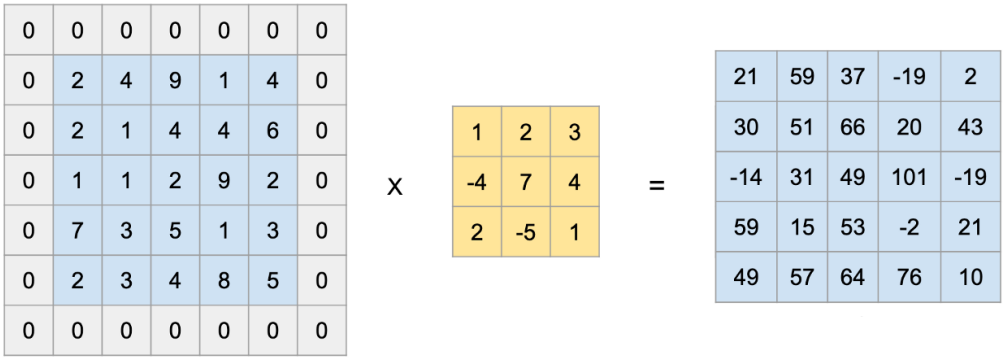

Figure 1.7: Convolution between a kernel and an input channel with padding. Source: [9].

There are other instances, however, in which it might be needed to compress the input data. The convolution operation can be performed in such a way that the output becomes even more shrunk than it would have

originally been: each kernel is applied over a certain portion of an input channel and the convolution operation is performed, then the kernel is shifted by a certain amount of steps and the next output value is computed. This number of steps is called *stride* and the larger this value is the smaller the output dimensions will be. Two examples are reported in Fig.1.8 with stride=1 and stride=2 at the top and at the bottom, respectively.

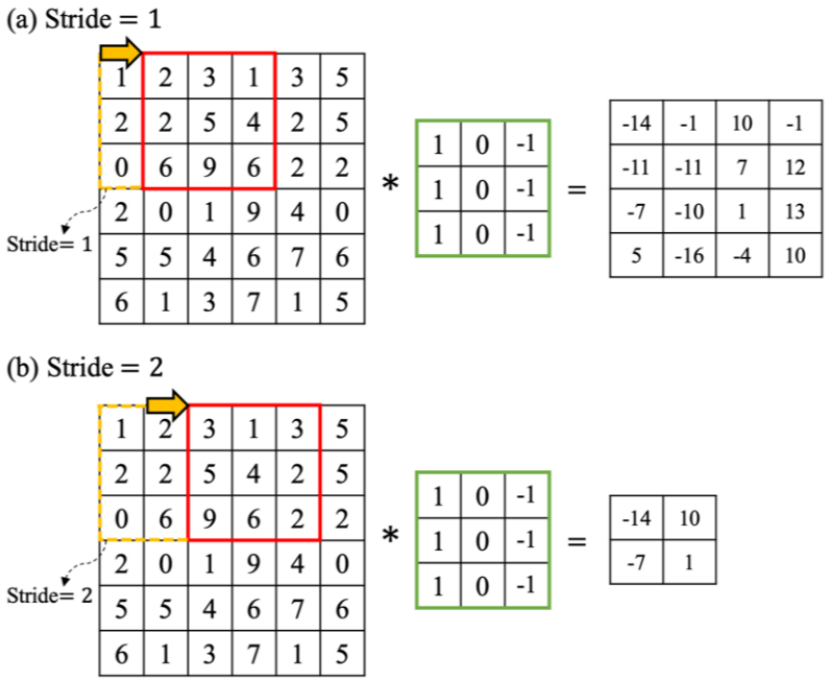

Figure 1.8: Comparison between two convolutions with different stride values. Source: [10].

Although convolutional layers alone are enough to build a functioning neural network, they are usually combined with other layers. The most commonly used are [2]:

- **Fully-connected**: each neuron applies a weight matrix to all the elements of the feature maps;
- **Nonlinear**: such as the ReLU and sigmoid functions, this layer is usually applied after convolutional or fully connected layers to each element of the feature map tensor. They are used to introduce nonlinearity in the ConvNet;

- **Pooling**: it reduces width and height of the feature map tensors to speed up computation and to make the features themselves more robust [11]. *Max* and *average* pooling are the main implementations. The pooling operation selects either the maximum or the average value in a fixed-size sub-region of the feature map tensor. These regions' dimensions are given by the *pooling size* parameter. Fig. 1.9 shows the pooling operation with size=2 on a 4×4 example matrix;
- **Normalization**: it is used to constrain the distribution of the features across different layers, which can help to improve accuracy and to speed up the training process.

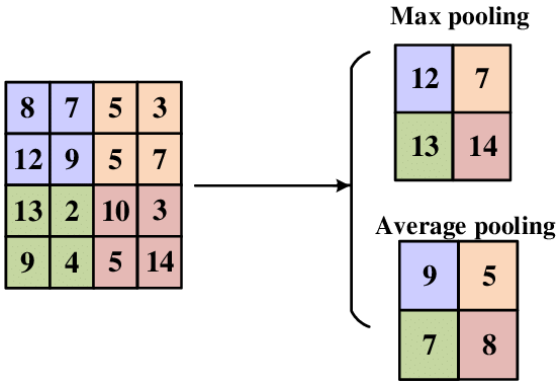

Figure 1.9: Example of Max and Average Pooling. Source: [12].

1.2.1 Case studies

Countless CNN models have been developed in the past few decades. It is important to take a look at some of the most successful ConvNet models for image classification in order to get an idea on how layers can be structured and what are the most effective ways to design convolutional neural networks.

The *LeNet* model was first suggested as far back as 1989 and it is considered a precursor of the modern CNNs. It is used for digit classification, hence for recognizing hand-written numbers between 0 and 9 from a small gray-scale image. The most well-known version is *LeNet-5* (1998): this particular model can identify digits with good accuracy from a gray-scale picture made of 32×32 pixels using only 5 layers, 3 of which are convolutional and 2 of which are fully-connected. Additionally, it features two *subsampling*

layers which perform various operations, including average pooling and the sigmoid function. It should be noted that padding, max pooling and ReLU are not adopted yet [13].

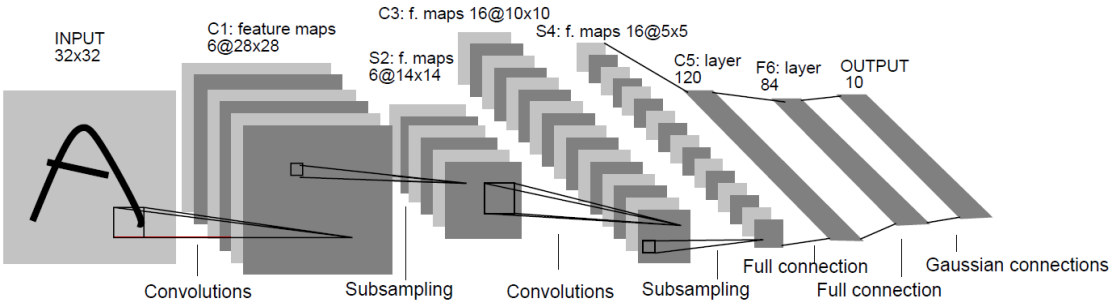

Figure 1.10: LeNet-5 architecture. Source: [13].

Starting from 2012, an annual challenge has been run by the *ImageNet* project, where contestants could train and test their CNNs on a given dataset containing more than 14 million images, trying to achieve the lowest amount of errors.

The ConvNet that won the first edition of the contest was *AlexNet*. Despite owing most of its success to the fact that it was trained for the first time on *Graphic Processing Units (GPU)* rather than *Central Processing Units (CPU)*, its architecture is also remarkable. It has 8 layers, 5 convolutional and 3 fully-connected, as shown in Fig. 1.11.

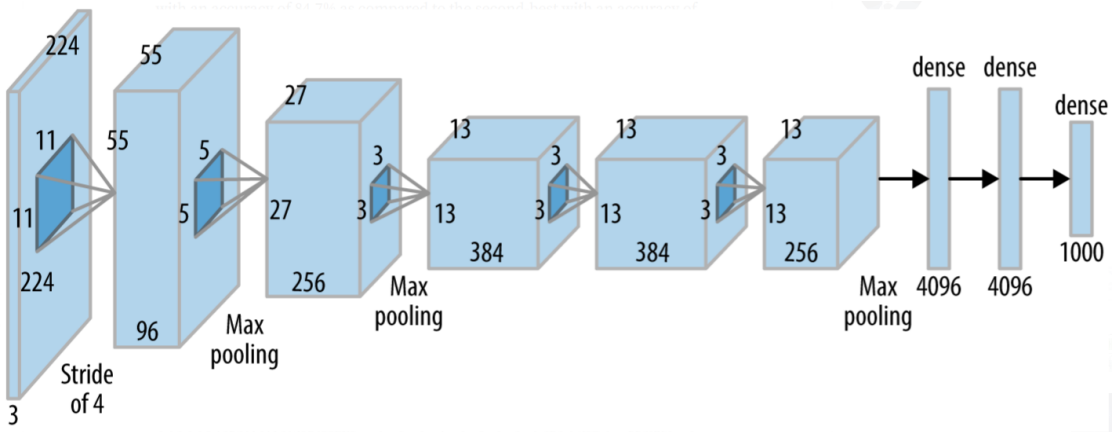

Figure 1.11: AlexNet architecture. Source: [14].

Although it may look quite similar to the LeNet model, AlexNet is much larger, since it processes 227×227 RGB images and uses many kernels with different sizes. It also uses max pooling, stride values different from one, ReLU activation functions and softmax to determine the output [14].

Two noteworthy models entered the ImageNet contest in 2014: *VGG-16* and *GoogLeNet*. The former is a 16-layer network with 13 convolutional layers and 3 fully-connected ones, so twice as deep as the 2012 winner. It is particularly appealing because of its simplicity. Indeed, the VGG model uses very few blocks repeated all over the network. In particular, it uses 3×3 kernels with a stride=1 and padding, and it reduces the height and width of feature maps by exploiting 2×2 max pooling [15].

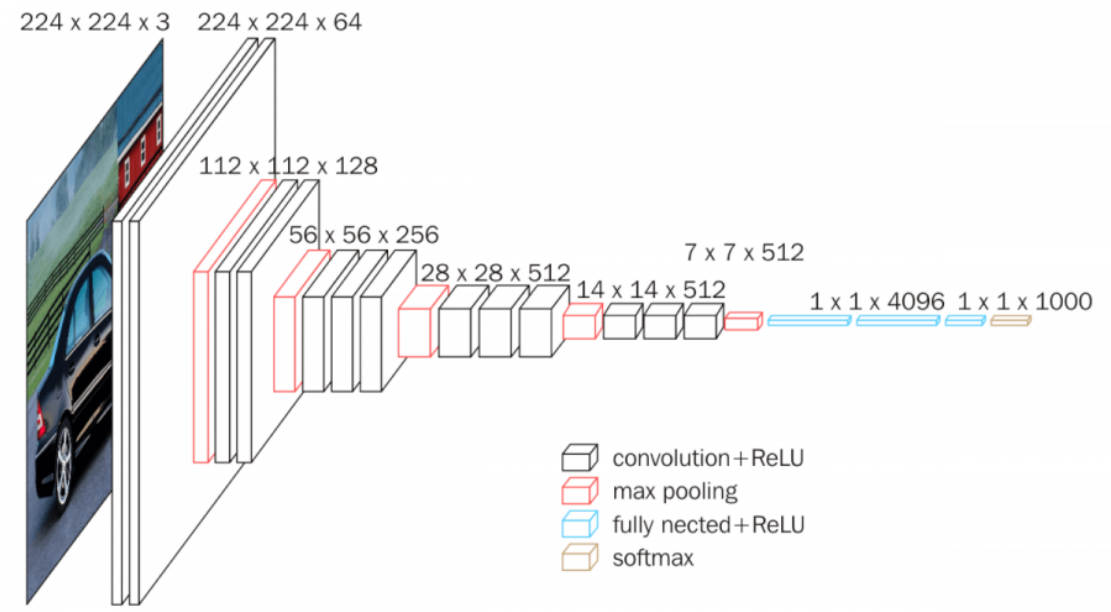

Figure 1.12: VGG-16 architecture. Source: [15].

On the other hand, the GoogLeNet model uses a different approach to process input data: it has 22 layers mainly organized as *inception modules*, meaning that the CNN is not simply composed by single serial connections, but of parallel ones as well. Each inception block is 2 convolutional layers deep and is structured as shown in Fig. 1.13 [16].

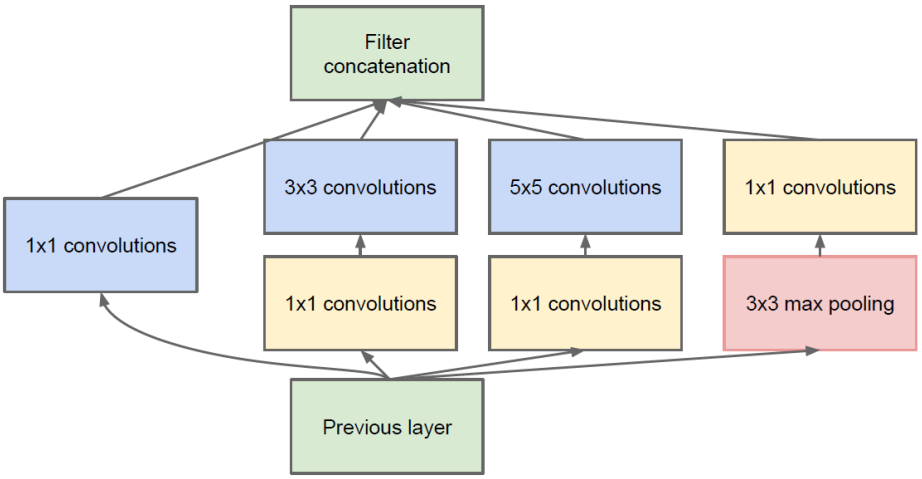

Figure 1.13: Inception module. Source: [16].

Finally, let us consider the CNN that surpassed for the first time human-level accuracy with a top-5 error rate of 5% on ImageNet: *ResNet*.
As neural networks become very deep, they do not get more accurate. On the contrary, they face the problem of *vanishing/exploding gradient*: in so-called *plain networks*, as errors go through back-propagation, the gradient is gradually reduced and the ability to update the weights in the early layers is compromised.

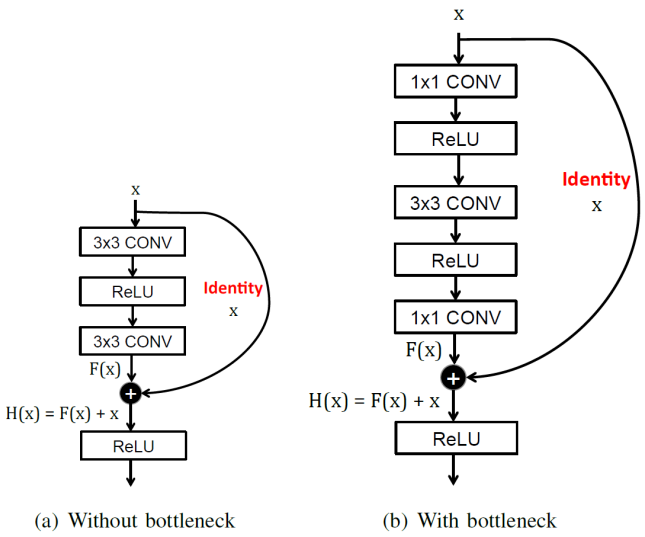

Figure 1.14: Residual block examples. Source: [17].

To overcome this issue, ResNet is based on *residual blocks* where the last layer in a block receives two inputs: one coming from the output of the previous layer (as usual) and the other coming from the input of the same block. This second connection is called a *shortcut* and is not associated to any weight [17].
Thanks to these blocks, extremely deep networks can be implemented without having to face the vanishing/exploding gradient problem.

Overall, performance improved over time and went from the *top-5 error rate* of 16.4% achieved with AlexNet to the astonishing 5.3% obtained by *ResNet-50*, a residual network with 50 layers. Complexity was also reduced by adopting innovative solutions such as the inception modules and the residual blocks. A comparison of the performance of the CNNs presented in this chapter is reported in Tab. 1.1 for the ImageNet dataset .

Metrics	**AlexNet**	**VGG-16**	**GoogLeNet**	**ResNet-50**
`Top-5 error`	16.4	7.4	6.7	5.3
`Input Size`	227×227	224×224	224×224	224×224
`# of CONV layers`	5	13	21	49
`# of FC Layers`	3	3	1	1
`Total Weights`	61M	138M	7M	25.5M
`Total MACs`	724M	15.5G	1.43G	3.9G

Table 1.1: Convolutional Neural Networks: case studies. Performance based on the ImageNet dataset.

Chapter 2

Hardware Accelerators

An essential component of both convolutional and fully-connected layers are multiply-and-accumulate operations, which can be executed in parallel to achieve high performance. Various techniques can be adopted to improve parallelism, including [2]:

- **Temporal architectures**, frequently used on CPUs and GPUs. They use centralized control for many different *Arithmetic-Logic Units* (*ALUs*) that do not exchange data between each other. In these particular architectures, ALUs only fetch and send data from/to the memory;
- **Spatial architectures**, often implemented in *Application-Specific Integrated Circuit* (*ASIC*) and *Field-Programmable Gate Array* (*FPGA*) designs, which rely on *hardware accelerators*. In these architectures, ALUs are all in the same processing array and can send/receive data directly to/from one another.

Modern CNNs need to perform an enormous amount of computations and memory accesses, which cause great latency and energy consumption. Many applications require *real-time processing*, but it can rarely be met by CPU and GPU implementations [18].

Hardware accelerators, on the other hand, aim at incrementing the computational speed by employing *ad-hoc* custom hardware, designed to implement a particular algorithm. They focus on performing specific functions and, if well designed, they are more efficient than software applications running on general-purpose CPUs and even GPUs.

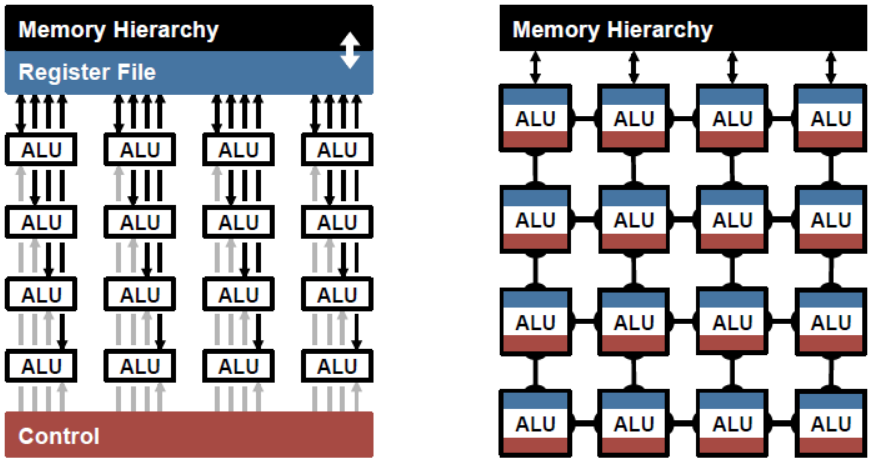

Figure 2.1: Comparison between temporal (on the left) and spatial (on the right) architectures. Source: [2].

A generic computer architecture that contains hardware accelerators is shown in Fig. 2.2.

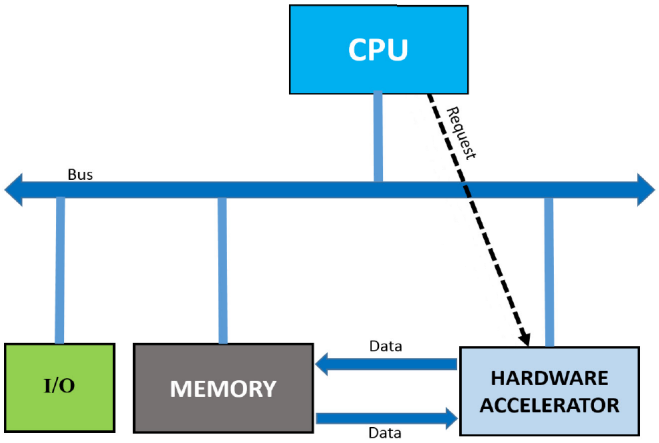

Figure 2.2: Hardware acceleration architecture. Source: [19].

This hardware acceleration architecture has the benefit of leaving the CPU free to execute other tasks after it has offloaded a specific computation to the hardware accelerator, which can be seen as a co-processor. The CPU has the only tasks of setting up the configuration of the accelerator, namely the starting memory address from which data can be read, and passively waiting for the interrupt of job completion (e.g. a "done" signal) coming from

the accelerator. After receiving this signal, the CPU can read the results of the accelerator's computation from the main memory. This architecture yields an improvement in performance when the overhead resulting from the communication between the CPU and the accelerator costs less than performing the computation on the CPU itself.

Since the focus of this thesis is to accelerate 2D-Convolution, from now on we will consider the hardware implementation of this algorithm.

In CPU and GPU implementations each MAC unit needs to read from the memory three times (filter's weight, feature map element and partial sum) and write on it once (updated partial sum). Accelerators, on the other hand, introduce many layers of local memory hierarchy, providing low-cost data accesses thus improving energy efficiency and performance.

Additionally, programs on FPGAs run on a *bare fabric*. This implies that data can be sent straight to the pins of the FPGA board using peripherals which help to reduce latency. In GPUs, on the other hand, data are delivered on standard buses, such as *Universal Serial Bus* (USB) or *Peripheral Component Interconnect* (PCI), which are subject to the Operating System (OS) resource scheduling. This makes FPGAs more suitable for designing hardware accelerators, because they guarantee deterministic performance and latency. ASICs are also a reliable alternative because they are extremely efficient in energy and area, but when compared to FPGAs, they cost more and only high production numbers can cover the high non-returning engineering costs, which make them sometimes less desirable [19].

Moreover, FPGAs are particularly suitable for making high-performance computations, since the reduced latency time makes up for the lower clock frequency. As an example, the `Ultra Scale` FPGA has a raw computational power of 38.3 Tera Operations per second (TOPS) at base frequency. However, a powerful `Nvidia Tesla P40` GPU performs a similar amount of computations (40 TOPS), but consumes twice the energy [19].

A generic FPGA-based design of a hardware accelerator for CNNs is reported in Fig. 2.3. It should be noted how there are buffers to cache inputs, coefficients to facilitate data re-use and blocks that perform specific computations.

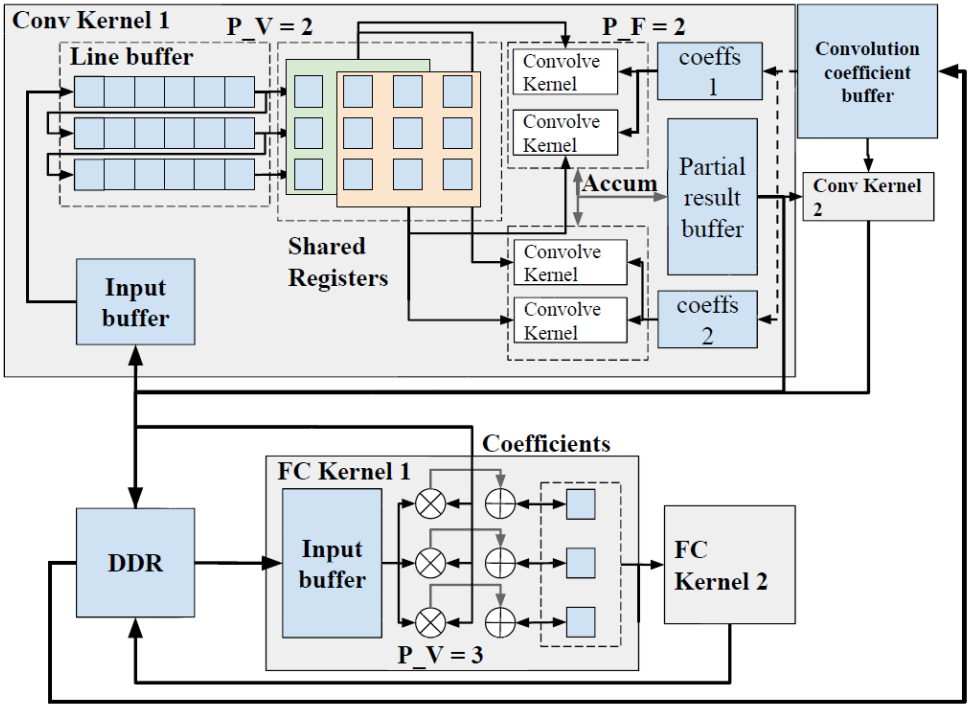

Figure 2.3: Generic FPGA-based hardware accelerator for CNNs. Source: [20].

2.1 High Level Synthesis

FPGAs can be programmed using specialized languages known as *Hardware Description Languages* (*HDLs*). They allow FPGAs to be designed at the *Register-Transfer Level* (*RTL*), making them incredibly flexible. FPGAs' adaptability and reconfigurability are qualities that make them appealing. However, HDLs have rigid syntax and operate at a very low level: errors are extremely likely and hard to fix. For this reason, HDL-programmed devices can be difficult to debug; this greatly increases design time thus compromising productivity, especially for complex applications.
Alternatively, at the beginning of this century, a high-level design process has been developed to simplify hardware design: *High Level Synthesis* (*HLS*). Also known as *algorithmic synthesis* or *behavioral synthesis*, HLS takes a high-level description of a design and compiles it into an RTL implementation according to user-specified constraints. The high-level property of HLS is due to two main aspects [21]:

- **Abstraction**: the description of the HLS design is an untimed (or partially timed) dataflow. This means that the operations that must be performed within a clock cycle do not have to be meticulously planned at design-time;
- **Specification Language**: concise and reusable design can be implemented thanks to the use of high-level programming languages such as `C`, `C++`, and `SystemC`. These *C-like* languages present useful features such as arrays, structures, loops and pointers.

The output of the HLS flow consists of multiple elements: the RTL implementation, which includes the RTL netlist automatically generated from the high-level description of the design, and some additional parameters required to synthesize it using regular logic synthesis tools. HLS also generates reports on hardware costs and performance bottlenecks. Additionally, simulation testbenches and scripts are generated to speed up the development and the debugging of the design.

The HLS output is generated based on user-specified design constraints, which can be related to: the *target hardware*, so that datapath costs can be estimated; the *performance goal*, based on which cycle-level timing is adjusted; the *memory architecture* which describes how arrays are mapped to memory interfaces; other *interfaces*, so that the generated netlist can be easily integrated with external hardware.

While it is not always true that HLS is superior to manual HDL, it has the great advantage of giving a lot of freedom to the designer who can just modify a few lines of high-level code to adjust the hardware design. Moreover, HLS acts as a bridge between hardware and software designers, since the former have the flexibility to design hardware at a high-level, while the latter can work on improving HLS tools, thus minimizing design time.

It becomes also quite simple to test the generated netlist, since the high-level C-like testbench can be used for both algorithmic and design verification. Because of its flexibility, HLS makes it relatively easy and quick to experiment with different algorithms and architectures, creating vast design-space explorations by simply adjusting the various HLS knobs the tool provides. The most common optimizations, also known as *directives*, are the following [19]:

- **Loop Unrolling**: it allows to replicate the content of *for* loops allowing parallel computation. This leads to maximum performance, at the expense of area;
- **Array Partitioning**: usually applied together with the previous directive, it divides large arrays into smaller ones in order to access them in parallel and to feed the unrolled datapath without memory bottlenecks;
- **Pipelining**: loops subjected to this directive are forced to start their next iteration after a fixed number of clock cycles (called *Initiation Interval*), even before the end of the current iteration. This implies that the operations inside these loops are pipelined with input and output registers, allowing parallel and faster execution with very low area overhead (only due to registers and control logic).

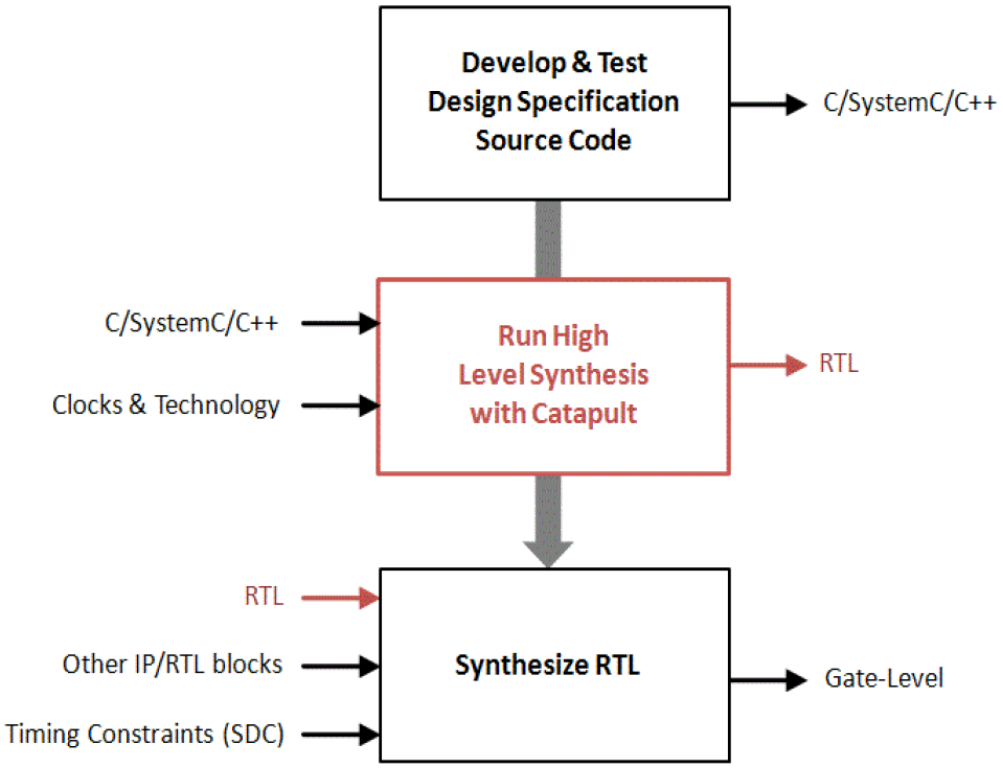

Figure 2.4: Catapult's high-level design flow. Source: [22].

There are many high-level synthesis tools. `Catapult HLS` by `Mentor Graphics` is the one we used in this thesis to design our accelerator: it is able

to "synthesize interfaces, data structures and loops to an ASIC or FPGA technology and produce an optimized RTL implementation", as shown in the red block in Fig. 2.4 [22]. The main output files of Catapult are: the resulting RTL implementation of the HLS design that can be simulated with logic synthesis tools such as `Synopsys Design Compiler`; HDL files (VHDL and Verilog) of the synthesized design; RTL simulation scripts; the log of the HLS commands and directives applied to the HLS code in input. It also supports a *Graphical User Interface* (GUI) with command line and scripts to easily set up the synthesis process.

Chapter 3

Embedded Scalable Platform

In recent years, *CMOS* scaling has been slowing down [23] and modern computing systems are relying more and more on *Systems-on-Chip* (*SoC*) designs which integrate many hardware components. Whether it is for *AI*, *Internet of Things* (*IoT*) or *multimedia* applications, modern SoCs couple general-purpose processors with specialized hardware accelerators. Therefore, their architecture is heterogeneous.

We have seen how the high-level synthesis flow facilitates the design of an accelerator. Nonetheless, integrating multiple heterogeneous components, especially accelerators, in a scalable way can be quite challenging. For this reason, an open source SoC research platform, known as `Embedded Scalable Platform` (*ESP)*, was created as the result of many years of research and teaching at the `Columbia University` in New York City [24].

ESP combines a flexible design methodology with a scalable architecture [25]. It is flexible because it accommodates various *Computer-Aided Design* (*CAD)* tools and design flows; it is scalable because it allows to easily reuse previously designed accelerators. It is open source since all its hardware components are without copyright and so free to be accessible and usable in anyone's projects.

As just said, multiple design flows are supported by the ESP methodology. This allows to create many heterogeneous components from different abstraction levels:

1. Components can be developed following the traditional RTL design flow from specifications in HDLs such as `SystemVerilog` or `VHDL`;

2. Accelerators can be designed from high-level descriptions written in C-like programming languages, leveraging commercial HLS tools like Catapult, together with the ESP automation tools;

3. For some specific *machine learning* (*ML*) applications, the HLS code of accelerators can be generated from network models in `PyTorch`, `Keras` and `TensorFlow`.

New accelerators can be integrated, as well as third-party *Intellectual Property* (*IP*) blocks that implement the ASCII protocol, like the `Ariane RISC-V` processor core or the `Nvidia Deep Learning Accelerator` (*NVDLA*).

ESP has a GUI that guides the designer through the interactive floor-planning of the SoC, which can also be quickly prototyped on FPGAs thanks to ESP's push-button flows.

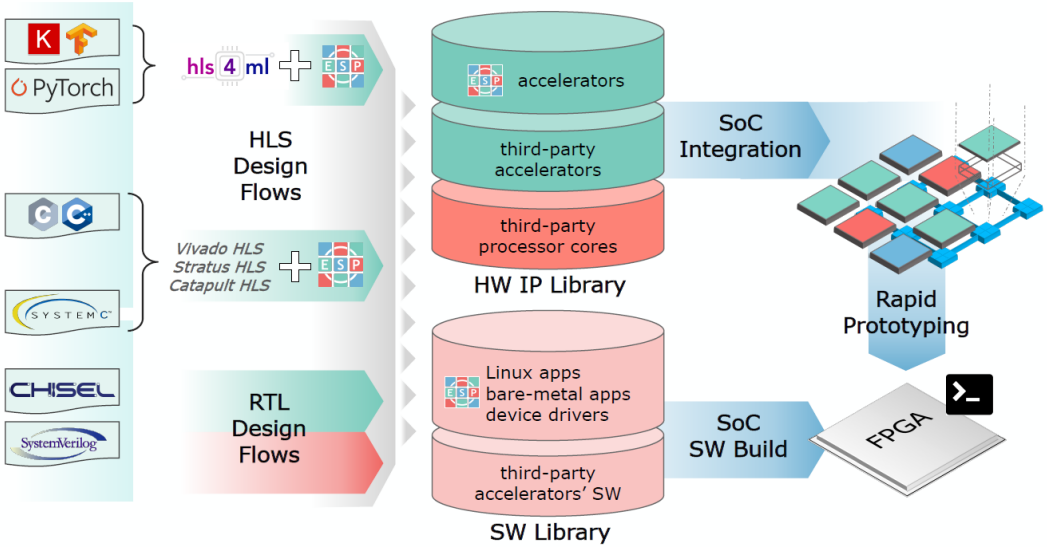

Figure 3.1: Design flows supported by ESP. Source: [25].

Tiles are at the foundation of the ESP architecture and their number can be selected at design time depending on the application. They may contain: a processor core, a loosely-coupled accelerator, a memory channel and some other peripherals. A multi-plane *Network-on-Chip* interconnects the various

tiles. Fig. 3.2 shows an example of SoC floor-planning based on 16 tiles organized in a 4×4 matrix.

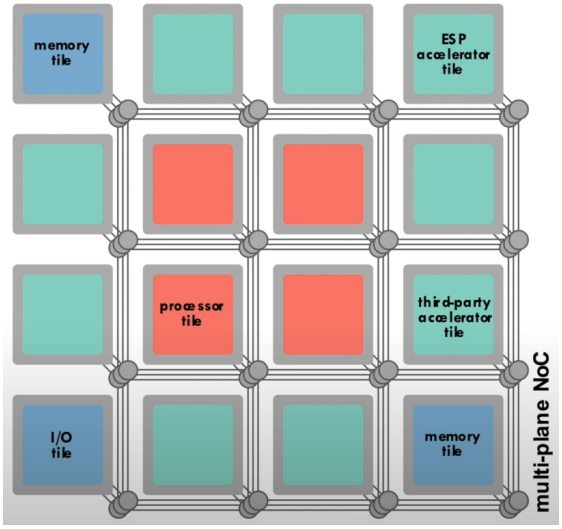

Figure 3.2: Tile-based ESP architecture. Source: [24].

While other open source RISC-V platforms for heterogeneous integration are modeled around the processor core, accelerators and processors have the same relevance in the system-centric ESP architecture.

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

- **Processor Tile**: three different cores are available in the ESP library: the 32-bit `SPARC V8 Leon3` core, the 64-bit `RISC-V CVA6-Ariane` core, and the 32-bit `RISC-V IBEX` core. These come with their private L1 cache and can boot `Linux`. A configurable private L2 cache can also be added to the cores. Moreover, ESP supports system-level coherency and dedicated planes in the NoC. Another 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 has one channel that connects the SoC to the external memory. Depending on the application, up to 4 tiles can be used. After their configuration, the hardware logic needed to handle the memory address space is automatically created. Also, the size of

the memories is configurable and data transfers from/to the memories is performed with coherent *Direct Memory Access* (*DMA*) units that are present inside accelerator tiles.

- **Accelerator Tile**: it allows to easily integrate loosely-coupled accelerators that perform specific tasks. Accelerators exchange large amounts of data with the memory hierarchy and can run independently from the processor cores. *Private Local Memories* (*PLMs*) are also present inside each accelerator tile to store local data and are generally much smaller than the external memory. Additionally, various cache coherency models can be selected at runtime and can be dynamically reconfigured.

- **Auxiliary Tile**: it is an optional tile that contains all the shared peripherals in the SoC, memories excluded. For instance, these peripherals can be: a digital video interface, a debug link or a monitor module. This particular tile is very flexible because it has to support many devices, but, for the same reason, it is also the most complex.

In addition, the ESP platform offers some services for the design of the accelerator tile, like dynamic voltage and frequency scaling, as well as performance counters and monitors. These services can be set at design time, while reconfigurability options are available at runtime.

However, the elements that simplify the most the SoC integration are the *hardware sockets*.

The accelerator socket allows different accelerators to be designed independently from the rest of the SoC. Newly designed accelerators, as well as third-party IP blocks like NVDLA, can be easily inserted into an accelerator tile without specifying any configuration for: virtual memory and DMA, data transfers from/to the PLMs, memory mapped registers, and interrupt requests. Indeed, the multi-plane NoC only interacts with the socket. Moreover, if a third-party accelerator has only a subset of these functionalities, the socket can be simplified to accommodate them. NVDLA, for instance, is a configurable accelerator and communicates in a way that is similar to the ESP-socket model.
A comparison between newly-designed and third party accelerator sockets is reported in Fig. 3.3.

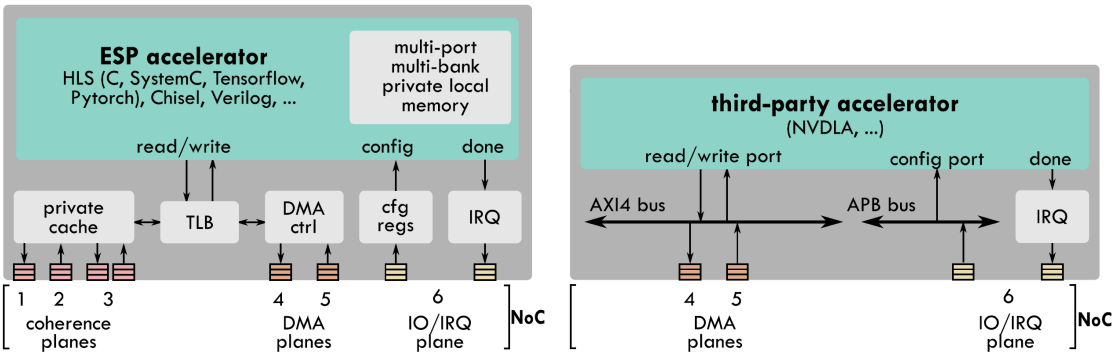

Figure 3.3: Comparison between newly-designed and third-party accelerator sockets. Source: [26].

The invocation of accelerators from a user application is made through the ESP accelerator's *Application Programming Interface* (*API*). This means that a computationally intensive software execution can be quickly replaced with the corresponding hardware accelerated version exploiting a single function code that starts the accelerator execution. In addition, the accelerator's PLMs are allocated in such a way that the performance is not affected. This is due to the fact that the ESP library preserves the concept of virtual address space.

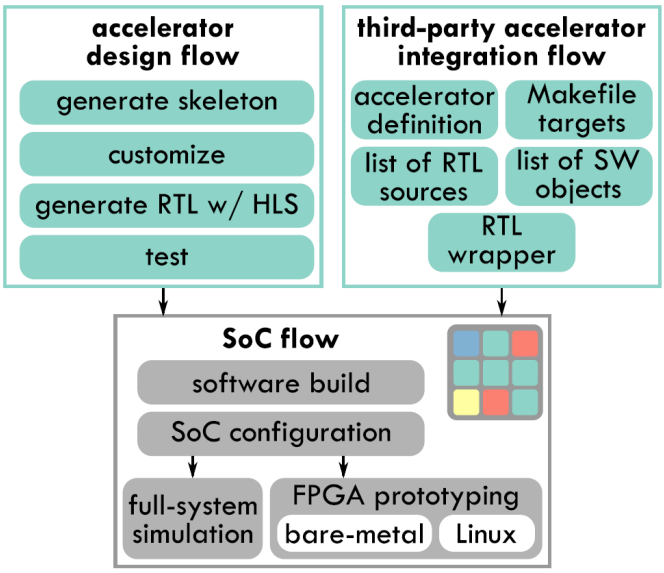

Figure 3.4: ESP accelerator's design and integration flow. Source: [26].

The design and integration of a new accelerator is facilitated by ESP because, as previously stated, it accommodates different design flows and it supports many CAD tools. These flows are interactive or automated for the most part and consist of the steps reported in Fig. 3.4 [26]. The figure shows how the newly-designed accelerators' design flow differs from the one of third-party accelerators. Since we are more interested in the former, we shall focus on the steps needed to design an accelerator from scratch, but it should be noted that a different, yet equivalent, design flow exists for the integration of already existing IP blocks like the NVDLA.

Depending on the chosen design flow, the user is invited to start from an accelerator' skeleton. It can be based either on a very simple accelerator available in the ESP documentation, or an automatically generated template produced by an interactive *template generator*. This generator also produces the device drivers and the *unit testbench* which models the behavior of the accelerator socket. *Bare-metal test applications* are also generated. These software applications can be run directly on the processor tile of the SoC without needing an OS. In this way, out-of-the-box simulation and full-system prototyping are enabled.

It should also be noted that ESP facilitates the invocation of accelerators from software applications running on top of `Linux`. However, this possibility has not been explored in this thesis and only bare-metal applications were used for testing RTL implementations.

While many phases of the accelerator design flow are either interactive or automated, designers are responsible for manually writing the algorithm needed for the computational part of the accelerator. Functions required to generate the inputs and to validate the outputs must also be customized for both the unit testbench and the bare-metal test applications.

Commercial HLS tools can be leveraged to produce the RTL implementation of the accelerator which is added to the ESP library of IP components and can later be integrated into the SoC.

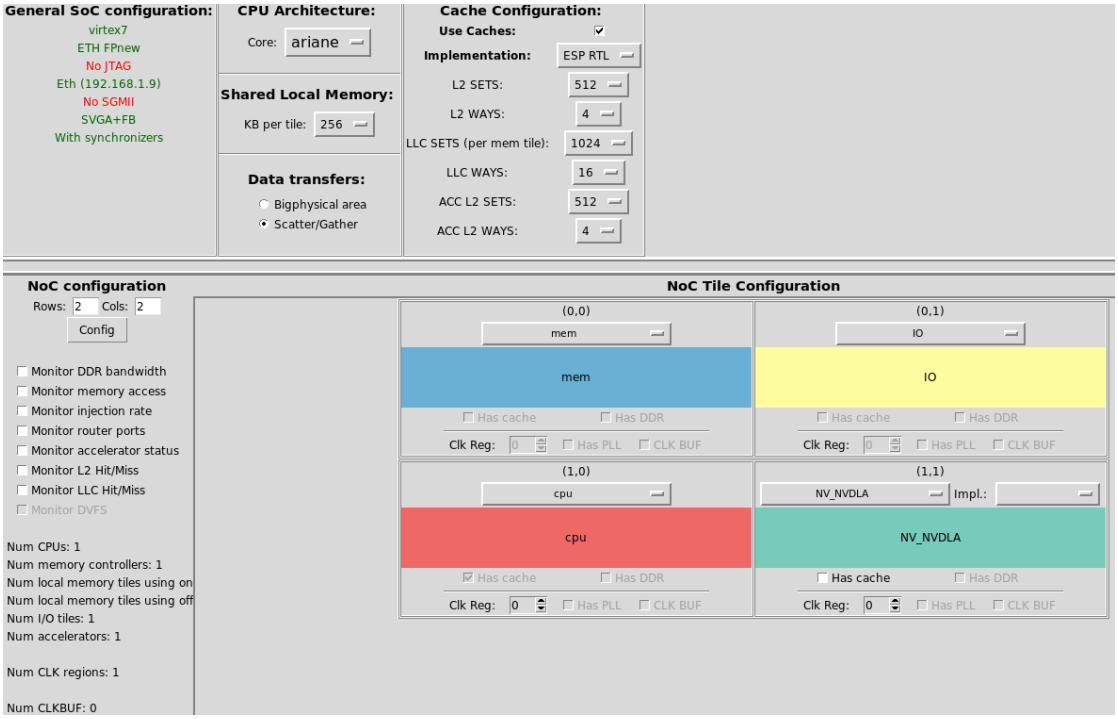

Figure 3.5: ESP Graphical User Interface for designing an SoC.

Fig. 3.5 shows the ESP GUI for designing an SoC. The GUI automatically discovers all the RTL codes of the already developed accelerators and it allows to insert them in the corresponding accelerator tiles. Additionally, bare-metal test programs can be compiled thanks to a set of *makefiles*. Finally, ESP produces the full RTL implementation of the SoC which includes also processor, memory and I/O tiles: now it is ready to be prototyped on top of an FPGA board.

Recently, the agile design methodology of ESP was employed to develop an SoC with a tile-based architecture for the application domain of swarm-based perception [27]. Third-party accelerators, as well as newly designed ones, were integrated on a SoC together with multiple processor cores and memory channels, as shown in Fig. 3.6. A team of ten full-time designers was able to complete this project in 4 months, 3 less than the 7 which would be needed to design this kind of chip without leveraging ESP. This achievement further shows the remarkable capabilities of the ESP platform.

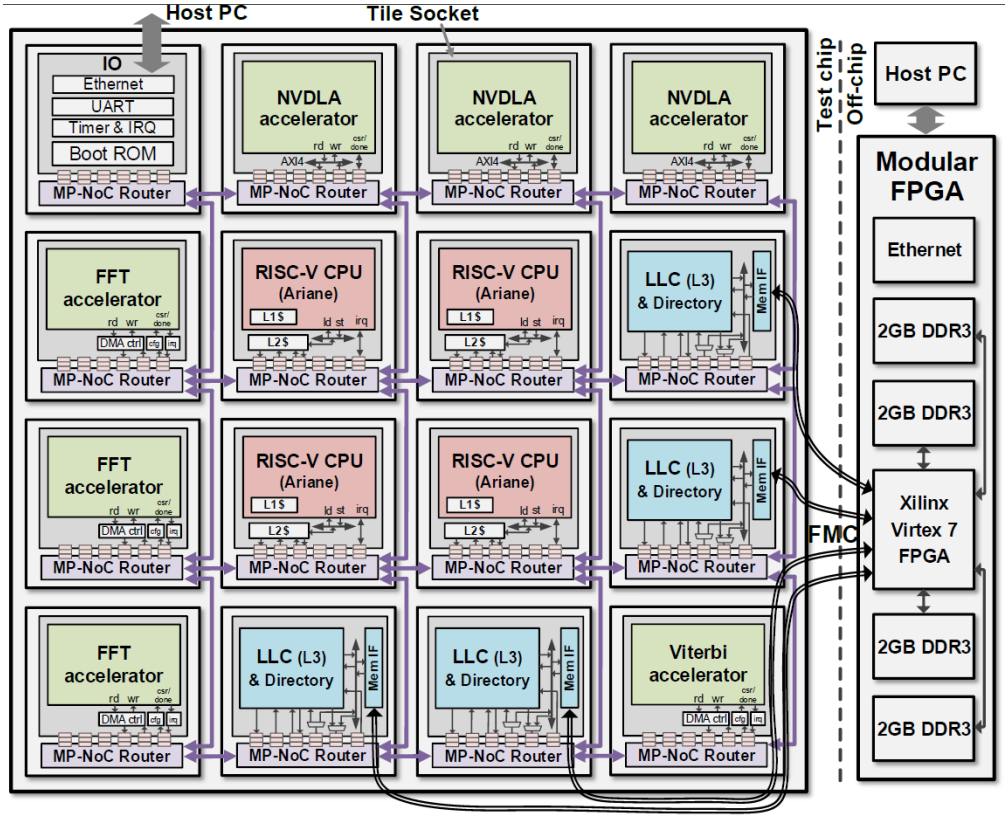

Figure 3.6: Architecture of the first chip based on the ESP platform. Source: [27].

Chapter 4

2D-Convolution Accelerator

So far, we have taken a look at various modern ConvNets and pinpointed the convolution as the most common operation in these types of networks, as well as the most computationally demanding. We saw how hardware accelerators can be used to perform specific functions in parallel with other operations performed by the processor core. Then, we discussed how these components can be easily designed and synthesized leveraging high-level synthesis tools. Finally, we illustrated the open source ESP platform which facilitates the integration of different hardware blocks in an SoC.
With this knowledge, the task of designing an accelerator performing a *2D convolution* between input and weight tensors can be addressed in the following paragraph.

4.1 2D Convolution

A tensor is a 3-dimensional data structure composed by 2-dimensional matrices stacked along the third spatial dimension. A 2-dimensional matrix is called "channel" or "kernel" whether it belongs to the input/feature map tensor or to the filter/weight tensor, respectively.
Usually, 2D convolution convolves more than one filter with the same input tensor. For each filter, each input channel is convolved with the corresponding kernel and then all these partial results are added together to generate one output channel, as shown in Fig. 4.1. For this reason, the number

of kernels is equal to the number of input channels. This results in the filter only moving in two directions: height and width, hence the name "2D convolution".

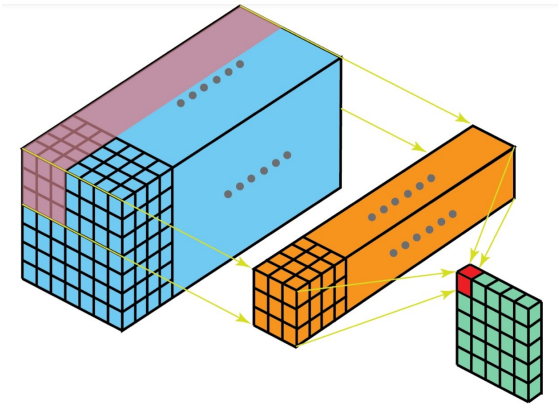

Figure 4.1: 2D Convolution with a Single Filter. Source: [28].

Repeating the same process for every filter gives as a result an output tensor with a number of channels determined by the total amount of filters, as reported in Fig. 4.2.

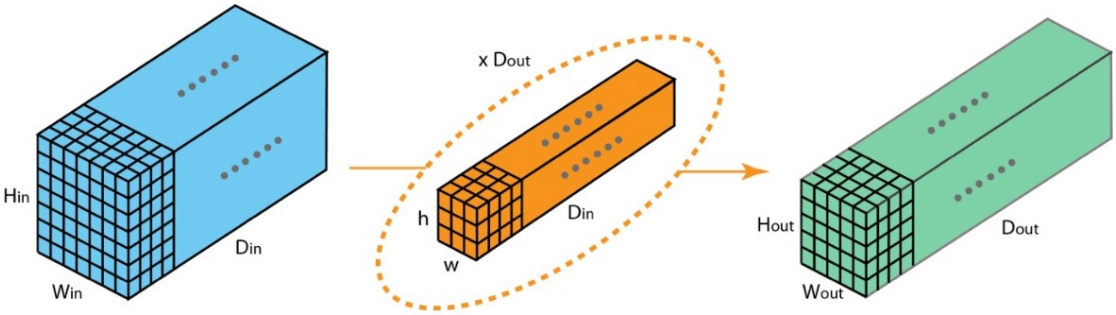

Figure 4.2: 2D Convolution with Multiple Filters. Source: [28].

4.2 ESP Accelerator Design

When designing hardware accelerators it can be quite challenging to devise an interface that can easily communicate with other hardware components. However, this is made easy by ESP, which grants support for various HLS tools. Since C++ is the language of choice for our design:

- Accelerators can be designed leveraging `Xilinx Vivado HLS`. For this particular tool, ESP provides an interactive template generator script which allows to easily produce the skeleton of a generic accelerator from some user-defined parameters, as reported in Listing 4.1;

```
========== Initializing ESP accelerator template ==========

* Enter accelerator name [dummy]: mac
* Select design flow (Stratus HLS, Vivado HLS) [S]: V
* Enter ESP path [/space/esp-master]:
* Enter unique accelerator id as three hex digits [04A]: 056
* Enter accelerator registers
  - register 0 name [size]: mac_len
  - register 0 default value [1]: 64
  - register 0 max value [64]:
  - register 1 name []: mac_vec
  - register 1 default value [1]: 100
  - register 1 max value [100]:
  - register 2 name []: mac_n
  - register 2 default value [1]:
  - register 2 max value [1]: 16
  - register 3 name []:
* Configure PLM size and create skeleton for load and store:
  - Enter data bit-width (8, 16, 32, 64) [32]:
  - Enter input data size in terms of configuration registers
    (e.g. 2 * mac_len}) [mac_len]: mac_len * mac_vec
    data_in_size_max = 6400
  - Enter output data size in terms of configuration registers
    (e.g. 2 * mac_len) [mac_len]: mac_vec
    data_out_size_max = 100
  - Enter an integer chunking factor (use 1 if you want PLM size equal
    to data size) [1]:
    Input PLM has 6400 32-bits words
    Output PLM has 100 32-bits words
  - Enter number of input data to be processed in batch (can
    be function of configuration registers) [1]: mac_n
    batching_factor_max = 16

========== Generated accelerator skeleton for mac ==========
```

Listing 4.1: Example of the interactive template generator script for an accelerator performing the multiply-and-accumulate operation. Source: [29].

- However, the accelerator generated by the aforementioned script does not support the co-simulation of C++ and RTL, because the C++ testbench is not able to model the DMA controller and to respond to the blocking requests of the accelerator [29]. Since we believe that the co-simulation of our accelerator is extremely useful at design time, we must rely on a different tool, `Catapult HLS`, that can perform the simulation of the high-level design and the co-simulation of its RTL

implementation. Although ESP does not provide a script to generate the skeleton of the accelerator for this particular tool, an example of a softmax accelerator is available in the ESP documentation [30].

4.2.1 Accelerator Files Organization

The provided softmax accelerator, which performs a simple computation between elements of the same input vector, does not simply consist of a single file with the high-level description of the accelerator, but it is divided in multiple directories. They include various C++ files as well as scripts that automate both the synthesis and the validation of the design. Therefore we decided to start from this set of files and directories to develop our `conv2d` accelerator.Our plan is to integrate it in an SoC with a 64-bit Ariane RISC-V processor. The file organization for this accelerator is reported in Lst. 4.2:

```
<esp>/accelerators/catapult_hls/conv2d_cxx_catapult
| - hw
|   | - conv2d_cxx.xml
|   | - hls
|   |   | - Makefile
|   |   | - build_prj.tcl
|   |   | - build_prj_top.tcl
|   | - inc
|   |   | - conf_info.hpp
|   |   | - fpdata.hpp
|   |   | - conv2d.hpp
|   | - src
|   |   | - basic
|   |   | - hier
|   | - tb
|       | - main.cc
| - sw
    | - baremetal
    |   | - conv2d_cxx.c
    |   | - Makefile
    | - linux
```

Listing 4.2: 2D-convolution accelerator - File Organization.

The main division is between the files needed to design the hardware (***hw***) and those related to the software (***sw***). The former comprises the source code, the unit testbench and the Catapult scripts for the accelerator, while the latter includes the test applications and the device drivers needed for the bare-metal and Linux execution on FPGA [31].

The `conv2d_cxx.xml` file contains some information about the accelerator, including the device ID and the name of the HLS tool that is being employed, in addition to the list of registers that can be overwritten by the user, as shown in Lst. 4.3.

```
<?xml version="1.0" encoding="UTF-8"?>
<sld>
  <accelerator name="conv2d_cxx" desc="Accelerator CONV2D [C++]" data_size="
    4" device_id="145" hls_tool="catapult_hls_cxx">
    <param name="batch" desc="batch" />
    <param name="n_w" desc="n_w" />
    <param name="n_h" desc="n_h" />
    <param name="n_c" desc="n_c" />
    <param name="kern" desc="kern" />
    <param name="filt" desc="filt" />
    <param name="same" desc="same" />
    <param name="stride" desc="stride" />
  </accelerator>
</sld>
```

Listing 4.3: `conv2d_cxx.xml` - General information about the 2D-convolution accelerator.

The ***hls*** folder contains the TCL scripts needed by Catapult to perform the synthesis and the co-simulation. In particular, the `build_prj_top.tcl` script allows to set the flags which indicate the operations to be performed by Catapult: simulation of the C++ code, high-level synthesis and RTL simulation. These flags may be enabled only one at a time or all together, as shown in the following Listing.

```
array set opt {
    # The 'csim' flag enables C simulation.
    # The 'hsynth' flag enables HLS.
    # The 'rtlsim' flag enables RTL simulation.
    # The 'debug' flag stops Catapult HLS before the architect step.
    # The 'hier' flag enables an implementation with hierarchical blocks.
    csim        1
    hsynth      1
    rtlsim      1
    debug       0
    hier        0
}
source ../../../common/hls/common.tcl
source ./build_prj.tcl
```

Listing 4.4: `build_prj_top.tcl` - Top Level script needed for building project.

The `build_prj.tcl` script, automatically called by the previous TCL script, contains all the Catapult commands required to carry out the HLS flow.

Many important directives are set such as clock period, pipeline constraints and design goal. For the synthesis of our first `conv2d` accelerator we initially set: clock frequency to 50 MHz, initiation interval to 1, and design goal to latency.

```
directive set -CLOCKS { \
    clk { \
        -CLOCK_PERIOD 20 \
        -CLOCK_EDGE rising \
        -CLOCK_HIGH_TIME 10 \
        -CLOCK_OFFSET 0.000000 \
        -CLOCK_UNCERTAINTY 0.0 \
        -RESET_KIND sync \
        -RESET_SYNC_NAME rst \
        -RESET_SYNC_ACTIVE low \
        -RESET_ASYNC_NAME arst_n \
        -RESET_ASYNC_ACTIVE low \
        -ENABLE_NAME {} \
        -ENABLE_ACTIVE high \
    } \
}
```

Listing 4.5: `build_prj.tcl` - Example of a Catapult directive. This one sets the clock constraints.

The ***inc*** directory contains the header files which are directly needed by the source files to design the accelerator.
The `conf_info.hpp` file reported in Lst. 4.6 hosts the structure needed to handle the user-defined accelerator parameters that will be stored in memory-mapped registers.

```
struct conf_info_t {
    uint32_t stride;
    uint32_t same;
    uint32_t filt;
    uint32_t kern;
    uint32_t n_c;
    uint32_t n_h;
    uint32_t n_w;
    uint32_t batch;
};
```

Listing 4.6: `conf_info.hpp` - Struct for configuration parameters.

The meaning of these configuration variables is the following:

- `n_w`: width of the input tensor;
- `n_h`: height of the input tensor;
- `n_c`: number of channels in the input tensor;

- `kern`: number of kernels in a filter;
- `filt`: number of distinct filters, hence output channels;
- `same`: 0 for no padding, or 1 for an output the same width and height of the input one;
- `stride`: number of input values that each kernel jumps when moving over the input channel;
- `batch`: number of input batches the accelerator has to compute.

The `fpdata.hpp` header file defines *fixed-point* datatypes using the format provided by Catapult (`ac_fixed`). In a fixed-point representation there is a decimal point that separates the total number of bits (W) into two sets: the most significant bits correspond to the integer part (I), while the remaining least significant bits are related to the fractional part (W - I).

```
ac_fixed<int W, int I, bool S, ac_q_mode Q, ac_o_mode O>
```

Listing 4.7: ac_fixed format provided by Catapult HLS [32].

The rest of its arguments are: S, which allows to select whether the value is either *signed* (true) or *unsigned* (false); Q, used to select the *quantization* policy; O, used to select the *overflow* policy.

For our design we chose to set the word length of our input and output data to 32 bits. Additionally, we set the integer and the fractional parts to have the same amount of bits, 16 each, and the signed flag to *true*, so that our accelerator could handle both positive and negative numbers.
The quantization policy decides what to do with the extra fractional bits. The most notable options are: `AC_TRN` which simply truncates/deletes them, and `AC_RND_CONV` which rounds them to the nearest even. For the input it is preferable to truncate the excess bits, while for the output they get rounded to achieve higher precision.
On the other hand, the overflow policy determines what to do when the integer part exceeds the predefined amount of bits. Most notably, there are the `AC_WRAP` option, which simply drops the bits to the left of the most significant bit (*MSB*), and the `AC_SAT` option, which saturates the values either to the minimum or to the maximum representable number, depending on which one is closer. So, the input and output data have the format reported in Lst. 4.8.

```
ac_fixed<32, 16, true, AC_TRN, AC_WRAP> FPDATA_IN;
ac_fixed<32, 16, true, AC_RND_CONV, AC_SAT> FPDATA_OUT;
```

Listing 4.8: `fpdata.hpp` - Datatypes used for input and output data.

However, most of the information regarding the design is in the main header file, `conv2d.hpp`, which contains the maximum values that the accelerator parameters can assume. In particular, we decided that our accelerator can handle at most:

- An $18 \times 18 \times 32$ input tensor;
- A $7 \times 7 \times 32 \times 32$ weight tensor;
- A stride of 2;
- 16 different batches.

Global `#define` directives can be used to set these constant parameters, as shown in Lst. 4.9.

```
// PLM and data dimensions
#define DATA_WIDTH 32

#define BATCH_MAX 16
#define N_W_IN_MAX 18
#define N_H_IN_MAX 18
#define N_C_MAX 32
#define KERN_MAX 7
#define FILT_MAX 32
#define STRIDE_MAX 2
#define N_W_OUT_MAX 18
#define N_H_OUT_MAX 18

#define FILTERS_SIZE_MAX KERN_MAX * KERN_MAX * N_C_MAX * FILT_MAX //50176
#define INPUTS_SIZE_MAX N_W_IN_MAX * N_H_IN_MAX * N_C_MAX //10368
#define OUTPUTS_SIZE_MAX N_W_OUT_MAX * N_H_OUT_IN_MAX * FILT_MAX //10368
```

Listing 4.9: `conv2d.hpp` - Define directives.

Based on these values, one additional datatype used for the partial sums of the MAC operations can be set in the `fpdata.hpp` file: `FPDATA_ACC`.
When multiplying and accumulating `FPDATA_IN` values whose integer part is on 16 bits, overflows may occur, meaning that 16 bits may not be enough to represent the resulting integer part. Therefore, we must consider the worst case scenario in which 32 different input channels are convolved with a $7 \times 7 \times 32$ filter. In this case, 1568 distinct products need to be added

together. The total number of bits (I_{acc}) needed to represent the integer part of this value is given by the following formula:

$$I_{acc} = \lceil 2 \cdot 16\ bits - 1 + \log_2(1568) \rceil = 42\ bits \tag{4.1}$$

Hence, the `FPDATA_ACC` datatype has the format shown in Lst. 4.10:

```
ac_fixed<58, 42, true, AC_RND_CONV, AC_SAT> FPDATA_ACC;
```

Listing 4.10: `fpdata.hpp` - Datatype used for partial sums.

`Conv2d.hpp` also contains some templated structures which facilitate the declaration of arrays and matrices. The structure called `plm_t` comprises an array that has S elements of a given T datatype. This structure is especially useful to describe the PLMs where features, weights and outputs can be stored, as shown in Lst. 4.11.

```
// The PLM array is encapsulated in a templated struct
template <class T, unsigned S>
struct plm_t {
public: T data[S];
};

// PLM typedefs
typedef plm_t<FPDATA_IN, INPUTS_SIZE_MAX> plm_inputs_t;
typedef plm_t<FPDATA_IN, FILTERS_SIZE_MAX> plm_filters_t;
typedef plm_t<FPDATA_OUT, OUTPUTS_SIZE_MAX> plm_outputs_t;
```

Listing 4.11: `conv2d.hpp` - Templated structure for PLMs.

We also defined a similar `buf_t` templated structure which comes in handy for defining the accumulation buffer used to store the partial sums of the multiply and accumulate operations.

```
// The Buffer matrix is encapsulated in a templated struct
template <class T, unsigned R, unsigned C>
struct buf_t{
public: T data[R][C];
};

// Buffer typedefs
typedef buf_t<FPDATA_ACC, N_W_OUT_MAX, N_H_OUT_MAX> buf_acc_t;
```

Listing 4.12: `conv2d.hpp` - Templated structure for the accumulation buffer.

Finally, the interface of the accelerator is reported in the header file, as shown in Lst. 4.13.

```
void 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);
```

Listing 4.13: `conv2d.hpp` - 2D-convolution accelerator interface.

The ESP accelerator interface is common to all the supported HLS flows and it allows to:

- interact with the CPU via memory-mapped registers through the `conf_info` port;
- set the DMA controller leveraging the `dma_read_ctrl` and `dma_write_ctrl` ports;
- exchange data with the DMA via `dma_read_chnl` and `dma_write_chnl`;
- signal the end of the task to the processor with `acc_done`.

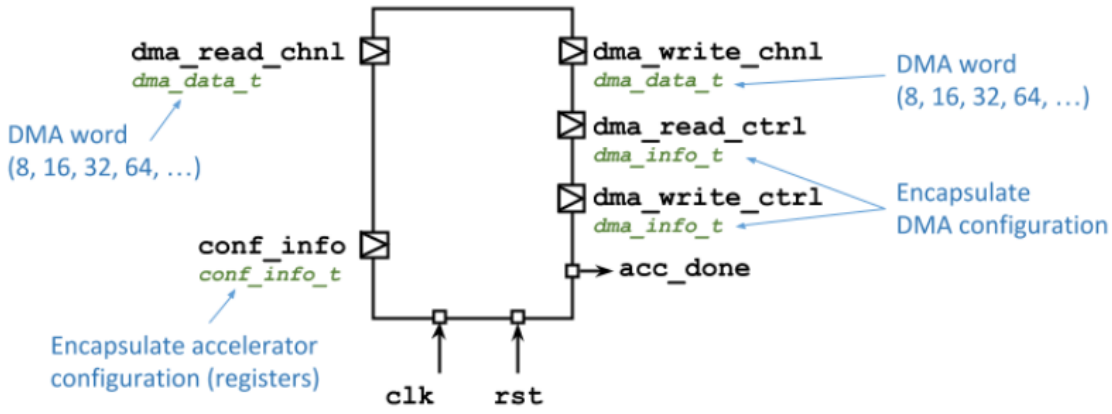

Figure 4.3: ESP accelerator interface.

This interface is able to properly communicate with external components if the following directives are set in the `build_prj.tcl` script.

```
# Top-Module I/O
directive set
    /$ACCELERATOR/conf_info:rsc -MAP_TO_MODULE ccs_ioport.ccs_in_wait
```

```
directive set
    /$ACCELERATOR/dma_read_ctrl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
directive set
    /$ACCELERATOR/dma_write_ctrl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
directive set
    /$ACCELERATOR/dma_read_chnl:rsc -MAP_TO_MODULE ccs_ioport.ccs_in_wait
directive set
    /$ACCELERATOR/dma_write_chnl:rsc -MAP_TO_MODULE ccs_ioport.ccs_out_wait
directive set
    /$ACCELERATOR/acc_done:rsc -MAP_TO_MODULE ccs_ioport.ccs_sync_out_vld
```

Listing 4.14: `build_prj.tcl` - Directives for the synthesis of the accelerator interface.

To guarantee that data are properly synchronized at the interface via a latency-insensitive protocol, certain specific datatypes for the accelerator ports must be used along with the aforementioned directories [33]. Indeed, the ports are based on either the `ac_channel` or the `ac_sync` models provided by Catapult. The former enables point-to-point connection between two instances through FIFOs and Catapult adds the necessary handshaking signals to the RTL implementation during synthesis. The latter is a synchronization channel that serves a similar purpose and is synthesized with the proper handshaking signals too.

4.2.2 The Sequential Architecture

The actual high-level description of the accelerator is stored in the ***src*** directive. There are two available C++ implementations: the first one describes a sequential architecture (*basic*), while the second one implements the accelerator based on a hierarchical architecture (*hier*).

In fact, the accelerator execution comprises four distinct phases: *configuration*, *load*, *compute* and *store*.
In the sequential implementation all these four phases are executed sequentially, while in the hierarchical design they can be pipelined.

Configuration Phase

In the configuration phase the parameters stored in the memory-mapped registers are read via the `conf_info` port. Data coming from this `ac_channel` are stored in the proper `conf_info_t` structure. In this way, each parameter can be assigned to a local variable and used for the subsequent phases. Some of these variables are: the amount of padding to add to the input tensor,

the resulting width and height of the padded input tensor, as well as the width and height of the output tensor. It should also be noted that the stride parameter is used to determine the dimensions of the output tensors. However, since there is no technology library, imported by Catapult HLS, that allows to set a non-constant variable as the divisor in a division, a multiplexed implementation is employed instead, as shown in Lst. 4.15.

```
// Parameters
struct conf_info_t params;
uint8_t batch = 0;
uint8_t n_w = 0;
uint8_t n_h = 0;
uint8_t n_c = 0;
uint8_t kern = 0;
uint8_t filt = 0;
uint8_t same = 0;
uint8_t stride = 0;

// Read accelerator configuration
params = conf_info.read();

batch = params.batch;
n_w  = params.n_w;
n_h = params.n_h;
n_c = params.n_c;
kern = params.kern;
filt = params.filt;
same = params.same;
stride = params.stride;

// Padding
const uint8_t pad = ((stride * (n_w - 1) - n_w + kern) / 2) * same;

// Width and Height of Padded Input Tensor
const uint8_t n_w_in = n_w + 2 * pad;
const uint8_t n_h_in = n_h + 2 * pad;

// Width and Height of Output Tensor - Multiplexed implementation
uint8_t n_w_out;
uint8_t n_h_out;
if (stride == 1){
    n_w_out = (n_w + 2 * pad - kern) + 1;
    n_h_out = (n_h + 2 * pad - kern) + 1;
}
else{
    n_w_out = (n_w + 2 * pad - kern)/2 + 1;
    n_h_out = (n_h + 2 * pad - kern)/2 + 1;
}
```

Listing 4.15: basic → `conv2d.cpp` - Configuration phase.

Whether they are weights or features, input values are stored in the external memory. They can be divided into different independent batches and we

decided to organize them as shown in Fig. 4.4.

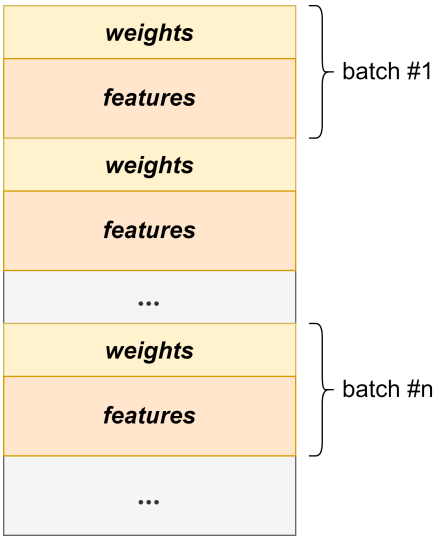

Figure 4.4: Input and weight data in the external memory.

Batching allows a single accelerator to perform multiple independent convolutions as long as the inputs are properly located. At each batch iteration the load, compute and store phases are all executed sequentially, as reported in the following pseudo-code in Lst. 4.16.

```
for (uint8_t b = 0; b < batch; b++) {

    LOAD_PHASE(...)

    COMPUTE_PHASE(...)

    STORE PHASE(...)
}
```

Listing 4.16: Batch iterations pseudo-code.

Load Phase

During the `load` phase, weights and features are read from the external memory and stored into smaller PLMs. The `dma_read_ctrl` and `dma_read_chnl` ports enable this data transfer. In particular, the former contains the following information:

- The *index* specifying the memory offset of the first input;

- The *length* of the DMA transaction;
- An encoding of the DMA *width*, which in our case is set to 64 bits so that it is compliant with the Ariane RISC-V core standards.

```
// DMA configuration
dma_info_t dma_read_info = {0, 0, 0};

// DMA variables
uint16_t dma_read_data_index = 0;
uint16_t dma_read_data_length = n_w * n_h * n_c + kern * kern * n_c * filt;

// Configure DMA read channel (CTRL)
dma_read_data_index = dma_read_data_length * b;
dma_read_info = {dma_read_data_index, dma_read_data_length, DMA_SIZE};
bool dma_read_ctrl_done = false;

LOAD_CTRL_LOOP:
do {dma_read_ctrl_done = dma_read_ctrl.nb_write(dma_read_info);}
while (!dma_read_ctrl_done);
```

Listing 4.17: basic → `conv2d.cpp` - Load phase pt.1: DMA configuration.

Once the DMA controller has been set as shown in Lst. 4.17, two distinct *for-loops* store the weights and the features in two different PLMs.
Storing the elements of the filters is quite straightforward: at every cycle of the `LOAD_WEIGHTS_LOOP` a new 64-bit data is read from the DMA channel and only the 32 least-significant bits are kept, as shown in Lst. 4.18.

```
// Private Local Memory
plm_filters_t plm_f;

LOAD_WEIGHTS_LOOP:
for (uint16_t i = 0; i < FILTERS_SIZE_MAX; i++) {
    // DMA_WIDTH = 64
    // DATA_WIDTH = 32
    // discard bits in the DMA range(63,32)
    // keep bits in the DMA range(31,0)
    ac_int<DATA_WIDTH, false> data_ac = dma_read_chnl.read().template slc<
    DATA_WIDTH>(0);

    FPDATA_IN data;
    data.set_slc(0, data_ac);

    plm_f.data[i] = data;

    if (i == kern * kern * n_c * filt - 1) break;
}
```

Listing 4.18: basic → `conv2d.cpp` - Load phase pt.2: loading weights.

Then, the features are loaded into another PLM in the `LOAD_FEATURES_LOOP`. However, this PLM is not simply initialized with the values coming from the external memory, but, depending on whether the padding parameter has been set or not, it may be necessary to load certain cells with zeroes. A specific algorithm has been implemented to support this functionality, which allows to temporarily pause the read operation from the DMA and to fill the PLM with zeroes where needed, as reported in Lst. 4.19.

```
// Private Local Memory
plm_inputs_t plm_in;

LOAD_FEATURES_LOOP:
for (uint8_t chan = 0; chan < N_C_MAX; chan++) {
    for (uint8_t row = 0; row < N_W_IN_MAX; row++) {
        for (uint8_t col = 0; col < N_H_IN_MAX; col++) {
            FPDATA_IN data;
            // Non-zero values
            if ((row >= pad) && (col >= pad) && (col < n_h_in - pad)
                 && (row < n_w_in - pad)) {
            ac_int<DATA_WIDTH, false> data_ac = dma_read_chnl.read().
    template slc<DATA_WIDTH>(0);
                data.set_slc(0, data_ac);
            }
            else // Zero values
                data = 0;

            uint16_t index_in = chan * n_w_in * n_h_in + row * n_h_in + col;
            plm_in.data[index_in] = data;

            if (col == n_h_in - 1) break;
        }
        if (row == n_w_in - 1) break;
    }
    if (chan == n_c - 1) break;
}
```

Listing 4.19: basic → `conv2d.cpp` - Load phase pt.3: loading features.

This algorithm allows to have the padded input ready to be convolved without having to worry about padding anymore. Once both PLMs have been loaded, the accelerator is ready to perform the convolution operation between the features and the weights.

Compute Phase

A set of nested for-loops allows to iteratively compute the partial sums for each element of the output tensor resulting in an output stationary conv2d implementation.
The complete algorithm is reported in Lst. 4.20.

```
// Accumulation buffer
buf_acc_t buf_acc;
for (uint8_t fl = 0; fl < FILT_MAX; fl++) {
    for (uint8_t k = 0; k < N_C_MAX; k++) {
        for (uint8_t i = 0; i < N_W_OUT_MAX; i++) {
            for (uint8_t j = 0; j < N_H_OUT_MAX; j++) {
                // This holds a single MAC result
                ac_fixed<FPDATA_WL + FPDATA_OUT_IL - 1, FPDATA_OUT_IL * 2 -
    1, true, AC_RND_CONV, AC_SAT> acc = 0;

                // First element of input tensor
                uint8_t x = i * stride;
                uint8_t y = j * stride;

                // Kernel rows and columns are m and n respectively
                for (uint8_t m = 0; m < KERN_MAX; m++) {
                    for (uint8_t n = 0; n < KERN_MAX; n++) {
                        uint16_t index_f, index_in;
                        index_in = n_w_in * n_h_in * k + x * n_w_in + y;
                        index_f = fl * kern * kern * n_c + k * kern * kern +
                                  m * kern + n;
                        acc += plm_f.data[index_f] * plm_in.data[index_in];

                        y++; // Move right by one position
                        if (n == kern - 1) break;
                    }
                    x++; // Move down by one position
                    y = j * stride; // Restart column position
                    if (m == kern - 1) break;
                }
                uint16_t index_out = n_w_out* n_h_out* fl + i* n_w_out + j;
                if (k == 0)
                    buf_acc.data[i][j] = acc;
                else
                    buf_acc.data[i][j] += acc;
                if (k == n_c - 1)
                    plm_out.data[index_out] = buf_acc.data[i][j];
                if (j == n_h_out - 1) break;
            }
            if (i == n_w_out - 1) break;
        }
        if (k == n_c - 1) break;
    }
    if (fl == filt - 1) break;
}
```

Listing 4.20: basic → `conv2d.cpp` - Compute phase.

We have already seen in paragraph 4.1 how for the 2D-convolution algorithm, the number of kernels in a filter corresponds to the amount of input channels. For this reason, each kernel is convolved with the corresponding channel one at the time. Each partial sum is temporarily stored in a small register (`acc`) before being written into a large one (`buf_acc`) in order to reduce the number of accesses to the latter in the same clock cycle. The large

accumulation buffer is only updated before striding the kernel in the next position of the receptive field.
Once each kernel in a filter has been convolved with all the input channels, the output values are moved to the output PLM (`plm_out`). Then, the same convolution operations between the input tensor and the other filters are executed until all the elements of the output tensor have been computed.

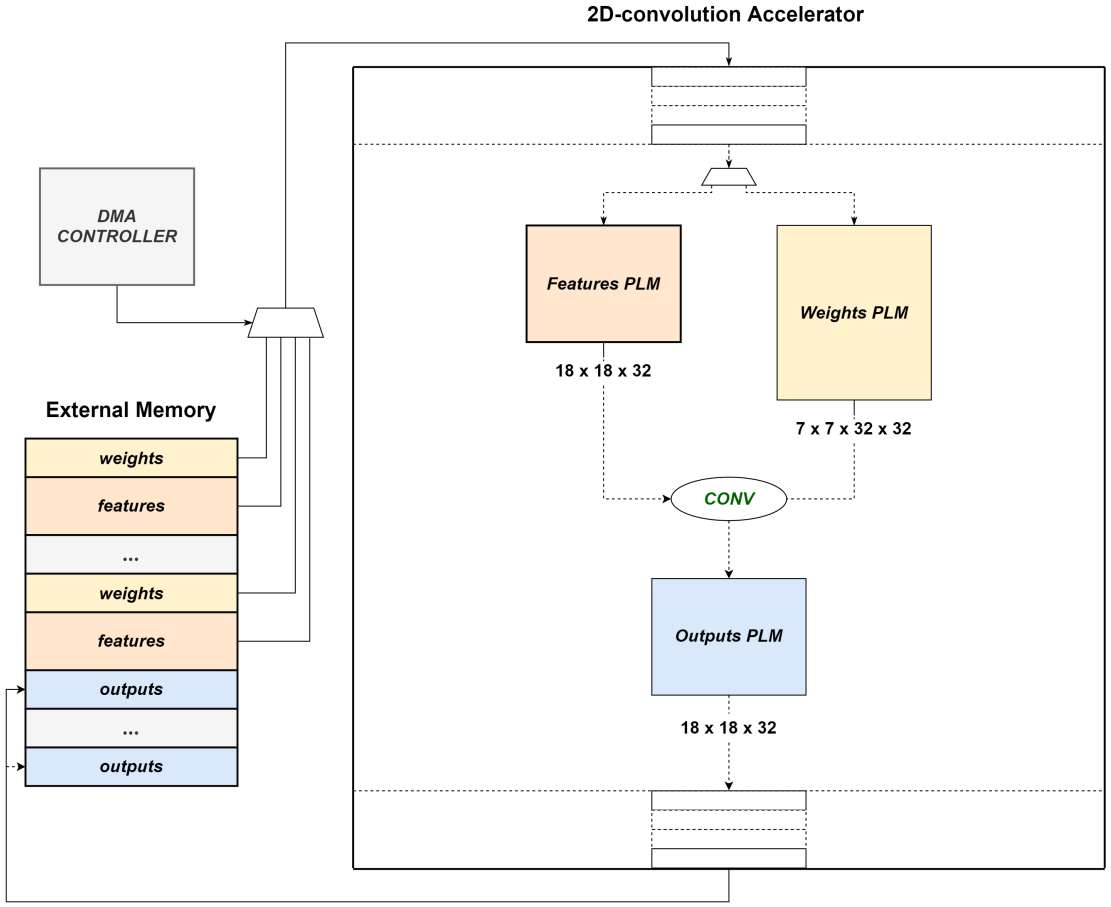

Figure 4.5: 2D-convolution Accelerator: sequential architecture.

Store Phase

Finally, once all output values have been written into the output PLM, they can be stored in the external memory by purposefully setting the DMA controller. Since in this phase we assume to have a single memory at our disposal, we want the accelerator to store the outputs in the external memory without overwriting the inputs. Thus, in the store phase the `dma_write_ctrl`

port must be set at each batch iteration in such a way that data allocation is compliant with the arrangement reported in Fig. 4.6.

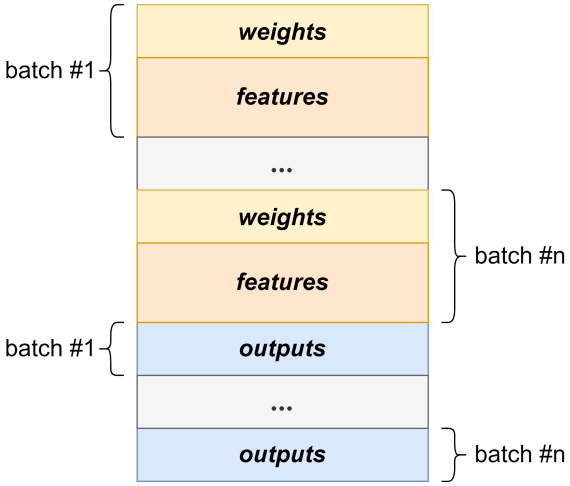

Figure 4.6: External memory with inputs and outputs.

After setting the DMA controller, data are written from the output PLM to the DMA write channel. Since the DMA channel is 64-bit wide and the output data are on 32 bits, the upper bits are set to a constant value (`DEADBEEF` in hexadecimal notation).

```
// Configure DMA write channle (CTRL)
dma_write_data_index = (dma_read_data_length * batch) +
    dma_write_data_length * b;
dma_write_info = {dma_write_data_index, 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);
// Force serialization between DMA control and DATA data transfer
if (dma_write_ctrl_done) {
STORE_LOOP:
    for (uint16_t i = 0; i < OUTPUTS_SIZE_MAX; i++) {
        FPDATA_OUT data = plm_out.data[i];
        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));
        dma_write_chnl.write(data_ac);

        if (i == dma_write_data_length - 1) break;
    }
}
```

Listing 4.21: basic → `conv2d.cpp` - Store phase.

After completing all the batch iterations, the accelerator execution is over. A pulse is sent via the `acc_done` signal to communicate the task completion to the processor.

The sequential architecture of our 2D-convolution accelerator is shown in Fig. 4.5.

4.2.3 The Hierarchical Architecture

In the hierarchical implementation the config, load, compute and store phases are described in distinct functions/blocks which are able to run concurrently. However, concurrency can only be ensured by applying specific HLS constraints and by adopting the proper coding style. If these conditions are fulfilled, the necessary synchronization signals are automatically added by the HLS toll, in this case Catapult HLS.

Each C++ function implementing a different accelerator phase is called by the top-level function. These blocks exchange data through globally defined PLMs, which are modelled as `ac_channels` and come with a set of handshaking signals that are crucial to ensure synchronization. Configuration parameters are also transferred through this type of channel. Moreover, synchronization signals (`ac_sync`) are used to determine the end of each phase.
Once all phases have been completed, the accelerator notifies the processor via the `acc_done` signal.

```
void 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)
{
    // Private Local Memories
    static ac_channel<plm_inputs_t> plm_inputs;
    static ac_channel<plm_filters_t> plm_filters;
    static ac_channel<plm_outputs_t> plm_outputs;

    // Configuration channels
    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;
```

```
    // Done signals
    static ac_sync config_done;
    static ac_sync load_done;
    static ac_sync compute_done;
    static ac_sync store_done;

    // Accelerator blocks
    config(conf_info, plm_conf_load, plm_conf_compute, plm_conf_store,
    config_done);
    load(plm_conf_load, plm_inputs, plm_filters, dma_read_ctrl,
    dma_read_chnl, load_done);
    compute(plm_conf_compute, plm_inputs, plm_filters, plm_outputs,
    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();
}
```

Listing 4.22: hier → `conv2d.cpp` - Top-level function.

In addition, each block must contain a local instance of the shared memory to perform memory operations. For example, the store block receives the PLM containing the outputs via an `ac_channel` port and then it copies its content into a local PLM. Finally, it reads the data from this local instance and it writes each output value in the external memory through the DMA channel, as shown in Lst. 4.23.

```
void store(
    // ...
    ac_channel<plm_outputs_t> &plm_outputs, // shared memory over ac_channel
    // ...
) {
    // read from the ac_channel into a local instance of the memory
    plm_outputs_t plm_tmp = plm_outputs.read();
    for (uint16_t i = 0; i < OUTPUTS_SIZE_MAX; i++) {
        FPDATA_OUT data = plm_tmp.data[i];
        dma_write_chnl.write(data);
    }
}
```

Listing 4.23: hier → `conv2d.cpp` - Store block.

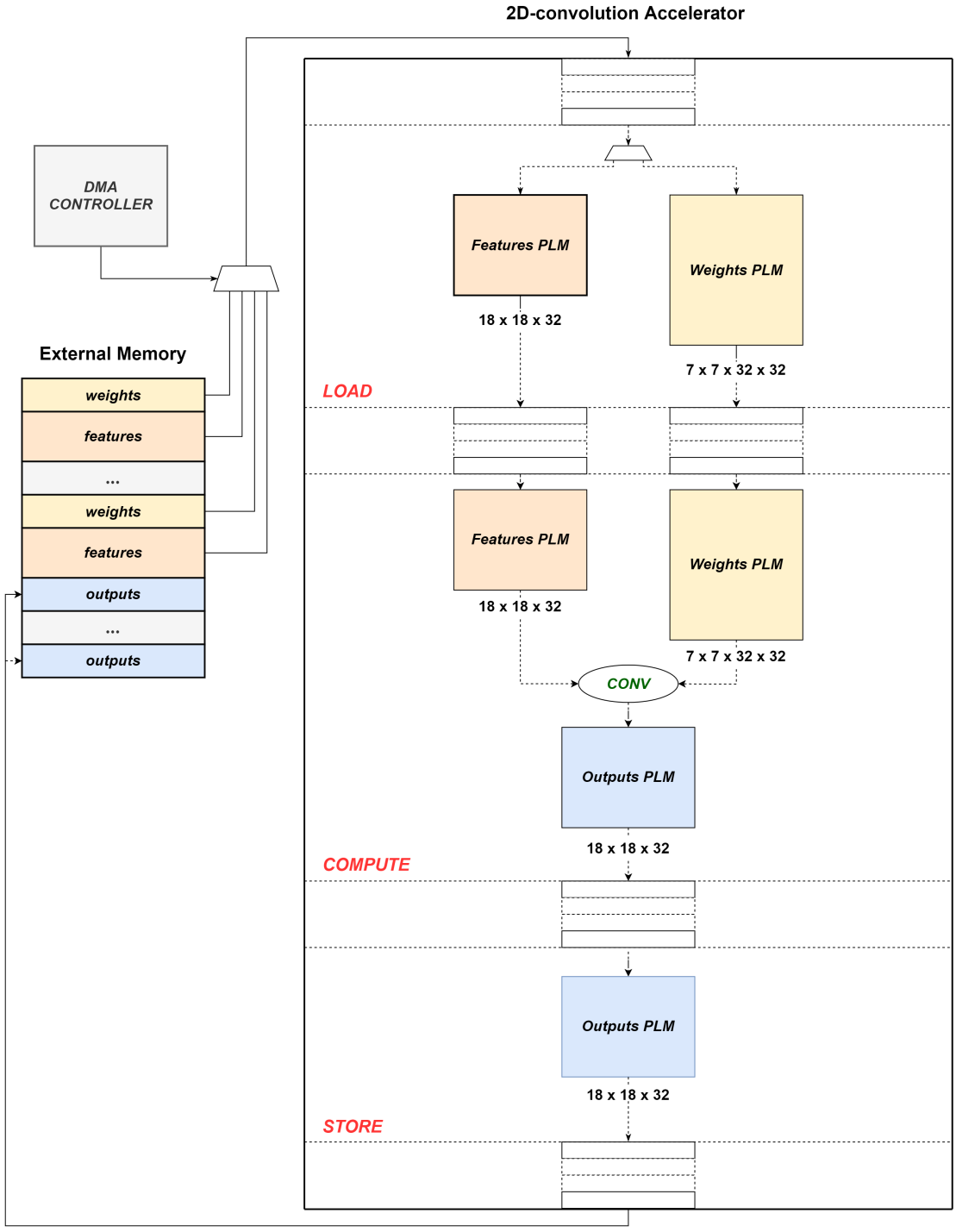

Figure 4.7: 2D-convolution Accelerator: hierarchical architecture.

The hierarchical architecture allows to execute the different phases concurrently, as long as data-dependencies are respected. However, this comes at the cost of a larger area compared to the sequential architecture, due to the fact that subsequent blocks require a copy of the same memory each and additional hardware to exchange data. The corresponding architecture is shown in Fig. 4.7.

4.3 Co-Simulation and Validation

The `main.cpp` file in the ***tb*** folder is the testbench used by Catapult and `QuestaSim` to carry out the simulation and co-simulation of the accelerator and so to validate the results.

In the C++ unit testbench the configuration parameters can be manually set and other parameters needed for validation can be derived from them.
For instance, we may perform the convolutions between 4 batches of $5 \times 5 \times 3$ input tensors and $3 \times 3 \times 3 \times 3$ filters with padding and a stride equal to 1 as shown in Lst. 4.24. This is useful especially during the debugging phase of the accelerator because the RTL simulation requires a huge amount of time when the accelerator has to process large input and weight tensors.

```
// Set config parameters
const uint8_t batch = 4;
const uint8_t n_w = 5;
const uint8_t n_h = 5;
const uint8_t n_c = 3;
const uint8_t kern = 3;
const uint8_t filt = 3;
const uint8_t same = 1;
const uint8_t stride = 1;

// Parameters for validation
const uint8_t pad = ((stride * (n_w - 1) - n_w + kern)/2) * same;
const uint8_t n_w_out = (n_w + 2 * pad - kern)/(stride) + 1;
const uint8_t n_h_out = (n_h + 2 * pad - kern)/(stride) + 1;

const unsigned input_size = n_w * n_h * n_c + kern * kern * n_c * filt;
const unsigned output_size = n_w_out * n_h_out * filt;

// Accelerator configuration
ac_channel<conf_info_t> conf_info;

conf_info_t conf_info_data;
```

```
conf_info_data.batch = batch;
conf_info_data.n_w = n_w;
conf_info_data.n_h = n_h;
conf_info_data.n_c = n_c;
conf_info_data.kern = kern;
conf_info_data.filt = filt;
conf_info_data.same = same;
conf_info_data.stride = stride;

// Pass configuration to the accelerator
conf_info.write(conf_info_data);
```

Listing 4.24: tb → `main.cpp` - Configuration parameters.

The testbench can also randomly initialize the input data of the accelerator, exploiting the `srand()` and `rand()` functions, as shown in Lst. 4.25. Weights and features are not only used to fill the `dma_read_channel`, so that the accelerator can fetch them, but they are also stored in an array, so that a specific *golden function* (`conv2d_tb`) can perform in software the same algorithm of the accelerator to create the *golden output*, used to compare the output of the accelerator.

```
// Communication channels
ac_channel<dma_info_t> dma_read_ctrl;
ac_channel<dma_data_t> dma_read_chnl;

// Testbench data
FPDATA_IN inputs[input_size * BATCH_MAX];

// initialize random seed:
srand (time(NULL));
for (unsigned i = 0; i < conf_info_data.batch * input_size; i++) {
    FPDATA_IN data_fp = (rand() % 200 - 100) * 0.25;
    inputs[i] = data_fp;

    ac_int<DMA_WIDTH, true> data_ac;
    ac_int<DMA_WIDTH/2, true> DEADBEEF = 0xdeadbeef;
    data_ac.set_slc(DMA_WIDTH/2, DEADBEEF.template slc<DMA_WIDTH/2>(0));
    data_ac.set_slc(0, inputs[i].template slc<DATA_WIDTH>(0));

    dma_read_chnl.write(data_ac);
}
```

Listing 4.25: tb → `main.cpp` - Input generation.

Once the configuration parameters and the inputs have been set, the accelerator can be run. Then, the output data are fetched from the `dma_write_chnl` port, as shown in Lst. 4.26.

```
// Testbench data
FPDATA_OUT outputs[output_size * BATCH_MAX];
```

```
// Run the accelerator
conv2d_cxx_catapult(conf_info, dma_read_ctrl, dma_write_ctrl, dma_read_chnl,
     dma_write_chnl, acc_done);

// Testbench stalls until data ready
while (!dma_write_chnl.available(conf_info_data.batch * output_size)) {}

// Fetch outputs from the accelerator
for (unsigned i = 0; i < conf_info_data.batch * output_size; i++) {
    ac_int<DATA_WIDTH, true> data = dma_write_chnl.read().template slc<
    DATA_WIDTH>(0);
    outputs[i].template set_slc<32>(0, data);
}
```

Listing 4.26: tb → `main.cpp` - Run conv2d accelerator and fetch its output data.

When all outputs have been stored into the corresponding array, the testbench can validate the results by invoking the golden function with the same inputs and weights provided to the accelerator.
The software convolution has a similar algorithm to the one employed by the accelerator. The main differences regard the datatypes, which are now `int` and `double`, and by the fact that overflows are handled by a few `if` statements in order to make the results compliant to the fixed point representation of output data in the accelerator.

```
void conv2d_tb(FPDATA_IN *input, double *output)
{
    // Parameters
    const int filt = 3;
    const int kern = 3;
    // ...
    const int n_w_out = (n_w + 2 * pad - kern)/(stride) + 1;
    const int n_h_out = (n_h + 2 * pad - kern)/(stride) + 1;

    // Padding
    double paddedBuffer[n_w_in * n_h_in * n_c];
    unsigned indexInp = kern * kern * n_c * filt;
    for (unsigned chan = 0; chan < n_c; chan++) {
        for (unsigned row = 0; row < n_w_in; row++) {
            for (unsigned col = 0; col < n_h_in; col++) {
                unsigned index_pad = chan*n_w_in*n_h_in + row*n_h_in + col;
                if ((row >= pad) && (col >= pad) && (col < n_h_in - pad)
                     && (row < n_w_in - pad)) {
                    paddedBuffer[index_pad] = input[indexInp].to_double();
                    indexInp++;
                }
                else {
                    paddedBuffer[index_pad] = 0;
                }
            }
```

```
        }
    }

    double acc = 0;
    unsigned x, y;
    // Set golden output
    for (unsigned fl = 0; fl < filt; fl++) {
        for (unsigned i = 0; i < n_w_out; i++) {
            for (unsigned j = 0; j < n_h_out; j++) {
                x = i * stride;
                y = j * stride;
                for (unsigned m = 0; m < kern; m++) {
                    for (unsigned n = 0; n < kern; n++) {
                        for (unsigned k = 0; k < n_c; k++) {
                            unsigned index1, index2;
                            index1 = fl * kern * kern * n_c + k * kern *
                                     kern + m * kern + n;
                            index2 = n_w_in* n_h_in* k + x* n_w_in + y;
                            acc += input[index1].to_double() *
                                   paddedBuffer[index2];
                        }
                        y++;
                    }
                    x++;
                    y = j * stride; // Restart column position
                }
                unsigned index3 = n_w_out* n_h_out* fl + i* n_w_out + j;
                // Overflow handling
                if (acc >= 32768)
                    output[index3] = 32767.999985;
                else if (acc < -32768)
                    output[index3] = - 32768;
                else
                    output[index3] = acc;
                acc = 0;
            }
        }
    }
}
```

Listing 4.27: tb → `main.cpp` - Golden function.

Each output of the 2D-convolution accelerator is compared to the corresponding golden value. If the difference between the two is larger than a very small `allowed_error`, the error counter is increased. If any error is detected, the testbench return value is set to 1, otherwise it is set to 0.

```
//Testbench data
double gold_outputs[output_size * BATCH_MAX];

// Testbench return value (0 = PASS, non-0 = FAIL)
int rc = 0;

// Validation
for (unsigned i = 0; i < conf_info_data.batch; i++) {
```

```
    conv2d_tb(inputs + i * input_size, gold_outputs + i * output_size);
}
unsigned errors = 0;
double allowed_error = 0.001;

for (unsigned i = 0; i < conf_info_data.batch * output_size; i++) {
    float gold = gold_outputs[i];
    FPDATA_OUT data = outputs[i];
    // Calculate absolute error
    double error_it = abs_double(data.to_double() - gold);
    if (error_it > allowed_error) {
        ESP_REPORT_INFO(VON, "[%u]: %f (expected %f)", i, data.to_double(),
    gold);
        errors++;
    }
}
if (errors > 0) {
    ESP_REPORT_INFO(VON, "Validation: FAIL (errors %u / total %u)", errors,
    output_size * ESP_TO_UINT32(conf_info_data.batch));
    rc = 1;
} else {
    ESP_REPORT_INFO(VON, "Validation: PASS");
    rc = 0;
}
ESP_REPORT_INFO(VON, "  - errors %u / total %u", errors, output_size *
    ESP_TO_UINT32(conf_info_data.batch));

CCS_RETURN(rc);
```

Listing 4.28: tb → `main.cpp` - Output validation.

In addition, the `ESP_REPORT_INFO` function, which is provided by ESP in the softmax documentation, can be exploited to print out information regarding the accelerator's configuration and validation.
The message in Lst. 4.29 is the result of a working conv2d accelerator.

```
# Info: main(): --------------------------------
# Info: main(): ESP - Conv2D [Catapult HLS C++]
# Info: main():       Single block
# Info: main(): --------------------------------
# Info: main(): Configuration:
# Info: main():   - batch: 4
# Info: main():   - n_w: 5
# Info: main():   - n_h: 5
# Info: main():   - n_c: 3
# Info: main():   - kern: 3
# Info: main():   - filt: 3
# Info: main():   - same: 1
# Info: main():   - stride: 1
# 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():   - Batch size: 156
# Info: main():   - memory in (words): 624
```

```
# Info: main():   - memory out (words): 300
# Info: main(): --------------------------------
# Info: main(): Validation: PASS
# Info: main():   - errors 0 / total 300
# Info: main(): --------------------------------
```

Listing 4.29: Message printed on the terminal by the conv2d testbench.

When simulating the RTL implementation of the accelerator, QuestaSim allows to visualize the waveforms of its signals. This can be very useful to test the ESP accelerator behavior. In particular, we can observe how and at which point: the configuration parameters are set, the input data are fetched from the DMA channel, the output values are written into the DMA. By measuring the number of clock cycles between the start and the end of the accelerator execution we can also derive information regarding the accelerator performance, i.e. the execution latency.

An example of the output of a QuestaSim simulation with batch = 4, input tensor = $5 \times 5 \times 3$, weight tensor = $3 \times 3 \times 3 \times 3$, same padding = 1 and stride=1 is shown in Fig. 4.8, where the two cursors are positioned at the start and at the end of the accelerator execution.

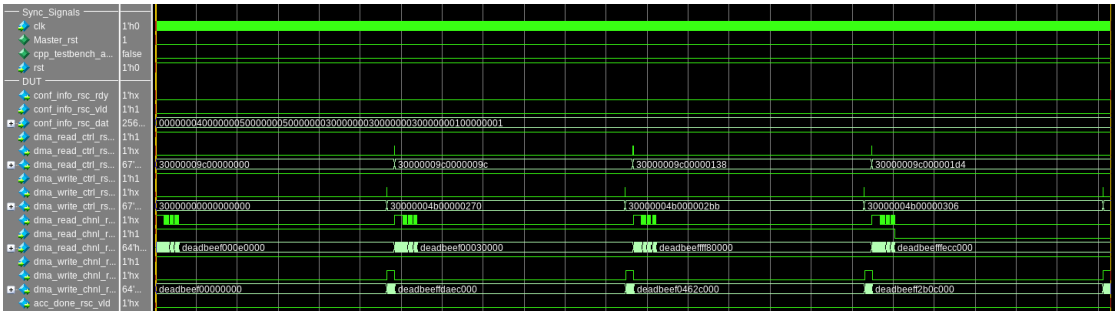

Figure 4.8: Complete Questasim simulation for the sequential implementation of the 2D-convolution accelerator.

4.3.1 Validation Results

Multiple tests have been made with different run-time parameters to validate both the sequential and hierarchical architectures of the 2D-convolution accelerator. In particular, we focused on the area and on the latency of each accelerator implementation. The former can be roughly derived during synthesis by Catapult. The latter is obtained from the QuestaSim RTL simulation of the accelerator and it is measured as the number of clock

periods between the beginning and the end of the accelerator execution. All the experiments are reported in Tab. 4.1, but the real value of FPGA resource usage is obtained by the Vivado reports after place and route, i.e. bitstream generation.

First, we considered the sequential implementation of our accelerator on a single batch of weights and features. We changed a few other parameters to check whether the area, latency and results would be affected by them. As reported in Tab. 4.1, we tested an accelerator with different input dimensions, from smallest to largest, while also testing different padding and stride values. As expected, no error is ever detected and the estimated area of the design is not affected by the user-defined parameters. However, parameters affect the latency which increases as the number of computations grows.

The results show how the number of weights, features and output values greatly affects latency. For the same reason, the number of batch iterations can also increase latency up to 16 times, the maximum number of batches allowed by the design.

Features	**Filters**	**Same**	**Stride**	**Area** [μm^2]	**Lat.** [***clks***]
1×1×1	1×1×1×1	0	1	83,268.3	12
5×5×3	3×3×3×3	0	1	83,268.3	918
5×5×3	3×3×3×3	1	1	83,268.3	2335
18×18×32	7×7×32×32	0	2	83,268.3	1,868,038
18×18×32	7×7×32×32	0	1	83,268.3	7,290,502

Table 4.1: Sequential implementation of the 2D-convolution accelerator with different run-time parameters and a single batch. Area estimation by Cataputlt HLS.

In the sequential implementation computations on each batch are performed, as the name may suggest, in a sequential way. On the other hand, the hierarchical implementation allows to perform the load phase of a new batch while the compute phase of a previous batch is still being executed. So if a new batch is ready to be computed, but the compute function has not finished yet, the load function stalls until the compute block loads the new batch. Vice versa, if the compute function has already finished to process the current batch, but the load function has not finished to read the new batch,

the compute function stalls until the new batch is ready. Data dependencies are not violated because data are stored into FIFO-like `ac_channel` ports which come with the necessary synchronization signals. Therefore, we expect a better overall latency when dealing with a number of batches greater than 1.
Tab. 4.2 shows a comparison between estimated area and latency of the sequential and hierarchical implementations of the same accelerator. The device under test performs the 2D convolution between $18 \times 18 \times 32$ input tensors and $7 \times 7 \times 32 \times 32$ filters with same padding and a stride value equal to 1.

# Batches	Architecture	Area		Latency [ms]	
1	sequential	83268.3	-	145.81	-
1	hierarchical	91576.3	+10%	145.81	0%
8	sequential	83268.3	-	1166.47	-
8	hierarchical	91576.3	+10%	1156.54	-0.85%
16	sequential	83268.3	-	2332.95	-
16	hierarchical	91576.3	+10%	2311.67	-0.91%

Table 4.2: Comparison between sequential and hierarchical implementations for different numbers of batches - same = 1, stride = 1, clock frequency = 50 MHz.

The table shows how for multiple batch iterations the hierarchical architecture improves performance at the cost of area. When dealing with 16 batches the performance is improved by 0.91%. Clearly, for a single batch the latency is the same and there is no advantage with respect to the sequential implementation.

4.4 ESP Heterogeneous Integration

ESP does not only facilitate the design flow of an accelerator, but it also eases its integration into heterogeneous SoCs.
ESP supports many FPGA boards [34]. In this thesis, we chose to work with the `proFPGA XC7V2000T` FPGA module.

To create an SoC, we need to launch the ESP scripts from inside the folder

whose name corresponds to the FPGA model we are going to use. When the SoC-design GUI provided by ESP is open, it shows a default SoC configuration, where tiles are organized in a 2×2 grid containing: the processor core, the memory controller and memory channel, the auxiliary tile, and an empty tile.

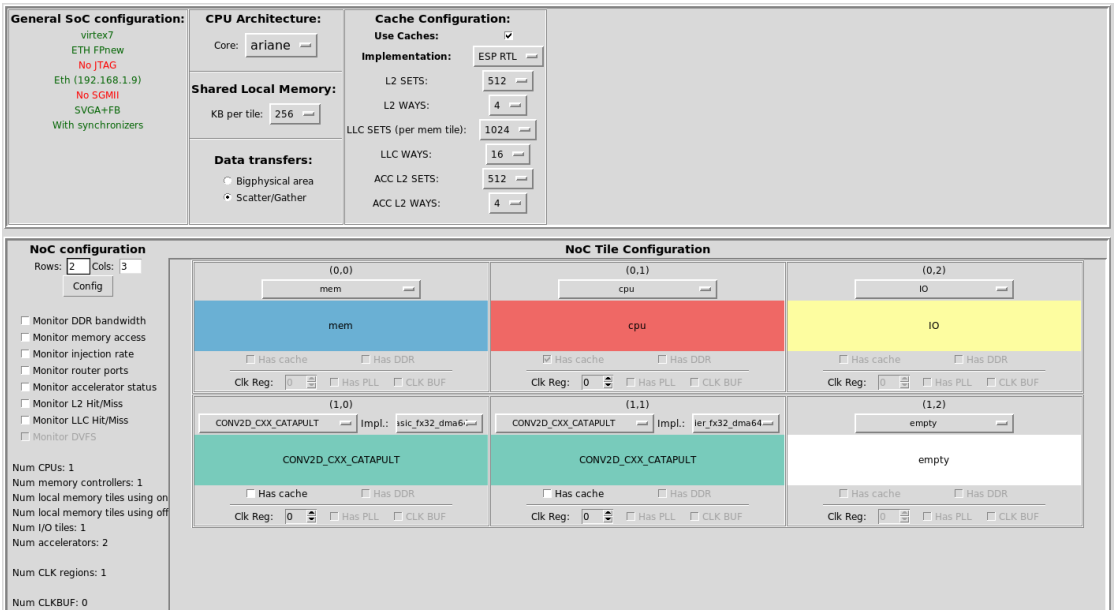

Figure 4.9: SoC design in the ESP GUI with both conv2d accelerator architectures.

For our design we selected the modern Ariane RISC-V processor core and replaced the empty tile with the accelerator tile containing the RTL implementation of one of our 2D-convolution accelerators. Both the sequential and the hierarchical architectures can be selected. If we increase the grid to a 2×3 mesh, both versions of the accelerator can be integrated in the same SoC chip. The resulting SoC configuration is shown in Fig. 4.9.

Based on the SoC configuration, the corresponding RTL implementation is automatically generated. A full-system RTL simulation can be performed and, thanks to a bare-metal test application which runs directly on top of the processor tile, we can validate our accelerators at system level, i.e. taking into account the processor and all the other peripherals.

The bare-metal test application is in the ***sw*** folder of the accelerator. We

again started from the softmax template and then we customized it to be similar to the testbench.
Indeed, as reported in Lst. 4.30, the `convd2d_cxx.c` bare-metal application allows to set the parameters as global variables and to indicate the address of each corresponding 32-bit memory-mapped register. Each address is increased by four bytes starting from `0x40` and these registers are arranged following to the order specified in `conf_info.hpp`.
In addition, the bare-metal application allows to set the name and identification number of the accelerator in case more accelerators of the same kind are present in the SoC.

```
// Accelerator identifiers
#define SLD_CONV2D_CXX 0x145
#define DEV_NAME "sld,conv2d_cxx_catapult"

/* User-defined parameters */
/* <<--params-->> */
const int32_t batch = 1;
const int32_t n_w = 18;
const int32_t n_h = 18;
const int32_t n_c = 32;
const int32_t kern = 7;
const int32_t filt = 32;
const int32_t same = 0;
const int32_t stride = 1;

/* Other parameters derived from user-defined ones */
const uint8_t pad = ((stride * (n_w - 1) - n_w + kern) / 2) * same;
const uint8_t n_w_in = n_w + 2 * pad;
const uint8_t n_h_in = n_h + 2 * pad;
const uint8_t n_w_out = (n_w + 2 * pad - kern)/(stride) + 1;
const uint8_t n_h_out = (n_w + 2 * pad - kern)/(stride) + 1;
const uint16_t IN_SIZE = n_w * n_h * n_c + kern * kern * n_c * filt;
const uint16_t OUT_SIZE = n_w_out * n_h_out * filt;

/* User defined registers */
/* <<--regs-->> */
#define CONV2D_CXX_STRIDE_REG 0x5c
#define CONV2D_CXX_SAME_REG 0x58
#define CONV2D_CXX_FILT_REG 0x54
#define CONV2D_CXX_KERN_REG 0x50
#define CONV2D_CXX_N_C_REG 0x4c
#define CONV2D_CXX_N_H_REG 0x48
#define CONV2D_CXX_N_W_REG 0x44
#define CONV2D_CXX_BATCH_REG 0x40
```

Listing 4.30: `conv2d_cxx.c` - Constant variables.

Based on the accelerator identifiers, the `probe` function[1] can be used to check whether the device has been correctly integrated in the SoC. Useful information about the device under test, including its location, is also stored in the `espdevs` structure.

```
int ndev;
struct esp_device *espdevs;

ndev = probe(&espdevs, VENDOR_SLD, SLD_CONV2D_CXX, DEV_NAME);

if (ndev == 0) {
    print_uart("conv2d_cxx not found\n");
    return 0;
}
```

Listing 4.31: `conv2d_cxx.c` - Probe function invocation.

The memory spaces needed to store the golden outputs and the inputs/outputs of the accelerator are allocated using `aligned_malloc`. In addition, the memory region needed to store the accelerator's input and output data is split into chunks and the corresponding page table is populated accordingly, as shown in Lst. 4.32.

```
typedef int64_t token_t;
token_t *gold;
token_t *mem;
unsigned **ptable;

// Allocate memory
gold = aligned_malloc(out_size);
mem = aligned_malloc(mem_size);

// Alocate and populate page table
ptable = aligned_malloc(NCHUNK(mem_size) * sizeof(unsigned *));
for (i = 0; i < NCHUNK(mem_size); i++)
    ptable[i] = (unsigned *) &mem[i * (CHUNK_SIZE / sizeof(token_t))];

// Initialize the memory
init_buf(mem, gold);
```

Listing 4.32: `conv2d_cxx.c` - Memory allocation and initlialization.

Thanks to the `init_buf` function, the elements of the input and weight tensors are stored into the external memory. However, `init_buf` doesn't generate only the input values to write into the memory, but it also calls a

[1]Most functions, structures and constants used by the bare-metal application are defined in the `esp_accelerator.h` and `esp_probe.h` headers files available at: `https://github.com/sld-columbia/esp`.

golden function (`conv2d_sw`), which is very similar to the one exploited by the previously mentioned C++ testbench, in order to set a golden output tensor. Thus, the golden output values are stored in the external memory so that they can be easily accessed during validation.

```
static void init_buf (token_t *in, token_t * gold)
{
    int i, j;
    // initialize input
    for (i = 0; i < batch; i++) {
        for (j = 0; j < IN_SIZE; j++) {
            float data_flt = ((i * IN_SIZE + j) % 200) * 0.25 - 25;
            token_t data_fxd = 0xdeadbeef00000000 | float_to_fixed32(
    data_flt, 16);
            in[i * in_words_adj + j] = (token_t) data_fxd;
        }
    }
    // set golden output
    for (i = 0; i < batch; i++) {
        conv2d_sw(in + (i*IN_SIZE), gold + (i*OUT_SIZE));
    }
}
```

Listing 4.33: `conv2d_cxx.c` - `init_buf` function.

Regarding the invocation of the 2D-convolution accelerator, it is not as straightforward as in the C++ testbench. First, the processor checks whether the DMA and the *Translation Look-aside Buffer* (*TLB*) are enabled or not, as reported in Lst. 4.34.

```
dev = &espdevs;

// Check DMA capabilities
if (ioread32(dev, PT_NCHUNK_MAX_REG) == 0) {
    print_uart("  -> scatter-gather DMA is disabled. Abort.\n");
    return 0;
}

// Check TLB capabilities
if (ioread32(dev, PT_NCHUNK_MAX_REG) < NCHUNK(mem_size)) {
    print_uart("  -> Not enough TLB entries available. Abort.\n");
    return 0;
}
```

Listing 4.34: `conv2d_cxx.c` - Check DMA and TLB capabilities.

Then, the processor overwrites the accelerator registers with both common (for instance the page table's address or the accelerator's coherency model) and accelerator-specific configuration parameters, as shown in Lst. 4.35.

```
// Pass common configuration parameters
iowrite32(dev, SELECT_REG, ioread32(dev, DEVID_REG));
iowrite32(dev, COHERENCE_REG, ACC_COH_NONE);
iowrite32(dev, PT_ADDRESS_REG, (unsigned long long) ptable);
iowrite32(dev, PT_NCHUNK_REG, NCHUNK(mem_size));
iowrite32(dev, PT_SHIFT_REG, CHUNK_SHIFT);

// Pass accelerator-specific configuration parameters
iowrite32(dev, CONV2D_CXX_BATCH_REG, batch);
iowrite32(dev, CONV2D_CXX_N_W_REG, n_w);
iowrite32(dev, CONV2D_CXX_N_H_REG, n_h);
iowrite32(dev, CONV2D_CXX_N_C_REG, n_c);
iowrite32(dev, CONV2D_CXX_KERN_REG, kern);
iowrite32(dev, CONV2D_CXX_FILT_REG, filt);
iowrite32(dev, CONV2D_CXX_SAME_REG, same);
iowrite32(dev, CONV2D_CXX_STRIDE_REG, stride);
```

Listing 4.35: `conv2d_cxx.c` - Pass configuration parameters to the accelerator.

Later, the accelerator is started by overwriting the register (`CMD_REG`) that contains accelerator-specific commands with the proper `CMD_MASK_START` bitmask. The status register (`STATUS_REG`) is continuously read (*polling*) to determine when the task execution is completed, as shown in Lst. 4.36.

```
// Run accelerator
iowrite32(dev, CMD_REG, CMD_MASK_START);

// Wait for completion
done = 0;
while (!done) {
    done = ioread32(dev, STATUS_REG);
    done &= STATUS_MASK_DONE;
}
iowrite32(dev, CMD_REG, 0x0);
```

Listing 4.36: `conv2d_cxx.c` - Start the accelerator and wait for completion.

Finally, the accelerator's results are validated by the `validate_buf` function which compares them with the golden outputs and the bare-metal application frees the allocated space before termination.

```
/* Validation */
errors = validate_buf(&mem[out_offset], gold);
if (errors)
    print_uart("  ... FAIL\n");
else
    print_uart("  ... PASS\n");

aligned_free(ptable);
aligned_free(mem);
aligned_free(gold);
```

```
return 0;
```

Listing 4.37: `conv2d_cxx.c` - Output validation and allocated space is freed.

The simulation of the execution of the bare-metal application on the SoC is done by QuestaSim. The simulation time is quite long since the 2D convolution is computed in both software and hardware, the former being much slower than the latter. However, after approximately 40 minutes (time depends on the accelerator run-time parameters) the outputs are validated and the following message is printed out:

```
[...]
# ESP-Ariane boot loader
# Scanning device tree...
# [probe] sld,conv2d_cxx_catapult.0 registered
#       Address   : 0x60010000
#       Interrupt : 6
# [probe] sld,conv2d_cxx_catapult.1 registered
#       Address   : 0x60010100
#       Interrupt : 7
# Found 00000002 devices: sld,conv2d_cxx
#   memory buffer base-address = 00000000A0100DA0
#   ptable = 00000000A01014F0
#   Generate input...
#        input  data @00000000A0100DA0
#   gold output data @00000000A0100B30
#   ... input ready!
#   -> Non-coherent DMA
# [probe] sld,llc_cache.0 registered
#       Address   : 0x6000e000
#       Interrupt : 12
# [probe] sld,l2_cache.0 registered
#       Address   : 0x6000d900
#       Interrupt : 12
#   Start...
#   Done
#   validating...
#   gold output data @00000000A0100B30
#        output data @00000000A0101280
#   total errors 00000000
#   ... PASS
#   -> Non-coherent DMA
# [probe] sld,llc_cache.0 registered
#       Address   : 0x6000e000
#       Interrupt : 12
# [probe] sld,l2_cache.0 registered
#       Address   : 0x6000d900
#       Interrupt : 12
#   Start...
#   Done
#   validating...
#   gold output data @00000000A0100B30
```

```
#        output data @00000000A0101280
#   total errors 00000000
#   ... PASS
# DONE
# ** Program Completed!
```

Listing 4.38: `conv2d_cxx.c` - Message printed by the bare-metal application.

4.5 FPGA Prototyping

Finally, the SoC design with either one 2D-convolution accelerator or both implementations can be deployed on the FPGA board. The corresponding *bitstream* (`top.bit`) is generated leveraging `Xilinx Vivado`.

The FPGA is connected via ethernet to a remote server (*host*) that can be accessed via *ssh tunnelling*. The UART interface of the SoC can be accessed with `Minicom`, a serial communication program, because the FPGA has a UART interface board connected to the host computer via USB. The same bare-metal test application used for RTL simulation can be run on the FPGA and its output is displayed on the host's screen thanks to Minicom.

As reported in Fig. 4.10, the FPGA output message corresponds to the one that had been printed out during the RTL simulation in QuestaSim, meaning that our SoC design has been correctly prototyped.

The bare-metal test application exploits the following function to measure the number of clock cycles needed to perform the 2D convolution in both software and hardware; `asm_volatile` is used to run assembly instructions inside a C/C++ code.

```
static inline uint64_t get_counter() {
    uint64_t counter;
    asm volatile (
        "li t0, 0;"
        "csrr t0, mcycle;"
        "mv %0, t0"
        : "=r" ( counter )
        :
        : "t0"
    );
    return counter; }
```

Listing 4.39: `conv2d_cxx.c` - `get_counter` function.

```
Welcome to minicom 2.6.2

OPTIONS: I18n
Compiled on Jun 10 2014, 03:20:53.
Port /dev/ttyUSB0, 14:54:56

Press CTRL-A Z for help on special keys

ESP-Ariane boot loader

Scanning device tree...
[probe] sld,conv2d_cxx_catapult.0 registered
                Address   : 0x60010000
                Interrupt : 6
[probe] sld,conv2d_cxx_catapult.1 registered
                Address   : 0x60010100
                Interrupt : 7
Found 00000002 devices: sld,conv2d_cxx
  memory buffer base-address = 00000000A0100DA0
  ptable = 00000000A01014F0
  nchunk = 00000001
  Generate input...
       input  data @00000000A0100DA0
  gold output data @00000000A0100B30
  ... input ready!
        -> Non-coherent DMA
[probe] sld,llc_cache.0 registered
                Address   : 0x6000e000
                Interrupt : 12
[probe] sld,l2_cache.0 registered
                Address   : 0x6000d900
                Interrupt : 12
  Start...
  Done
  validating...
  gold output data @00000000A0100B30
       output data @00000000A0101280
  total errors 00000000
  ... PASS
        -> Non-coherent DMA
[probe] sld,llc_cache.0 registered
                Address   : 0x6000e000
                Interrupt : 12
[probe] sld,l2_cache.0 registered
                Address   : 0x6000d900
                Interrupt : 12
  Start...
  Done
  validating...
  gold output data @00000000A0100B30
       output data @00000000A0101280
  total errors 00000000
  ... PASS
DONE
```

Figure 4.10: Output of the bare-metal test application running on the SoC synthesized on the FPGA.

Tab. 4.3 shows the comparison between the execution times of the 2D convolution being run on the SoC's processor core and being computed by the sequential implementation of our custom accelerator, for a 50 MHz clock frequency.

Features	**Filters**	**Proc. Lat. [ms]**	**Acc. Lat. [ms]**	**Reduc.**
5×5×3	3×3×3×3	0.5205	0.0184	96.47 %
18×18×3	7×7×3×3	3358.21	145.81	95.65 %

Table 4.3: Latency comparison of the 2D-convolution algorithm between software execution and our accelerator computation for different input dimensions - batch = 1, same = 0, stride = 1 - clock frequency = 50 Mhz.

As expected, the hardware accelerator provides a remarkable latency improvement. In particular, compared to the software execution, our 2D-convolution accelerator provides a 96.47 % latency reduction with the smaller input and a 95.65 % improvement with the larger one.
This further demonstrates with numbers that accelerating convolutional layers in hardware leads to an enormous latency reduction, enabling CNNs inference in many latency critical scenarios, such as autonomous driving.

Chapter 5

Optimized 2D-Convolution Accelerator

In the previous chapter we illustrated two different implementations of our 2D-convolution accelerator. The results reported in Tab. 4.2 show how the hierarchical architecture provides a latency decrease at the cost of area with respect to the sequential implementation, but only for multiple batch iterations. However, when convolving a single batch of tensors, the hierarchical solution does not provide any advantage whatsoever. This is due to the fact that, as shown in Fig. 4.7, all input data fetched from the external memory are saved inside the corresponding PLMs during the load phase before being transferred to the compute block through `ac_channels`. Similarly, the results are sent back to the external memory only after they have all been computed and temporarily stored within the outputs' PLM. For this reason, all phases are executed in a sequential way just as in the sequential implementation, despite the additional hardware.

5.1 2D Convolution with Sliding Window

Our aim is to fetch/store data from/to the external memory as the compute phase is being executed, i.e. in a transparent way. Thus, our datapath needs to carry out the convolution operation as soon as input data are fetched.

This can be achieved by modifying the high-level description of the hierarchical implementation of the 2D-convolution accelerator. Let us call this new implementation "*optimized*".

In particular, we employed a *sliding-window* architecture which allows to reuse feature map data while fetching only the new elements needed by the current iteration of the algorithm. In this architecture only one kernel is fetched from the external memory and sent to the `compute` block, while *line buffers* are exploited to transfer only the feature map elements that are needed right away.

Since our accelerator can handle a 7×7×32×32 weight tensor and a 18×18×32 input tensor, a register with 7×7 elements is enough to store the kernel. Consequently, 7 line buffers are used to temporarily hold 18 feature map elements each.

As the first set of partial sums is being stored into the accumulation buffer in the compute block, the load block concurrently fetches either one or two rows of features, depending on the stride parameter. The newly fetched elements are then replaced into the line buffers in a circular way.

Usually, the same input tensor is convolved with multiple filters; this would require each input channel to be fetched from the external memory more than once. In order to avoid to increase the number of accesses to the outside memory, when features are fetched for the first time they are also saved in a *features PLM*, in addition to being stored into the line buffers as previously described. In this way, feature map elements can be quickly reloaded into the line buffers from this local memory when needed.
The resulting load algorithm of our `conv2dlb` accelerator is reported in Lst. 5.1.

```
for (uint8_t fl = 0; fl < FILT_MAX; fl++) {
    for (uint8_t k = 0; k < N_C_MAX; k++) {
        // Fetch Kernel
        // Configure DMA read channel (CTRL)
        dma_read_data_index = batch_size*b + kern*kern*n_c*fl + kern*kern*k;
        dma_read_data_length = kern * kern;
        dma_info_t dma_read_info = {dma_read_data_index,
    dma_read_data_length, DMA_SIZE};
        bool dma_read_ctrl_done = false;
LOAD_CTRL_LOOP1:
```

```
        do {dma_read_ctrl_done = dma_read_ctrl.nb_write(dma_read_info);}
        while (!dma_read_ctrl_done);
        // Force serialization between DMA control and DATA transfer
        if (dma_read_ctrl_done) {
            for (uint8_t m = 0; m < KERN_MAX; m++) {
                for (uint8_t n = 0; n < KERN_MAX; n++) {
                    FPDATA_IN data;
                    uint8_t index_f = m * kern + n;

                    ac_int<DATA_WIDTH, false> data_ac = dma_read_chnl.read()
    .template slc<DATA_WIDTH>(0);
                    data.set_slc(0, data_ac);
                    plm_tmp_f.data[index_f] = data;
                    if (n == kern - 1) break;
                }
                if (m == kern - 1) break;
            }
            plm_kernel.write(plm_tmp_f);
        }
        // Fetch Input Channel
        if (fl == 0) {
            // Configure DMA read channel (CTRL)
            dma_read_data_index = batch_size*b + filters_size + k*n_w*n_h;
            dma_read_data_length = n_w * n_h;
            dma_read_info = {dma_read_data_index, dma_read_data_length,
    DMA_SIZE};
            dma_read_ctrl_done = false;
LOAD_CTRL_LOOP2:
            do {dma_read_ctrl_done = dma_read_ctrl.nb_write(dma_read_info);}
            while (!dma_read_ctrl_done);
        }
        // Indicate when to write the line buffers for the first time
        uint8_t write_buf = kern - 1;
        if (dma_read_ctrl_done) {
            for (uint8_t row = 0; row < N_W_IN_MAX; row++) {
                for (uint8_t col = 0; col < N_H_IN_MAX; col++) {
                    FPDATA_IN data;
                    if (fl == 0) {
                        // Padding
                        uint16_t index_in = k * n_w_in * n_h_in +
                                            row * n_w_in + col;
                        if ((row>=pad) && (col>=pad) && (col<n_h_in-pad)
                             && (row< n_w_in-pad)) {
                            ac_int<DATA_WIDTH, false> data_ac =
    dma_read_chnl.read().template slc<DATA_WIDTH>(0);
                            data.set_slc(0, data_ac);
                         }
                         else {
                            data = 0;
                         }
                        plm_tmp_in.data[index_in] = data;
                    }
                    else {
                        uint16_t index_in = k * n_w_in * n_h_in +
                                            row * n_w_in + col;
                        data = plm_tmp_in.data[index_in];
```

```
                    }
                    uint8_t row_norm;
                    switch (kern) {
                        case 1:
                            row_norm = row % 1;
                            break;
                        case 3:
                            row_norm = row % 3;
                            break;
                        case 5:
                            row_norm = row % 5;
                            break;
                        case 7:
                            row_norm =  row % 7;
                            break;
                    }
                    buf_tmp_lin.data[row_norm][col] = data;
                    if (col == n_h_in - 1) break;
                }
                if (row == write_buf) {
                    buf_linear.write(buf_tmp_lin);
                    // next buffer overwrite depends on stride value
                    write_buf += stride;
                }
                if (row == n_w_in - 1) break;
            }
        }
        if (k == n_c - 1) break;
    }
    if (fl == filt - 1) break;
}
```

Listing 5.1: `conv2dlb.cpp` - Load Phase.

The `compute` algorithm must also be slightly modified, so that each weight is convolved with the correct line buffer. This is done by means of a `switch` statement, synthesized as a *multiplexer* which selects for each MAC operation the respective input channel's row. The complete algorithm is shown in Lst. 5.2.

```
for (uint8_t fl = 0; fl < FILT_MAX; fl++) {
    for (uint8_t k = 0; k < N_C_MAX; k++) {
        plm_tmp_f = plm_kernel.read();
        for (uint8_t i = 0; i < N_W_OUT_MAX; i++) {
            buf_tmp_lin = buf_linear.read();
            for (uint8_t j = 0; j < N_H_OUT_MAX; j++) {
                FPDATA_ACC acc = 0;
                uint8_t x = i * stride; // input row
                uint8_t y = j * stride; // input col
                for (uint8_t m = 0; m < KERN_MAX; m++) {
                    for (uint8_t n = 0; n < KERN_MAX; n++) {

                        uint8_t index_f = m * kern + n;
                        uint8_t x_tmp;
```

```
                        switch (kern) {
                            case 1:
                                x_tmp = x % 1;
                                break;
                            case 3:
                                x_tmp = x % 3;
                                break;
                            case 5:
                                x_tmp = x % 5;
                                break;
                            case 7:
                                x_tmp = x % 7;
                                break;
                        }
                        acc += buf_tmp_lin.data[x_tmp][y] *
                               plm_tmp_f.data[index_f];
                        y++;
                        if (n == kern - 1) break;
                    }
                    x++;
                    y = j * stride; // Restart column position
                    if (m == kern - 1) break;
                }
                if (k == 0)
                    buf_tmp_acc.data[i][j] = acc;
                else
                    buf_tmp_acc.data[i][j] += acc;
                if (k == n_c - 1) {
                    var_tmp_out = buf_tmp_acc.data[i][j];
                    var_output.write(var_tmp_out);
                }
                if (j == n_h_out - 1) break;
            }
            if (i == n_w_out - 1) break;
        }
        if (k == n_c - 1) break;
    }
    if (fl == filt - 1) break;
}
```

Listing 5.2: `conv2dlb.cpp` - Compute Phase.

It should also be noted that, as soon as an element of the accumulation buffer is completed, it is sent to the store block, while the partial sums of the MAC operations remain stored into the accumulation buffer. This enables the execution of both load and store phases to be transparent to the actual computation of the 2D convolution.
The store block is the same as in the unoptimized 2D-convolution accelerator with the exception that its corresponding PLM is implemented as a 32-bit register, since it contains just a single element of the output tensor instead of an 18×18×32 RAM block, as reported in Lst. 5.3.

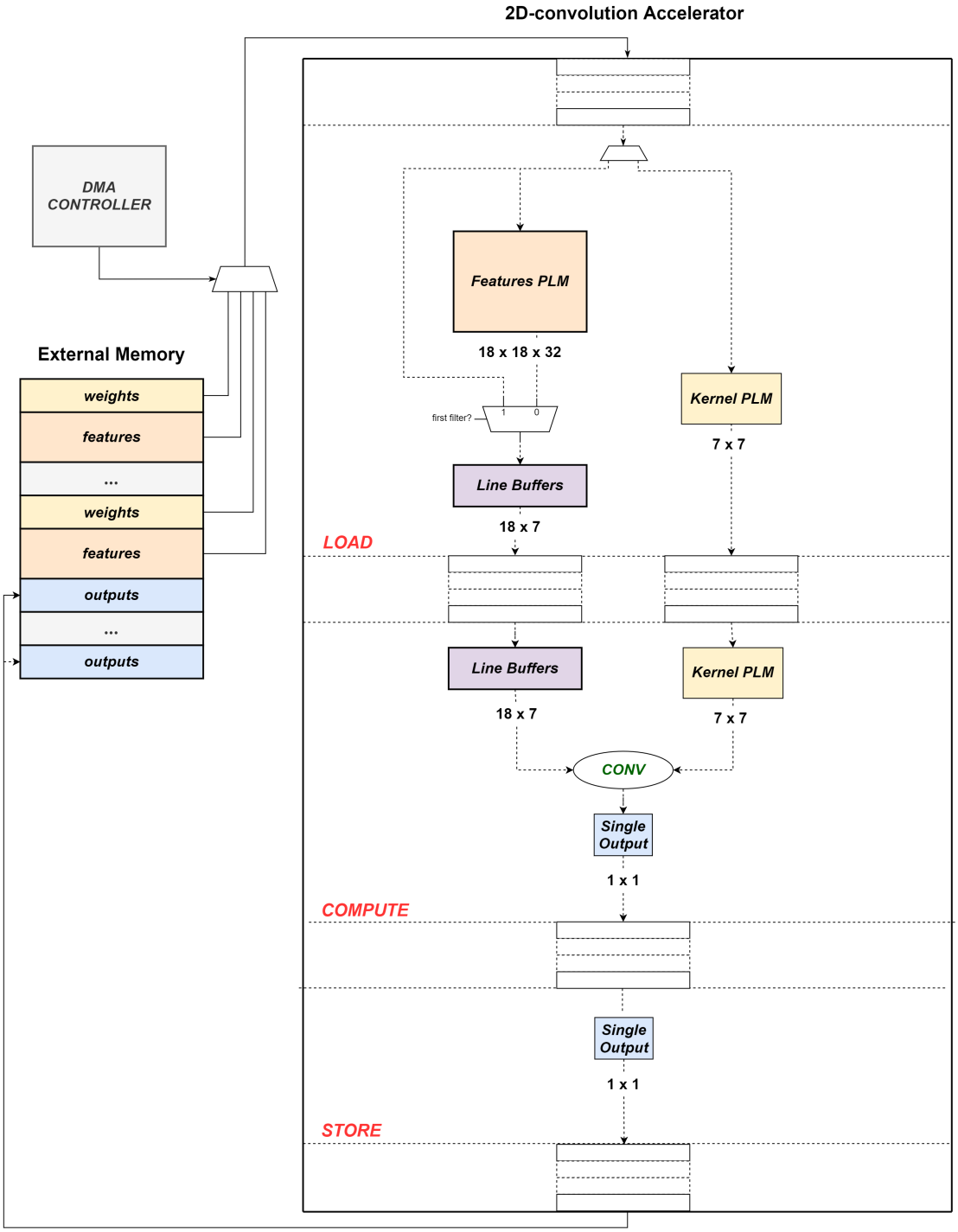

Figure 5.1: 2D-convolution Accelerator: optimized architecture.

```
for (uint16_t i = 0; i < OUTPUTS_SIZE_MAX; i++) {
    FPDATA_OUT var_tmp_out = var_output.read();
    FPDATA_OUT data = var_tmp_out;
    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));
    dma_write_chnl.write(data_ac);

    if (i == dma_write_data_length - 1) break;
}
```

Listing 5.3: `conv2dlb.cpp` - Store Phase.

The architecture of the optimized 2D-convolution accelerator is shown in Fig. 5.1.

5.2 Synthesis Results and Comparisons

As previously seen, the 2D-convolution accelerator has been optimized to reduce its latency by making the `load`, `compute` and `store` phases run in parallel. However, by comparing the optimized hierarchical implementation shown in Fig. 5.1 with the unoptimized hierarchical architecture in Fig. 4.7, we might notice that the dimensions of PLMs and shared memories have also been reduced.

For our accelerator design, we decided to prioritize latency as design goal. For this reason we synthesized PLMs as registers rather than as RAM blocks, with the exception of the features PLM. Using registers for the two Kernel PLMs, the two Line Buffers, the two Single Output registers and all the corresponding `ac_channels` was required to maintain the Initiation Interval equal to 1 for the three phases, since using RAM blocks would have taken more than one clock cycle to pass data between two consecutive phases.

Architecture	**Slice Logic** [%]	**Block RAM** [%]	**DSP** [%]
Sequential	20.31	39.32	2.31
Hierarchical	20.33 (+ 0.02)	50.81 (+ 11.49)	2.31 (+ 0.00)
Optimized	22.70 (+ 2.41)	25.70 (- 13.62)	2.13 (- 0.18)

Table 5.1: FPGA resource usage of an SoC for each architecture. The sequential implementation is taken as reference.

Tab. 5.1 shows the FPGA resource usage obtained by the Vivado reports after place and route for an SoC integrating the sequential, the hierarchical or the optimized implementation of our 2D-convolution accelerator. With respect to the sequential implementation of our 2D-convolution accelerator the unoptimized hierarchical architecture yields a 11.49% block RAM usage increase. However, the optimized architecture requires a slight increase of slice logic being used (+2.41 %) because of the additional registers and logic (MUXs and registers for Kernel PLM, Line Buffers, Output and the corresponding `ac_channels`), but, on the other hand, the block RAM usage is significantly reduced (−13.62 %). Since the compute phase has not been substantially altered, the number of Digital Signal Processing (*DSP*) blocks being employed is not affected as much.

The comparison between unoptimized and optimized architectures of our 2D-convolution accelerator in terms of latency, for batch = 1, is reported in Tab. 5.2 and it is measured as the number of clock periods between the beginning and the end of the accelerator execution.

Arch.	**Features**	**Filters**	**Same**	**Stride**	**Latency [*clks*]**
Seq./Hier.	5×5×3	3×3×3×3	0	1	918
Opt.	5×5×3	3×3×3×3	0	1	760 (-17.21%)
Seq./Hier.	5×5×3	3×3×3×3	1	1	2335
Opt.	5×5×3	3×3×3×3	1	1	2063 (-11.65%)
Seq./Hier.	18×18×32	7×7×32×32	0	2	1,868,038
Opt.	18×18×32	7×7×32×32	0	2	1,806,518 (-3.29%)
Seq./Hier.	18×18×32	7×7×32×32	0	1	7,290,502
Opt.	18×18×32	7×7×32×32	0	1	7,225,526 (-0.90%)

Table 5.2: Comparison between latency, as clock periods, between the sequential/hierarchical and the optimized implementations of the 2D-convolution accelerator, for different parameters and batch = 1.

The latency results prove that the different accelerator phases are executed in a transparent way. For instance, in the case of a 5×5×3 input tensor being convolved with a 3×3×3×3 weight tensor with no padding (lines 1 and 2), 156 values need to be fetched and 27 must be stored from/to the external memory. The sequential architecture needs 183 clock cycles to carry out these two phases. On the other hand, the optimized implementation

only needs an overhead of 25 clock periods after which the load and store phases are seamlessly performed. If we add the overhead difference (158) to the latency of the optimized architecture (760), we obtain the latency of the sequential architecture (918).

Thus, our goal of executing the different accelerator phases in parallel with a single batch, while also reducing the FPGA resource usage, has been achieved. However, latency reduction can be rather substantial (17.21 % for the aforementioned case) or it can be quite small (0.90 % with maximum input dimensions), depending on how similar the execution times for the different accelerator phases are. In fact, in the first example (-17.21%) the time of the compute phase is slightly bigger than the time of the load and store phases, while in the last example (-0.90%) the time of the compute phase is much larger. So, The execution of our 2D-convolution accelerator is now heavily *compute-bound*. Nonetheless, if we were to parallelize the execution of the datapath by unrolling completely (32 times) the *for-loop* on the output channels, for instance, we would obtain a much more significant latency reduction, as reported in the last column of Tab. 5.3. This optimization would come at the cost of some additional slice logic and of around 32 times the DSPs being used, which wouldn't be an issue since our accelerator is currently using just 2.13 % of the available DSP resources.

Arch.	**Features**	**Filters**	**Same**	**Stride**	**Latency** [***clks***]	
Seq./Hier.	5×5×3	3×3×3×3	0	1	428	-
Opt.	5×5×3	3×3×3×3	0	1	270	-36.9%
Seq./Hier.	5×5×3	3×3×3×3	1	1	981	-
Opt.	5×5×3	3×3×3×3	1	1	709	-27.7%
Seq./Hier.	18×18×32	7×7×32×32	0	2	118,145	-
Opt.	18×18×32	7×7×32×32	0	2	56,625	-52.1%
Seq./Hier.	18×18×32	7×7×32×32	0	1	290,945	-
Opt.	18×18×32	7×7×32×32	0	1	225,969	-22.3%

Table 5.3: Comparison between expected latency, as clock periods, between the unoptimized and optimized implementations of the 2D-convolution accelerator, for different parameters and batch = 1. The expected latency results have been computed from the values in Tab. 5.2 assuming a complete unrolling (32 times) of the *for-loop* on the output channels.

Thus, we encourage future hardware designers to further optimize the `compute` phase of our architecture, since the load and store phases have been made transparent to it.

Bibliography

[1] S. Lynch. *Andrew Ng: Why AI Is the New Electricity*. 2017. URL: https://www.gsb.stanford.edu/insights/andrew-ng-why-ai-new-electricity (cit. on p. 14).

[2] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. Aug. 2017. URL: https://arxiv.org/abs/1703.09039 (cit. on pp. 14, 21, 27, 28).

[3] *CS231n Convolutional Neural Networks for Visual Recognition*. URL: https://cs231n.github.io/neural-networks-1/ (cit. on p. 15).

[4] *IBM - Gradient Descent*. URL: https://www.ibm.com/cloud/learn/gradient-descent (cit. on p. 16).

[5] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on p. 17).

[6] *Convnet - notebook*. URL: https://people.minesparis.psl.eu/fabien.moutarde/ES_MachineLearning/TP_convNets/convnet-notebook.html (cit. on p. 18).

[7] C. Kevin. *Feature Maps*. URL: https://medium.com/@chriskevin_80184/feature-maps-ee8e11a71f9e (cit. on p. 19).

[8] A. Ng, Y.B. Mourri, and K. Katanforoosh. *Convolutional Neural Networks*. Coursera. URL: https://www.coursera.org/learn/convolutional-neural-networks (cit. on p. 19).

[9] K. Patel. *Convolutional Neural Networks — A Beginner's Guide*. URL: https://towardsdatascience.com/convolution-neural-networks-a-beginners-guide-implementing-a-mnist-hand-written-digit-8aa60330d022 (cit. on p. 20).

[10] A. Wang. *Convolutional Neural Networks(CNN) num. 1: Kernel, Stride, Padding*. URL: `https://www.brilliantcode.net/1584/convolutional-neural-networks-1-convolution-layer-stride-padding-kernel/` (cit. on p. 21).

[11] H. Gholamalinezhad and H. Khosravi. «Pooling methods in deep neural networks, a review». In: *arXiv preprint arXiv:2009.07485* (2020) (cit. on p. 22).

[12] H. Yingge, I. Ali, and K.-Y. Lee. «Deep Neural Networks on Chip - A Survey». In: Feb. 2020, pp. 589–592 (cit. on p. 22).

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. «Gradient-Based Learning Applied to Document Recognition». In: (Nov. 1998) (cit. on p. 23).

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. «ImageNet classification with deep convolutional neural networks». In: *NIPS'12: Proceedings of the 25th International Conference on Neural Information Processing System*. Dec. 2012, pp. 1097–1105 (cit. on pp. 23, 24).

[15] K. Simonyan and A. Zisserman. «Very deep convolutional networks for large-scale image recognition». In: *arXiv preprint arXiv:1409.1556* (2014) (cit. on p. 24).

[16] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. «Going deeper with convolutions». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9 (cit. on pp. 24, 25).

[17] K. He, X.Zhang, S.Ren, and J.Sun. «Deep residual learning for image recognition». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cit. on pp. 25, 26).

[18] Han Jia and Xuecheng Zou. «An FPGA-Based Resource-Saving Hardware Accelerator for Deep Neural Network». In: *International Journal of Intelligence Science* 11 (Jan. 2021), pp. 57–69 (cit. on p. 27).

[19] 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. 28, 29, 31).

[20] R. Zhao, W. Luk, X. Niu, H. Shi, and H. Wang. «Hardware Acceleration for Machine Learning». In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2017, pp. 645–650 (cit. on p. 30).

[21] Y. Sun, G. Wang, B. Yin, J. R. Cavallaro, and T. Ly. «Chapter 8 - High-level Design Tools for Complex DSP Applications». In: *DSP for Embedded and Real-Time Systems*. Ed. by R. Oshana. Oxford: Newnes, 2012, pp. 133–155 (cit. on p. 30).

[22] *Catapult® Synthesis User and Reference Manual*. Mentor, a Siemens Business. 2018 (cit. on pp. 32, 33).

[23] N. Haron and S. Hamdioui. «Why is CMOS scaling coming to an END?» In: Jan. 2009, pp. 98–103 (cit. on p. 34).

[24] *ESP - The open source SoC platform*. URL: `https://www.esp.cs.columbia.edu` (cit. on pp. 34, 36).

[25] 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)*. IEEE. 2020, pp. 1–9 (cit. on pp. 34, 35).

[26] 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. 38, 39).

[27] T. Jia et al. «A 12nm Agile-Designed SoC for Swarm-Based Perception with Heterogeneous IP Blocks, a Reconfigurable Memory Hierarchy, and an 800MHz Multi-Plane NoC». In: *Proceedings of the 48th European Solid-State Circuits Conference (ESSCIRC 2022)*. 2022 (cit. on pp. 40, 41).

[28] K. Bai. *A Comprehensive Introduction to Different Types of Convolutions in Deep Learning*. Towards Data Science. URL: `https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215` (cit. on p. 43).

[29] *How to: design an accelerator in C/C++ (Xilinx Vivado HLS)*. URL: `https://www.esp.cs.columbia.edu/docs/cpp_acc/cpp_acc-guide/` (cit. on p. 44).

[30] *ESP Softmax Accelerator*. URL: https://www.esp.cs.columbia.edu/prebuilt/mentor_cpp_acc/ (cit. on p. 45).

[31] *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 p. 45).

[32] *Algorithmic C (AC) Datatypes Reference Manual*. Siemens EDA. 2022 (cit. on p. 48).

[33] L. P. Carloni. «From Latency-Insensitive Design to Communication-Based System-Level Design». In: *Proceedings of the IEEE* 103.11 (2015), pp. 2133–2151 (cit. on p. 52).

[34] *ESP - The ESP Vision*. URL: https://www.esp.cs.columbia.edu (cit. on p. 70).