# POLITECNICO DI TORINO

## Master's Degree in Electronic Engineering

### Master's Degree Thesis

# Efficient Tiling Architecture for Scalable CNN Inference: Leveraging High-Level Design and Embedded Scalable Platform


**Supervisors**

**Prof. Mario Roberto CASU**

**Dr. Luca URBINATI**

**Candidate**

**Diego Ricardo BUENO PACHECO**


**Academic year 2022-2023**

# Summary

In recent years, Convolutional Neural Networks (CNNs) have gained significant prominence across a multitude of computer vision and deep learning applications, driving notable advancements in fields such as image classification, object detection, face recognition, and medical imaging. As the demand for deep learning and computer vision applications continues to rise, the pressing need for more efficient and scalable solutions to meet the computational demands of cutting-edge CNN models becomes increasingly evident. Furthermore, the necessity of extending these applications to a broader range of devices, without relying on cloud solutions, has led to a shift in computation from cloud servers to edge devices. However, this transition presents a formidable challenge due to the limited computational and memory resources inherent in edge devices. Addressing this challenge requires the utilization of hardware accelerators and the partitioning of tensors into manageable tiles that align with memory constraints. Within this context, this thesis presents an innovative tiling architecture tailored to facilitate large-scale Convolutional Neural Network (CNN) inference, with a particular emphasis on harnessing the potential of High-Level Design (HLD) and the Embedded Scalable Platform (ESP). HLD empowers the description of hardware using more abstract, higher-level functional representations and architectural constraints, subsequently translating into more concise and debug-friendly C++ code. ESP augments the integration design process by seamlessly incorporating the architecture into a System-on-Chip (SoC), primarily comprising at least one RISC-V processor, one or more external memory tiles, one or more accelerator tiles, and an I/O tile. ESP streamlines the design of accelerators using C/C++ in conjunction with diverse HLD tools, thereby facilitating straightforward integration and testing of bare-metal software applications. This thesis introduces a tiling algorithm that takes into account several critical factors, including the organization

and addressing of tensors within external memory, the maximum number of available processing elements in the accelerator design (e.g. MAC units), the required precision for Multiply-Accumulate (MAC) operations (e.g., 16, 8, or 4 bits), and the memory sizes of private local memories (PLMs) within the accelerator. Each of these considerations is meticulously integrated into the tiling algorithm to optimize performance and resource utilization. The architecture is rigorously tested through RTL simulation and FPGA deployment, establishing its feasibility and effectiveness in real-world applications. The experiments results show the tiling algorithm's effectiveness and in which tensor dimensions cases the tiling architecture performs the best.

# Acknowledgements

I would like to express my heartfelt gratitude to my family, friends, and professors for their unwavering support and guidance throughout the completion of my Master's thesis.

To my family, your belief in me and sacrifices have been invaluable, and I am deeply thankful for your unwavering support.

To my friends, thank you for your patience and understanding during late-night study sessions and moments of stress. Your friendship has been a source of strength and joy.

To my professors and advisors, your expertise and feedback have been instrumental in shaping the direction and quality of this thesis.

Lastly, to all those who provided words of encouragement and acts of kindness, your support has not gone unnoticed.

Completing this thesis has been a challenging yet rewarding journey, and I am grateful for the role each of you played in this achievement. Thank you for being part of this milestone in my academic career.

*"Success is not the key to happiness. Happiness is the key to success. If you love what you are doing, you will be successful."*
*Albert Schweitzer*

# Table of Contents


**List of Tables** VIII

**List of Figures** IX

**1 Convolutional Neural Networks** 1
- 1.1 Typical layers in CNNs . . . 2
- 1.2 2D Convolution . . . 4
- 1.3 Memory and computation requirements for CNN . . . 6
  - 1.3.1 Memory requirements . . . 7
  - 1.3.2 Computation Requirements . . . 7
  - 1.3.3 Edge devices constraints . . . 8
  - 1.3.4 Tiling and methods to meet memory constrains . . . 8

**2 Hardware CNN accelerators** 11
- 2.1 Tiling methods . . . 12
- 2.2 Dataflow Techniques . . . 19
- 2.3 Precision-Scalable Hardware Accelerators . . . 23

**3 Embedded Scalable Platforms and CNN accelerator design using High-Level Synthesis** 26
- 3.1 ESP and its design flows . . . 27
- 3.2 CNN accelerator design using High-Level Synthesis . . . 30

**4 Tiling Architecture for CNN** 35
- 4.1 Tiling algorithm . . . 35
- 4.2 Software Implementation . . . 40
  - 4.2.1 2D Convolution Tiling . . . 40
  - 4.2.2 Depthwise Convolution Tiling . . . 44
  - 4.2.3 Fully-connected Tiling . . . 45

4.2.4 2D Convolution Tiled Process and Loops Order . . . 47
4.2.5 Depthwise Convolution Tiled Process and Loops Order 55
4.2.6 Fully-connected Tiled Process and Loops Order . . . 57
4.3 Hardware Implementation . . . . . . . . . . . . . . . . . . . . 60
4.3.1 2D Convolution accelerator . . . . . . . . . . . . . . . . 60
4.3.2 Depthwise Convolution accelerator . . . . . . . . . . . 75
4.3.3 Fully-connected accelerator . . . . . . . . . . . . . . . . 76

**5 Simulation, FPGA implementation and Results** 81
5.1 C/C++ simulation . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.2 High Level Synthesis and RTL simulation . . . . . . . . . . . 83
5.3 FPGA Prototyping and validation results . . . . . . . . . . . 87
5.4 Performance results . . . . . . . . . . . . . . . . . . . . . . . . 89

**6 Conclusions** 96

**Bibliography** 98

# List of Tables

4.1 Variables definition for 2D-Conv tiled . . . . . . . . . . . . . . 51
4.2 Variables definition for DW-Conv tiled . . . . . . . . . . . . . 55
4.3 Variables definition for FC tiled . . . . . . . . . . . . . . . . . . 58
4.4 List of parameters to configure the 2D-Conv accelerator . . . 62
4.5 List of arguments of the function that implements the computation phase . . . . . . . . . . . . . . . . . . . . . . . . . 73

5.1 2D-Conv Performance results . . . . . . . . . . . . . . . . . . . 95

# List of Figures


1.1 Multi layer Convolutional Neural Network example [1] . . . 2
1.2 Typical layers in a CNN [2] . . . . . . . . . . . . . . . . . . . . 4
1.3 2D Convolution [3] . . . . . . . . . . . . . . . . . . . . . . . . 6
1.4 Depthwise Convolution. In this example, Filters and image have been broken into three different channels and then convolved separately and stacked thereafter [3] . . . . . . . . . . 9
1.5 Pointwise Convolution. In this example, three input channels are combined into one output channel [3] . . . . . . . . . . . 9

2.1 Block convolution example [8] . . . . . . . . . . . . . . . . . . 13
2.2 Fixed and Hierarchical blocking [8] . . . . . . . . . . . . . . . 13
2.3 DORY L3-L2-L1 layer routine example [10] . . . . . . . . . . 15
2.4 Tile and Pack algorithm of the layers on the 34 IMAs [11] . 17
2.5 The GrateTile data structure [12] . . . . . . . . . . . . . . . . 18
2.6 PE for the WS dataflow technique [14] . . . . . . . . . . . . . 20
2.7 Example of a PE set for the WS dataflow technique [14] . . 20
2.8 PE for the OS dataflow technique. [14] . . . . . . . . . . . . . 21
2.9 Example of a PE set for the OS dataflow technique. [14] . . 21
2.10 PE for the RS dataflow technique. [14] . . . . . . . . . . . . . 22
2.11 Example of a PE set for the RS dataflow technique. [14] . . 22
2.12 Example of a PE set for the NLR dataflow technique [14] . . 22
2.13 Generic ST multiplier [16]. . . . . . . . . . . . . . . . . . . . . 23
2.14 Configurations of a ST multiplier [16] . . . . . . . . . . . . . 23
2.15 Working principles of ST-based DNN accelerators [16] . . . . 25

3.1 The ESP architecture and its main types of tiles [18] . . . . 28
3.2 Agile SoC design and integration flows in ESP [17] . . . . . . 30
3.3 HLS-based accelerator design in ESP [17] . . . . . . . . . . . 31

3.4 Overview of the accelerator and SoC design flows with an example of SoC design configuration on the ESP GUI [17] . 32

4.1 Tiling of the weight tensor across output channel dimension 38
4.2 Tiling of the input/weight tensor across input channel dimension 39
4.3 Tiling of the input tensor across the height dimension . . . . 39
4.4 Tensor data organization in external memory . . . . . . . . . 48
4.5 Organization of the tensors in external memory . . . . . . . 48
4.6 Interfaces for a generic ESP accelerator . . . . . . . . . . . 61

5.1 C Testbench simulation result of a 2D Convolution operation 83
5.2 Loop list and configurations . . . . . . . . . . . . . . . . . . . 85
5.3 2D-Conv loop's schedule . . . . . . . . . . . . . . . . . . . . 85
5.4 Multipliers synthesised for FPGA technology . . . . . . . . . 86
5.5 Multipliers synthesised for ASIC technology . . . . . . . . . 86
5.6 RTL Simulation of Tiled convolution . . . . . . . . . . . . . 87
5.7 2D-Conv results in FPGA . . . . . . . . . . . . . . . . . . . . 90
5.8 RTL simulation of the 2D-Conv computation for the worst case scenario . . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.9 RTL simulation of the 2D-Conv computation for the best case scenario . . . . . . . . . . . . . . . . . . . . . . . . . . 92

# Chapter 1

# Convolutional Neural Networks

In recent years, Convolutional Neural Networks (CNNs) have emerged as a cornerstone of computer vision and deep learning, enabling remarkable progress in a wide range of applications. Their ability to automatically extract intricate features from visual data has revolutionized fields such as image classification, object detection, and medical imaging. However, the widespread adoption of CNNs on edge devices, where computational resources are limited, has been impeded by the substantial computational and memory demands of convolutional operations inside these networks.
Convolutional Neural Networks are a type of artificial neural networks uniquely designed to process and analyze visual data or data which is ordered in a grid-like topology, making them exceptionally suited for tasks involving images and videos. The architecture of CNNs is inspired by the visual processing hierarchy in the human brain, where layers of neurons progressively extract increasingly abstract features from raw sensory input. This hierarchical feature extraction enables CNNs to automatically learn relevant patterns and representations from data, significantly reducing the need for handcrafted features and manual classification.
The genesis of CNNs can be traced back to the pioneering work of Yann LeCun and his colleagues in the early 1990s, however it was not until the arrival of deep learning and the availability of large-scale labeled datasets that CNNs truly gain importance in academic and industrial sectors. The success of models like AlexNet, VGGNet, and ResNet in image classification competitions marked a turning point in the development of CNNs. These

models highlights the remarkable capacity of deep CNNs to outperform traditional computer vision techniques and even surpass human-level performance in image classification tasks.
As CNNs continued to evolve, researchers began to work in various architectural innovations and training techniques that contributed to improve their performance. The introduction of concepts like skip connections, batch normalization, and residual networks has permitted the development of deeper and more efficient CNN architectures. The versatility of Convolutional Neural Networks extends beyond traditional computer vision tasks. They have found applications in medical image analysis, in autonomous vehicles, or even in artistic and creative fields.

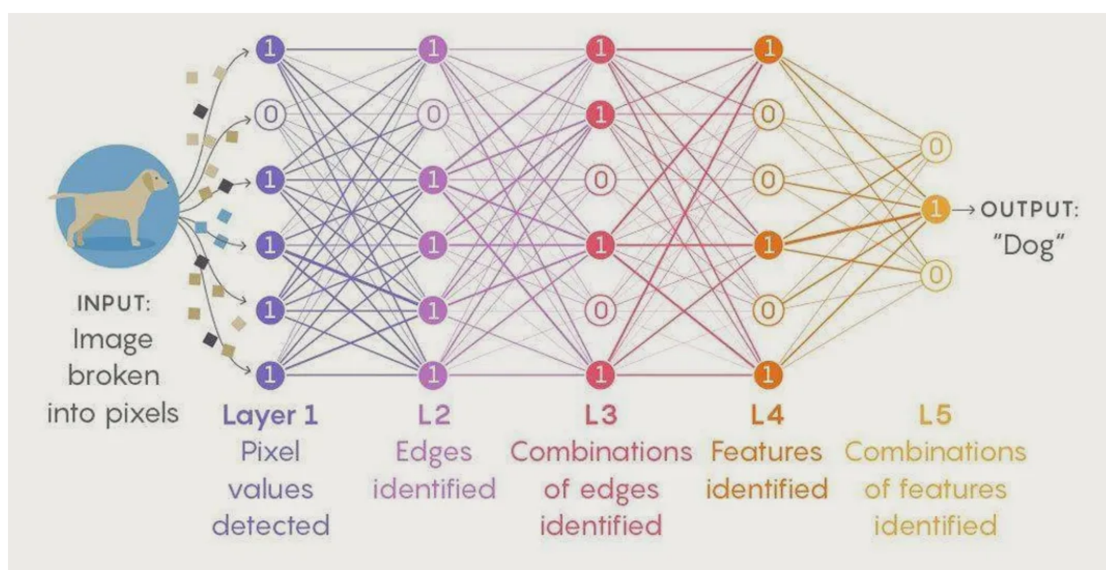


**Figure 1.1:** Multi layer Convolutional Neural Network example [1]

## 1.1 Typical layers in CNNs

A typical Convolutional Neural Network (CNN) consists of several layers that are designed to process and extract features from input data. The exact architecture of a CNN can vary depending on the specific target task and design choices, however the most common layers found in a typical CNN are:

- **Input Layer**
  The input layer receives the raw input data, which is usually an image

or a multi-dimensional tensor. The dimensions of this layer match the dimensions of the input data.

- **Convolutional Layers** Convolutional layers are the main blocks of CNNs. They are in charge of applying filters to the input data using the convolution operation. These filters permit feature extraction from the input data. A typically convolutional layer consists of multiple filters or kernels which are used to detect different features in the input data. The layer may include activation functions such as ReLU (Rectified Linear Unit) to introduce non-linearity right after the convolution operation.

- **Pooling Layers** The main purpose of Polling Layers is to reduce the spatial dimensions of the feature maps, output of convolutional layers, keeping the most important information of the input. This operation helps to reduce the computational workload and the number of parameters in the network without impacting the performance of the network.

- **Fully-connected layers** Fully-connected layers are traditional neural network layers in which each neuron is connected to every neuron of the previous layer. When used as last layer of a CNN, Fully-connected layer's output represents the network's prediction or classification.

- **Activation Functions** As mentioned before, activation functions introduce non-linearity into the network. Thanks to non-linearity the network can model complex relationships in the data. Most common activations functions are: ReLU, sigmoid, and tanh.

- **Normalization Layers** Normalization layers are used to stabilize and accelerate training. The scope of the layer is to normalize the activations of the neurons to have zero mean and unit variance.

- **Dropout Layer** This layer drops randomly a fraction of neurons during training, preventing overfitting in the inference process and promoting the generalization of the model.

- **Output Layer** The output layer produces the final predictions or classifications in the network. The number of neurons presented in this layer usually matches the number of classes in a classification task.

Figure 1.2 shows an example of the multiple layers found in a Convolutional Neural Network and how them are commonly organized to classify a car's image.

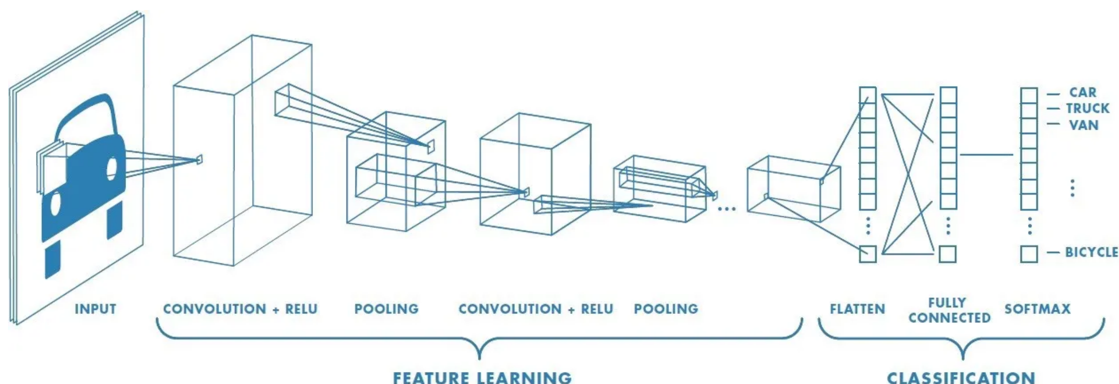


**Figure 1.2:** Typical layers in a CNN [2]

## 1.2 2D Convolution

2D Convolution or 2D-Conv operation is a fundamental building block in Convolutional Neural Networks used for feature extraction from two-dimensional data, most commonly images. It applies a convolutional filter to an input image to extract features. The 2D-Conv operation is performed multiple times in a Convolutional Neural Network using different filters to extract and reconize features from the input image. These features are then used in subsequent layers of the network for tasks like: image classification, object detection, and segmentation. 2D-Conv operation can be explained with the following concepts:

- **Input** The input of the 2D-Conv operation is a two-dimensional matrix, typically representing an image. In the case of color images, this matrix usually has one channel per color Red, Green, and Blue (RGB) or just one channel in the case of gray scale images. In the following layers across the network, the number of input channels may vary depending on the topology of the network.

- **Convolutional Filter - Kernel** The convolutional filter, typically a smaller two-dimensional matrix, is used to detect specific patterns

or features within the input image. These filters have values usually called weights which are set and learned by the network during training process.

- **Sliding the Filter** The filter is systematically moved across the input image. The movement is performed in both horizontal and vertical directions passing through all the values or pixels of the input data. The filter's movement is controlled by a parameter called stride, which defines the steps of the sliding . A stride equals to 1 means that the filter moves one position at a time, while larger strides skip some positions depending on stride's value.

- **Element-wise Multiplication** At each position where the filter overlaps with the input image, element-wise multiplication is performed between the filter and the corresponding pixel values in the input. The element-wise operations are also performed across all the input channels for the same positions in each channel.

- **Summation** After element-wise multiplication, the results of the multiplications are summed together to obtain a single scalar value for that position, the summation also considers the values obtained in all the input channels which leads to a sum of elements in three dimensions. This result is often referred to as the convolutional response or feature activation.

- **Output Feature Map** The scalar value obtained in the Summation step is placed in the corresponding position of the output feature map. After placing all the values of an output channel, the process should be repeated for the number of output channels that are present in the layer. The output feature map represents the result of applying the filters to the input image at that particular location. Each output channel represents the result of different set of filters applied to the same input data.

- **Padding** Padding might be added to the input image to adjust the size of the output feature map. Padding consists in adding extra rows and columns, typically filled with zeros, around the input image. This helps maintain the spatial dimensions, which are also affected by the stride value, and can be useful to avoid edge-related issues when applying the filters in the image's borders.

- **Strides** The stride as mentioned before, defines how much the filter is moved across input's positions. A larger stride results in a smaller output feature map, which can be beneficial for reducing computational requirements due to the omission of certain positions in the image.

Figure 1.3 depicts the 2D Convolution operation for an input tensor with three input channels and an output tensor with one output channel.

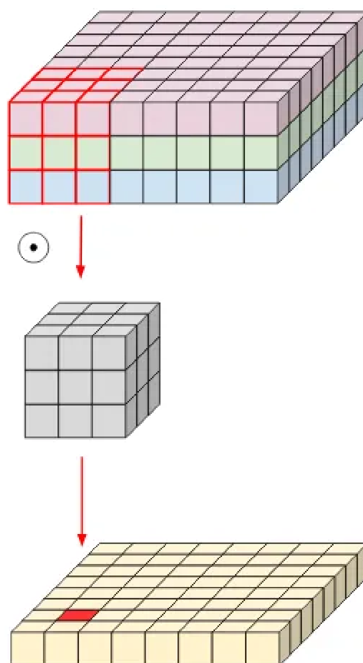

**Figure 1.3:** 2D Convolution [3]

## 1.3 Memory and computation requirements for CNN

Convolutional Neural Networks (CNNs) have proven to be highly effective for many tasks and applications in which they can perform better compared with a human, however they come with substantial memory and computation requirements which can suppose a limitation for the deployment. In real-world scenarios, the deployment of CNNs on edge devices, such as smartphones, IoT devices, or embedded systems, presents significant challenges due to the

limited resources that are present in these devices and also for the necessity of maintaining a low energy consumption in devices that are usually battery powered. Hardware accelerators and tiling methods are crucial strategies to address these requirements.

### 1.3.1 Memory requirements

- **Model Parameters:** Modern CNN architectures are deep -composed by many layers- and often have millions of trainable parameters which most of them are the filters applied to the input. Storing these parameters in memory suppose a challenge due to the dimensions of the data and the store capacity of the memories, especially on resource-constrained edge devices.
- **Activation Maps:** Covolutional Neural Networks generate intermediate feature maps at each layer during inference. These feature maps require memory for storage and further processing, and as the network deepens, more intermediate feature maps are needed increasing memory requirements.
- **Batch Processing:** Usually CNNs process input data in batches to improve computational efficiency. Each batch requires additional memory for storing intermediate results .

### 1.3.2 Computation Requirements

- **Convolution Operations:** In CNNs the most computational demanding task is the convolutional operation due to the large number of multiply and accumulate operations performed in the inference part. The real-time behaviour of the applications makes also a computational demanding task when trying to achieve a specific latency that meets the application requirements.
- **Fully-Connected Layers:** These layers also demands a great number of matrix multiplications increasing the computational load of the network.
- **Activation Functions:** Applying activation functions (e.g., ReLU, sigmoid) introduces non-linearity which leads to some advantages as explained before, however it also introduces additional computation workload.

- **Pooling Layers:** Pooling layers reduce spatial dimensions, but they still require computational resources to perform operations like max-pooling or average-pooling which also introduces workload overhead to the network.

### 1.3.3 Edge devices constraints

Deploying CNNs on edge devices introduces several constraints and challenges:

- **Limited Processing Power:** Edge devices, such as smartphones and IoT devices, have limited computational capabilities compared to high-end servers or desktops. Running complex CNNs in real-time applications needs further optimization of hardware and software.

- **Energy Efficiency:** Edge devices are often battery-powered, and energy efficiency is a critical concern. Power-hungry CNN computations can rapidly drain the device's battery. To avoid this scenario the CNN computation can be modified and special hardware such as custom accelerators can be used.

- **Memory Constraints:** As the need to use less area is in Edge devices increases, the amount of memory available is limited and the storing of large models and intermediate feature maps become more complex. This issue underscores the need to perform iterative computations with small data chunks to meet memory constraints.

### 1.3.4 Tiling and methods to meet memory constrains

Many techniques have been proposed to alleviate the CNN's memory requirements and to minimize partial results in intermediate layers which often are transferred off-chip memory and then back again to on-chip memory causing an increase in latency and energy consumption. Tiling methods are a set of techniques used to split the computation of CNN operations into smaller set of chunks or tiles. This helps to overcome limitations in memory and computational resources as discussed in section 1.3.3. The main approaches to mitigate the effect of large CNN models are:

1. **Spatial Tiling:** It divides the input data into smaller spatial regions or tiles for processing. In this case each tile is computed separately reducing

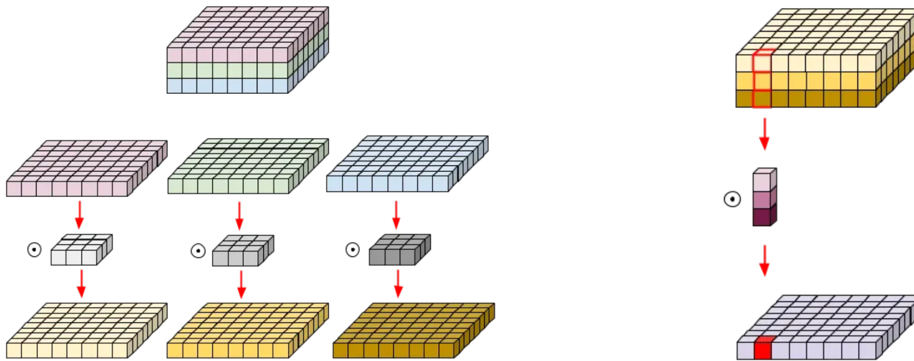

**Figure 1.4:** Depthwise Convolution. In this example, Filters and image have been broken into three different channels and then convolved separately and stacked thereafter [3]

**Figure 1.5:** Pointwise Convolution. In this example, three input channels are combined into one output channel [3]

the memory requisite and allowing a parallelization in the processing. Then the results from each tile are combined obtaining the final result. Spatial tiling reduces the memory footprint because just a portion of the input data is loaded at a given time. However, managing the tile's boundaries is a challenge due to the data dependency between adjacent tiles and also combining tile's results can introduce synchronization overheads.

2. **Channel Tiling:** This approach divides the channels of the input data into smaller groups searching a reduction of memory requirements as well. In this case, the data subset is processed at a time and parallelization can be leveraged too. Nevertheless, the way the channels are divided has to be carefully handled to ensure coherence in the results and avoid to impact the model's performance.

3. **Depthwise Separable Convolutions:** besides tiling or dividing data into smaller parts, other effective way to reduce memory and computation requisites is the use of a specialized form of convolution which divides the operation into depthwise and pointwise convolutions. Depthwise convolution applies a separate convolutional filter to each input channel,

generating a set of feature maps per each channel. Then Pointwise convolution combines these feature maps using a 1x1 kernel to produce the final output. This process significantly reduces the number of parameters and computations compared to traditional convolutions maintaining a similar accuracy in the inference results. Figure 1.4 and 1.5 show a depthwise convolution followed by a pointwise convolution.

These methods help strike a balance between computational efficiency and memory constraints, enabling real-time and energy-efficient processing of deep neural networks. However, selecting the appropriate tiling method and optimizing its implementation often involves trade-offs between memory usage, computational speed, and model performance, and it may require careful experimentation and fine-tuning for specific use cases. In this work, both tiling approaches, spatial and channel, will be used with specific constrains which are hardware and layer topology dependents. This will be discussed in detail in Ch. 4.

# Chapter 2

# Hardware CNN accelerators

Many methods and strategies has been developed to address the memory and computation constraints for deploying state-of-the-art CNN models. These methods usually target a specific set of devices depending on the application of the network. Therefore, there are tiling methods and dataflows aiming at high-performance computing systems, DNN accelerators for desktop computers, or small microcontrollers for embedded systems; each of them with different resource capabilities and different memory hierarchies depending on the system needs. Having a memory hierarchy implies data movements between memory levels. The challenge arises from the high volume of weight and activation data that needs to be moved during the execution of a DNN. This movement of data has several implications:

- **Data Transfer Bottleneck:** moving data between levels of the memory hierarchy can be a time-consuming and resource-intensive task. Due to the different access times and bandwidths present in each memory level, memory operations often results in significant latency and resource usage.

- **Memory Bandwidth Constraints:** The share of data between memory levels is subject to bandwidth limitations depending on the memory level. In DNN, especially in deep CNN, the volume of data being moved can saturate available memory bandwidth, causing performance bottlenecks.

- **Energy and Power Consumption:** each data movement between memory levels consumes energy and power; therefore as the rate of data transfers increase, the energy consumption increases as well making a critical concern for battery-powered devices, such as mobile phones and IoT devices.

Memory hierarchy management is particularly critical in CNN deployment, especially on resource-constrained devices like edge devices. The objective is to optimize the memory usage and data transfer strategies to minimize the impact on the DNN's execution time and energy consumption. To address this challenge, various techniques and hardware accelerators have been developed [4], including the use of specialized hardware like GPUs [5], TPUs [6], and optimized software libraries [7]. These solutions target to reduce data movement, exploit parallelism, and enhance the efficiency of memory hierarchy management for CNNs. Tiling methods also deal with this challenge and some solutions are able to mitigate the problem up to a point in which data transfer's overhead is almost negligible for the system performance [8],[9], [10].

## 2.1 Tiling methods

The tiling problem has been addressed from diverse approaches considering the different variables that concern a CNN application deployment. Among the state-of-the-art methods, the following ones have been considered in this thesis:

- Block convolution [8] proposes a method which consists in dividing the feature maps into independent tiles thanks to the application of a block padding to each tile. Figure 2.1 depicts the padding process applied to each block of the tensor. Therefore, the convolution can be performed separately on individual blocks and layers, eliminating the dependency of feature map in adjacent tiles. The results of each block are then spliced together to obtain the final output. Block convolution is a hardware-friendly and efficient approach which can eliminates the off-chip transfer of intermediate results during inference. The paper also propose two strategies to manage the block's sizes across the layers:
  - **Fixed blocking:** the block size remains consistent across layers. After pooling, adjacent output blocks with reduced resolution are

combined into a larger block for further processing. The number of blocks in a layer decreases as the network deepens, and the receptive field of output blocks increases.

– **Hierarchical blocking:** the number of blocks remains constant in each layer. As the network deepens, the size of individual blocks gradually becomes smaller. Unlike fixed blocking, the block's receptive field in each layer stays unchanged. The entire network is divided into independent sub-networks along the spatial dimension.

Fixed blocking allows information fusion between independent blocks, expanding the receptive field and maintaining accuracy. Hierarchical blocking lacks this information fusion, leading to larger accuracy degradation. Figure 2.2 represents graphically the difference between fixed and hierarchical blocking.

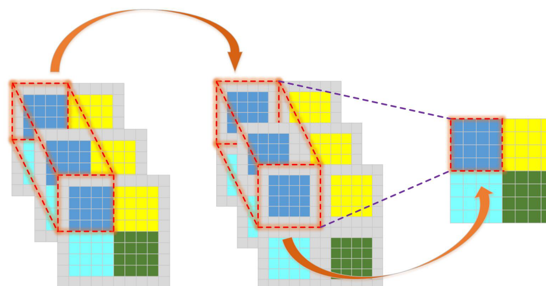

**Figure 2.1:** Block convolution example [8]

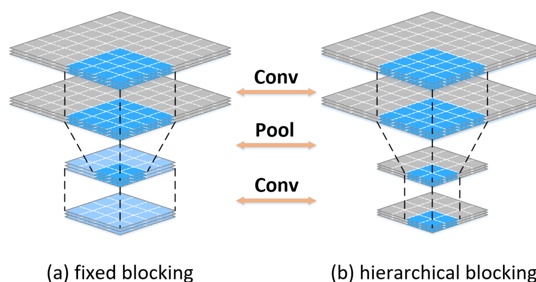


**Figure 2.2:** Fixed and Hierarchical blocking [8]

- DORY [10] proposes a tiling approach based on Constraint Programming (CP). It targets devices with three hierarchical levels (L1, L2, L3) and it

maximizes L1 memory utilization under topological constraints imposed by each DNN layer. The approach generates ANSI C code to orchestrate off- and on-chip transfers and computation phases. Furthermore, to maximize speed, the CP formulation is augmented with heuristics promoting performance-effective tile sizes. The operation of DORY is organized in three steps, performed offline before network deployment:

- **ONNX decoder:** The purpose of the ONNX (Open Neural Network Exchange) Decoder is to decode an input ONNX graph, which represents an already quantized DNN. The decoded ONNX graph is then reorganized into a set of layers. Each layer includes a Linear-add-pooling operation, an optional Batch-Normalization operation, and a Quantization/Activation operation. The ONNX Decoder helps to prepare the input ONNX graph for deployment on low-cost MCUs with limited on-chip memory.
- **Layer analyzer:** a component that is used in the first optimization phase, it considers each layer of the Deep Neural Network (DNN) separately from each other and uses weight dimension information from the previous layer to optimize the current layer. The Layer Analyzer consists of three sub modules that work together to optimize the layer:
  * **Tiling Solver:** it relies on a 2-step engine, which solves the L3-L2 tiling constrained problem first, and the L2-L1 one afterwards. L3-L2 tiling enables storing activations and weights in the L3 off-chip memory instead of the on-chip L2, which supports significantly larger layers. The Solver uses a five-stage cascaded procedure to search for an L3 tiling solution, where each stage tries to tile a different selection of buffers to fit the constraint. In the case of L2-L1 tiling, DORY abstracts tiling as a CP problem and uses the CP solver from the open-source OR-Tools developed by Google AI to meet hardware and geometrical constraints while maximizing an objective function. The objective function of the solver is to maximize L1 memory utilization by manipulating tile dimensions.
  * **Target-specific Heuristics and constraints:** The objective function of the tiling solver can be augmented with a series of heuristics targeting a specific backend to maximize performance.

  * **Software-cache generator:** is a tool used by DORY to automatically generate C code for executing a whole layer of a DNN. The tool uses the tiling solution found by the Tiling Solver to instantiate asynchronous data transfers and calls to the back-end kernels without any manual effort. All data transfers are pipelined and asynchronous, and with this approach, the memory transfer overhead can be almost completely hidden. The code generator is not platform-agnostic, but the approach can be easily generalized to any computing node with a three-level memory hierarchy.

- **Network parser:** a component of the DORY tool that is responsible for building a network graph after the Layer Analyzer has completed layer-wise tiling. The network graph considers each layer as a callable function, and DORY uses the information extracted from all the layers to create this graph.

Figure 2.3 depicts the Dory's dataflow through the memory hierarchy level.

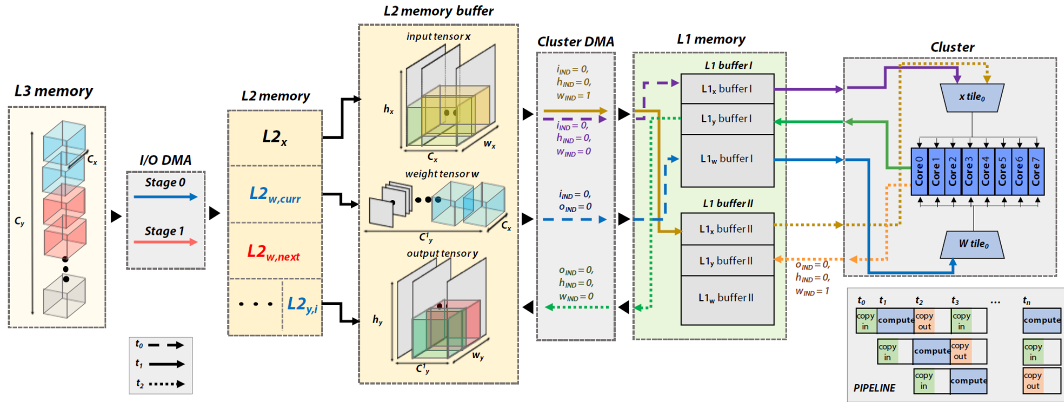


**Figure 2.3:** DORY L3-L2-L1 layer routine example [10]

- TTILE [9] is a tool to optimize the tiling process for convolution operations. It involves composing micro-kernels to perform a 2D convolution efficiently. The authors use a technique called "loop fusion" to combine multiple loops into a single loop, which can improve performance by reducing the overhead associated with multiple loops. The microkernel is created by fully unrolling innermost parallel loops and may also include

perfectly nested reduction loops. The purpose of the microkernel is to improve performance by reducing the overhead of loop control and allowing for more effective code optimization. The authors demonstrate that combining micro-kernels is more effective than relying on partial tiles, which can be suboptimal.

- Tile and Pack [11] algorithm splits a layer over multiple IMAs (in-memory computer accelerator) only when it does not fit the size of the cross-bar (memory architecture). The algorithm does not allow tiling to fill unfilled IMA locations, aiming at the highest utilization area of the cross-bar on a per-tile basis. Packing is based on the Maximal Rectangles Best Short Side Fit fitting algorithm. The Tile and Pack algorithm improves the system performance by tiling all layers and packing their contributions in the smallest number of IMAs. Here is a step-by-step explanation of the algorithm:

  1. **Input Parameters:** several input parameters are taken, including the names, heights, and widths of all layers in the neural network, the size of each IMA (default is 256), and the number of available IMAs.
  2. **Creating Tiles:** tiles for each layer of the neural network are created. A tile represents a portion of a layer that can be processed by an IMA. The size of each tile is determined by the size of the IMA.
  3. **Calculating the Number of Tiles:** the number of tiles needed to cover the entire layer is calculated. This is done by dividing the height and width of the layer by the size of the IMA. The remainder of the division is stored for later use.
  4. **Creating Tiles for Full Rows:** tiles for each full row of the layer are created. The algorithm iterates over the number of rows and columns and assigns a tile size of (S, S) to each tile.
  5. **Creating Tiles for Partial Rows:** tiles for the remaining partial rows of the layer are created. The algorithm iterates over the number of rows and assigns a tile size of (hrem, S) to each tile.
  6. **Creating Tiles for Partial Columns:** tiles for the remaining partial columns of the layer are created. The algorithm iterates over the number of columns and assigns a tile size of (S, wrem) to each tile.

7. **Creating Tile for Remaining Area:** a tile for the remaining area that is not covered by the full rows or partial rows/columns. It assigns a tile size of (hrem, wrem) to this tile.
8. **Removing 0-sized Tiles:** The algorithm removes any tiles that have a height or width of 0, as they do not contain any meaningful data.
9. **Bin Packing:** The algorithm uses a bin packing algorithm called BINBESTFIT to organize the tiles into bins. This step ensures efficient utilization of the IMAs.
10. **IMA Mapping:** The algorithm uses a rectangle packing algorithm called MAXRECTSBSSF to map the bins onto the available IMAs. This step determines the optimal placement of the tiles on the IMAs.
11. **Output:** The algorithm returns the IMA Mapping, which represents the mapping of the tiles onto the IMAs.

Figure 2.4 shows the application of the tiling algorithm to the weights of MobileNetV2 network.

**Figure 2.4:** Tile and Pack algorithm of the layers on the 34 IMAs [11]

- GrateTile [12] is an uneven tile division for efficient storage and processing of sparse CNN feature maps. The algorithm divides the data

into uneven-sized subtensors and stores them in a compressed format with small indexing overhead and keeping the random access capability. The goal is to avoid accessing partially compressed subtensors and minimize the number of subtensors to reduce data fragmentation. This design enables modern CNN accelerators to fetch and decompress subtensors on-the-fly in a tiled processing manner and to reduce the DRAM bandwidth utilization in the process inference. Figure 2.5 depicts the difference between an uniform tile division and the GrateTile division applied to compressed tensors.

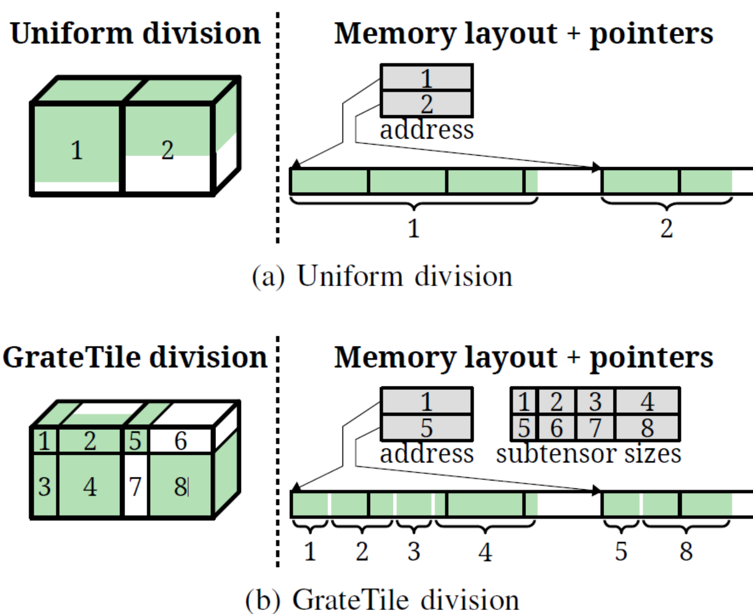


**Figure 2.5:** The GrateTile data structure [12]

- Timeloop [13] splits the input data into smaller tiles that can fit into the accelerator's memory. The size of the tiles is determined taking into account the size of the memory and the size of the input data. The tiles are processed sequentially, and the results are combined to produce the final output. Tile analysis involves the following steps:
  - **Varying the tile size:** The size of the tiles used in loop tiling is varied to analyze the performance of the accelerator for different tile sizes.

  - **Generating a performance model:** For each tile size, a performance model is generated that estimates the performance of the accelerator for a given DNN workload.
  - **Analyzing the performance model:** The performance model is analyzed to determine the optimal tile size that maximizes the performance of the accelerator.

## 2.2 Dataflow Techniques

Dataflow techniques for CNN are strategies and architectures designed to optimize the flow of data -input, output, weights- through the hardware in charge of managing and compute the convolution operations, the main goal is to improve efficiency of computations and minimize latency in the process. These techniques are crucial for achieving high performance in CNN accelerators. The state-of-the-art methodologies and literature have deeply analyzed the different dataflow techniques used in CNN accelerators. [14] focuses on efficient design space exploration (DSE) of dataflow techniques such as weight-stationary (WS), output-stationary (OS), row-stationary (RS), and no local reuse (NLR) techniques. It analyzes the processing element (PE) structure and computational pattern of each technique, then it calculates performance metrics like throughput, computation-to-communication ratio, on-chip memory usage, and off-chip memory bandwidth. The paper proposes the roofline model, an analysis model used to explore the performance of a system in terms of hardware and software design. It provides a visual representation of the performance limits of a system based on the ratio of computational resources (CR) and performance.
The four representative dataflow techniques are:

- **Weight-Stationary (WS):** This dataflow technique stores the weights in stationary memory locations, increasing the reuse of weight values during MAC operations. Instead, input activations are streamed continuously through the computation units. The main advantages are the reduction of weights movement and the reuse of them in architectures where the weights are shared across multiple input activations. If the reduction of on-chip buffer memory is the goal, weight stationary dataflow offers an appropriate design point [14]. Figure 2.6 shows PE structure for the WS dataflow technique and Figure 2.7 shows an example of a

PE set for 5x5 input feature maps, 3x3 weight, and a 3x3 output feature map.

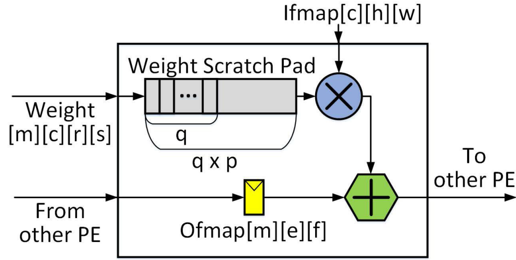


**Figure 2.6:** PE for the WS dataflow technique [14]

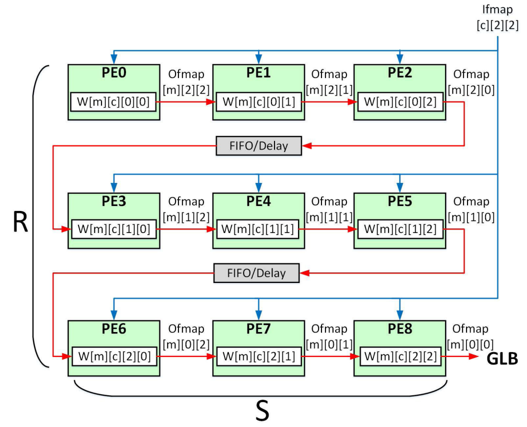


**Figure 2.7:** Example of a PE set for the WS dataflow technique [14]

- **Output-Stationary (OS):** The output stationary dataflow keeps output features stationary in memory. Input and weight activations are streamed continuously through the computation units. This technique optimizes throughput and on-chip memory usage, specially for layers with small output feature map sizes. The main advantages are the reduction of output feature maps movement and the performance efficiency in scenarios when the output maps are reused or further processed. The implementation of an accelerator, in which the design goal is to minimize the area and hardware resources, has an optimal design point applying an output stationary dataflow [14]. Figure 2.8 shows PE architecture for the OS dataflow technique and Figure 2.9 shows an example of a PE set for 5x5 input feature maps, 3x3 weight, and a 3x3 output feature map.

- **Row-Stationary (RS):** The RS dataflow technique arranges data in a specific manner allowing that the entire rows of input activations or feature maps can be processed in parallel. The technique also minimizes energy consumption by leveraging local data reuse on a spatial architecture, specially in scenarios where the accelerator can exploit parallelism across rows in an efficient way. These scenarios are present in large CNN architectures and when dealing with large convolution operations. [15]

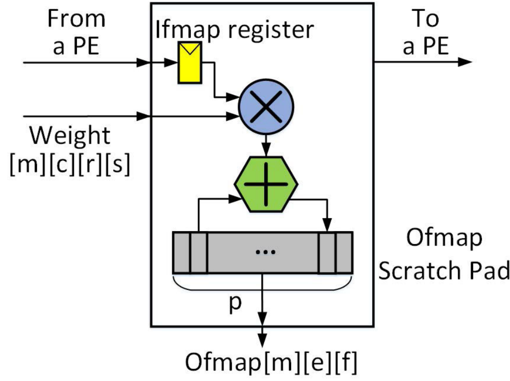


**Figure 2.8:** PE for the OS dataflow technique. [14]

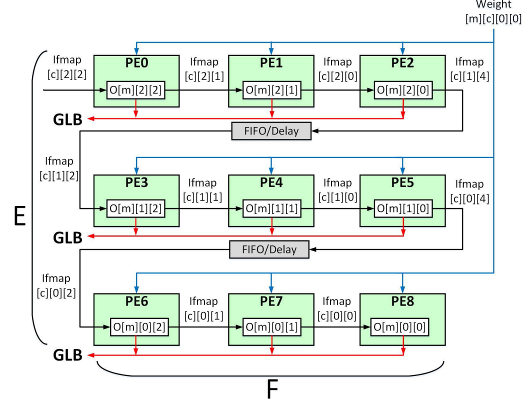


**Figure 2.9:** Example of a PE set for the OS dataflow technique. [14]

uses RS dataflow technique in its CNN accelerator's architecture. It breaks the high dimensional convolution down to 1D convolution primitives allowing a parallel processing. Then each primitive is mapped to one PE and it operates on one row of filter weights and one row of input feature map, and generates one row of partial sums. The computation of each row pair stays stationary in the PE, which creates convolutional reuse of filter weights and ifmap pixels at the register file level. Figure 2.10 shows PE architecture for the RS dataflow technique and Figure 2.11 shows an example of a PE set for 3x3 input feature maps, 2x2 weight, and a 2x2 output feature map.

- **No local reuse (NLR):** In NLR dataflow, the accelerator uses global buffers (GLBs) to store input, weight and output feature maps. Weight data is unicast to specific processing element and Input data is multicast to each procesing element that compute a specific output feature. The NLR dataflow technique tends to have higher throughput for convolutional layers with large input and output feature map sizes. If the main goal of the system is to achive the maximum performance, the NLR dataflow technique has an optimal point [14]. Figure 2.12 shows an example of a PE set for the NLR datafow technique.

These dataflow techniques are usually used or adapted depending on the specific architecture and requirements of the CNN accelerator. The goal

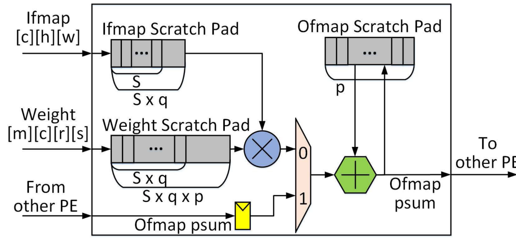


**Figure 2.10:** PE for the RS dataflow technique. [14]

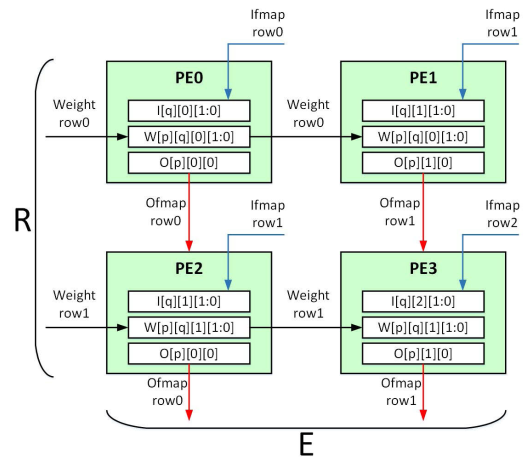


**Figure 2.11:** Example of a PE set for the RS dataflow technique. [14]

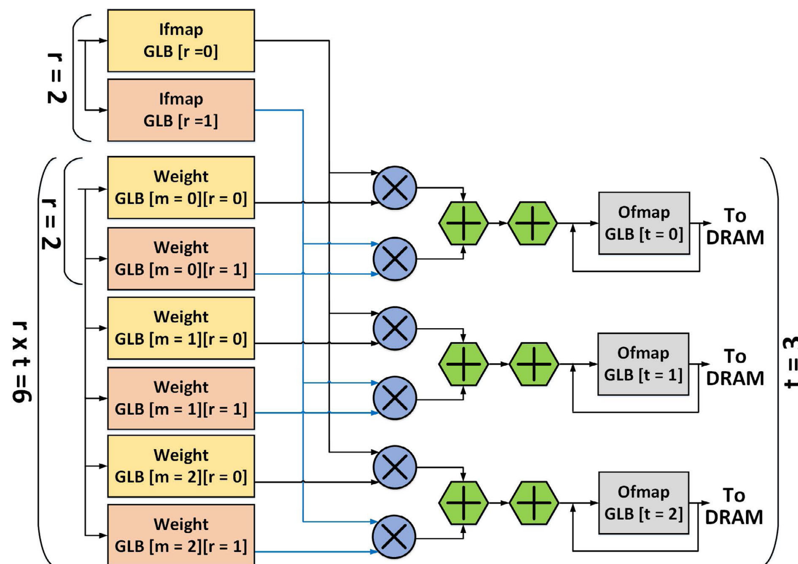


**Figure 2.12:** Example of a PE set for the NLR dataflow technique [14]

is always to efficiently utilize available resources, reduce data movement, and maximize parallelism, finally improving the performance and energy efficiency of the accelerator during CNN inference.

## 2.3 Precision-Scalable Hardware Accelerators

An effective way to find a trade-off between performance and accuracy when working with DNN layers is the use of Mixed-Precision Quantization, Precision-Scalable Multiply-and-accumulate units, and specifically Mixed-precision accelerators as the work presented in [16]. The work just mentioned designs an accelerator based on Sum-Together (ST) multipliers, whose inputs usually pack N = 1, 2, 4 operands depending on the configuration, and obtaining a precision inversely proportional to N. The main advantage of this approach is the computation in one shot of N multiplications in parallel, together with the addition of the low-precision products. Then the reconfiguration of the multiplier permits to perform a full-precision multiplication or a dot-product at a lower precision increasing the overall speed of a layer computation by a factor of N. Figure 2.13 depicts the basic architecture of a generic ST multiplier, and Figure 2.14 shows the different configurations of the operands that are supported by the multiplier. [16] leverages the use of the ST multipliers to design three kind of accelerators targeting: 2D-Conv, DW-Conv, and FC layers. The operational concept of the three ST-based accelerators is depicted in Figure 2.15. Depending on the configuration, the initial column illustrates the capability of the ST multiplier to handle the number of activation/weight pairs (N) within the 16-bit input operands. The subsequent three columns delineate the processing methodology for the input (depicted in blue) and weight (depicted in orange) tensors in 2D-Conv, DW-Conv, and FC operations, respectively.

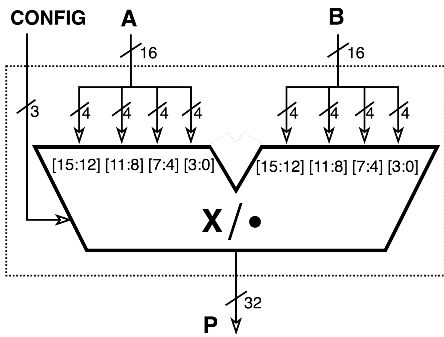


**Figure 2.13:** Generic ST multiplier [16].

| CONFIG | P |
|---|---|
| 16x16 (000b) | A[15:0] x B[15:0] |
| 16x8 (100b) | A[15:0] x B[7:0] |
| 8x8 (010b) | A[15:8] x B[7:0] + A[7:0] x B[15:8] |
| 8x4 (011b) | A[15:8] x B[3:0] + A[7:0] x B[11:8] |
| 4x4 (001b) | A[15:12] x B[3:0] + A[11:8] x B[7:4] + + A[7:4] x B[11:8] + A[3:0] x B[15:12] |

**Figure 2.14:** Configurations of a ST multiplier [16]

The three accelerators have different ways to fetch data for the multipliers due to the actual topology of the layer, these methods are explained as follows:

- Within the 2D-Conv operation, each orange filter featuring C kernels undergoes a process where the C input tensor channels are multiplied by the corresponding weight kernels. The resultant partial outcomes are subsequently summed channel-wise. At full precision (N = 1), the ST multiplier handles one weight and one "pixel" of the input channel at a time. However, when operating at reduced precision, the ST multiplier can receive two (N = 2) or four (N = 4) pixel/weight pairs from the channels dimension. These configurations are highlighted in red in the second and third rows of Figure 2.15. Thanks to the internal dot product performed by the ST multiplier at low precision, the number of external channel-wise additions reduces from N-1 to N/2 -1 or N/4 -1.

- A distinct strategy is essential for DW-Conv. In the context of depth-wise convolution, each output channel is derived by convolving every input channel with its corresponding weight kernel. As there is no accumulation process across the channel dimension, the utilization of the ST multiplier, as in 2D-Conv, is not feasible. Illustrated in the third column, depending on the configuration, the ST multiplier is supplied with 1, 2, or 4 pairs originating from the receptive field of the input tensor and the corresponding weight kernel. Hence the results of the low-precision multiplications can be summed together by the multiplier itself.

- The operational mechanism of the ST-based FC accelerator executes the matrix-vector product by computing between the weight matrix and the linear array of input activations. The ST multiplier operates on the activations array and a weights row to generate an output activation derived from the summation of low-precision products. Similar to the 2D-Conv scenario, the number of Multiply-Accumulate (MAC) cycles scales in proportion to C/N, while the corresponding latency scales inversely as 1/N.

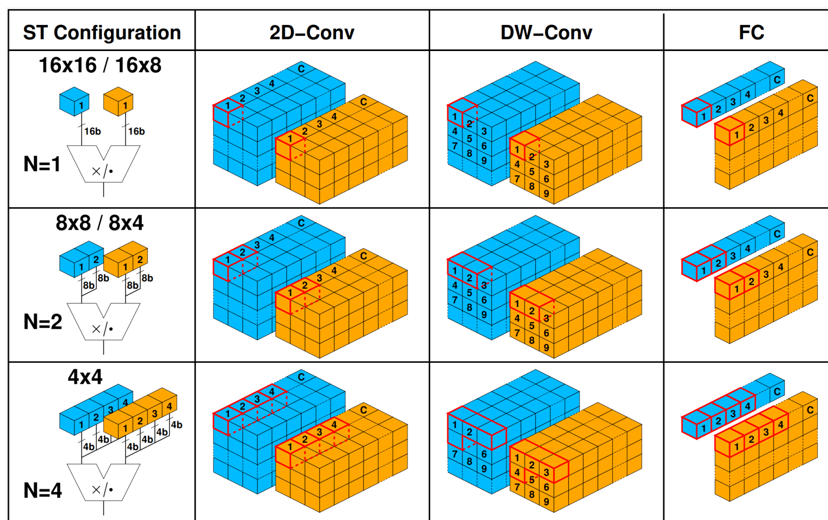


**Figure 2.15:** Working principles of ST-based DNN accelerators [16]

# Chapter 3

# Embedded Scalable Platforms and CNN accelerator design using High-Level Synthesis

The continuous growth in chip design complexity, driven by the slowdown of Moore's Law and the end of Dennard's scaling, has led to the adoption of heterogeneous System-on-Chip (SoC) architectures. These architectures, which enable multi-core processors and specialized hardware accelerators on a single die, have become the preferred solution for achieving both performance and energy efficiency across diverse application domains. However, the escalating costs of developing cutting-edge SoCs demands new methodologies and platforms supporting design reuse to significantly reduce design time and design expenses. In this context, the role of open-source hardware (OSH) is crucial, as it can uniquely contribute to design reuse by boosting entrepreneurial innovation and collaborative engineering across industry and academia. The success of the RISC-V open standard Instruction Set Architecture (ISA) has triggered the emergence of numerous SoC architectures. As the options of open-source hardware components expands, there is a growing need for Computer-Aided Design (CAD) methodologies within the open-source community. These methodologies should push the transformation of these components into diverse SoC designs tailored for specific domain applications. The predominant emphasis in Open-Source Hardware's context has been the

developing of processor cores that adhere to the RISC-V ISA. There also has been a notable concentration on small-scale System-on-Chips (SoCs) where these cores are tightly-coupled with functional units and co-processors, commonly facilitated by bus-based interconnects. In contrast, there has been a relatively limited emphasis on creating solutions for large scale SoCs that use RISC-V cores with many loosely-coupled components, such as coarse-grained accelerators, interconnected through a Network-on-Chip (NoC). The open-source platform, Embedded Scalable Platforms (ESP) was released taking into account all the previously issues presented in the development of SoCs targeting the relative new Artificial Intelligence demands. The following explanation about ESP is taken from references [17], [18].

## 3.1 ESP and its design flows

As stated before, ESP is a developing platform for heterogeneous SoC design and programming. It integrates a scalable architecture with a flexible system-level design methodology. Illustrated in Figure 3.1, the ESP architecture is organized as a heterogeneous tile grid constructed on a 2D mesh, multi-plane NoC. Each type of tile fulfills a distinct role within the SoC, yet all tiles are encapsulated in modular sockets that can be decoupled from the NoC design. In this architecture, processor and accelerators have the same importance in the SoC implementation. This approach proposes a system-centric view instead of the traditional processor-centric view. The main types of tiles present in ESP architecture are:

- **Processor Tile:** Each processor tile incorporates a processor core selected during the design phase from available options. The current options includes the RISCV 64-bit Ariane core from ETH Zurich, the 32-bit RISC-V Ibex core, and the SPARC 32-bit LEON3 core from Cobham Gaisler. All cores support Linux and are equipped with their dedicated L1 caches. The integration of processors into the distributed ESP system is seamless, requiring no ESP-specific software patches for Linux booting. Each processor communicates on a local bus independently of the rest of the system. The processor socket includes a unified private L2 cache with a configurable size, implementing a directory-based MESI cache-coherence protocol and allowing the core to transparently communicate in the ESP coherence protocol. Processor requests directed to memory-mapped I/O registers are forwarded by the socket to the IO/IRQ NoC

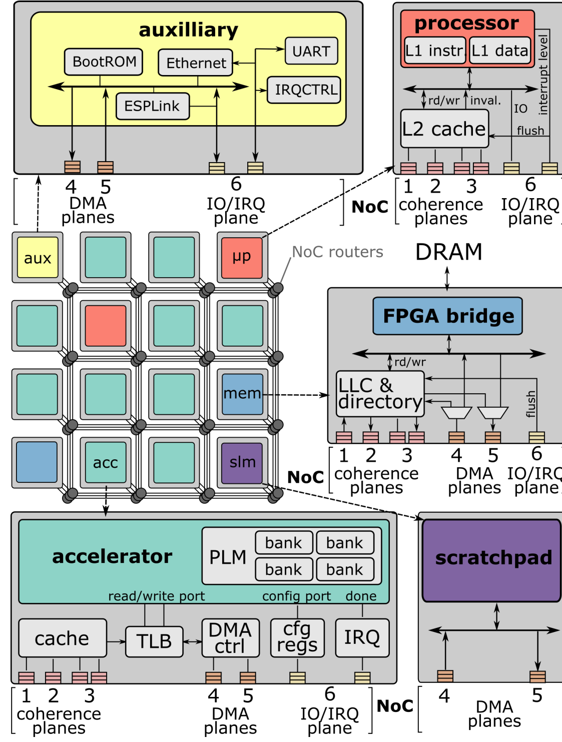


**Figure 3.1:** The ESP architecture and its main types of tiles [18]

plane through an APB adapter.

- **Memory Tile:** it establishes a connection to the external memory in the system. ESP allows for the straightforward instantiation of multiple memory tiles to meet the bandwidth requirements of large SoC designs. In such instances, each memory tile serves to a distinct partition of the global address space, and the mechanism to direct requests to the appropriate tile is generated automatically as well as the logic to support the partitioning, which is totally transparent to the software. When the ESP cache hierarchy is activated, the memory tile incorporates the ESP last-level cache (LLC). In conjunction with the L2 cache, ESP LLC implements a standard directory-based MESI coherence protocol specifically adapted to function over a NoC. Moreover, the ESP LLC is designed to handle Direct Memory Access (DMA) requests directly from accelerators in an LLC-coherent manner.

- **Accelerator Tile:** Within the ESP framework, the Accelerator Tile assumes a fundamental role as ESP embraces heterogeneity in SoC

design. These accelerators, characterized as loosely-coupled, execute coarse-grained tasks upon invocation by a processor core through a designated device driver. It also exchanges large datasets with the memory hierarchy without processor's intervention. When interacting with the memory hierarchy, an accelerator employs various coherence modes. These modes span from completely bypassing the cache hierarchy with Direct Memory Access (DMA) to participating in the system's coherence protocol when equipped with a private L2 cache. The socket of an accelerator tile provides platform services for tasks such as address translation, DMA, configuration registers, and coherence. This design approach enables designers to concentrate on optimizing their accelerators without the need to reimplement fundamental capabilities. Accelerators must adhere to a simple interface, providing load/store ports for latency-insensitive channels, signaling mechanisms for configuring and initiating the accelerator, and a signal to indicate the completion of accelerator operations and trigger an interrupt for the processors.

- **Auxiliary Tile:** The auxiliary tile accommodates all shared peripherals in the system, excluding memory. These peripherals include the Ethernet Network Interface Card (NIC), UART, a digital video interface, a debug link for controlling ESP prototypes on FPGA, and a monitor module responsible for gathering various performance counters and periodically transmitting them through the Ethernet interface. The socket of the auxiliary tile is the most complex because it must provide most platform services to the devices hosted by this tile. The Ethernet connection supports remote connection through SSH and enables the ESPLink debug application.

ESP offers multiple design flows for developing new hardware accelerators. These can be implemented at the Register-Transfer Level (RTL), or at higher abstraction level with High-Level Synthesis (HLS) tools, or directly from high-level machine learning models utilizing the open-source HLS4ML tool. Additionally, ESP provides a streamlined process for integrating pre-designed third-party accelerators, such as the NVIDIA NVDLA, as long as they adhere to a standard interface like AXI. This flexibility ensures compatibility and ease of integration for a diverse range of accelerators within the ESP architecture. Figure 3.2 depicts the different design and integration flows used in ESP. Depending on the abstraction level, the describing languages (C/C++ , VHDL, Verilog, etc.) and tools can vary as follows:

- Cycle-accurate RTL descriptions with languages like VHDL, Verilog, SystemVerilog, or Chisel.

- Loosely-timed or un-timed behavioral descriptions with SystemC or C/C++ that get synthesized into RTL with high-level synthesis (HLS) tools. ESP supports the three main commercial HLS tools: Cadence Stratus HLS, Mentor Catapult, and Xilinx Vivado HLS.

- Domain-specific libraries for deep learning like Keras TensorFlow, PyTorch, and ONNX, for which ESP offers a flow combining HLS tools with hls4ml.

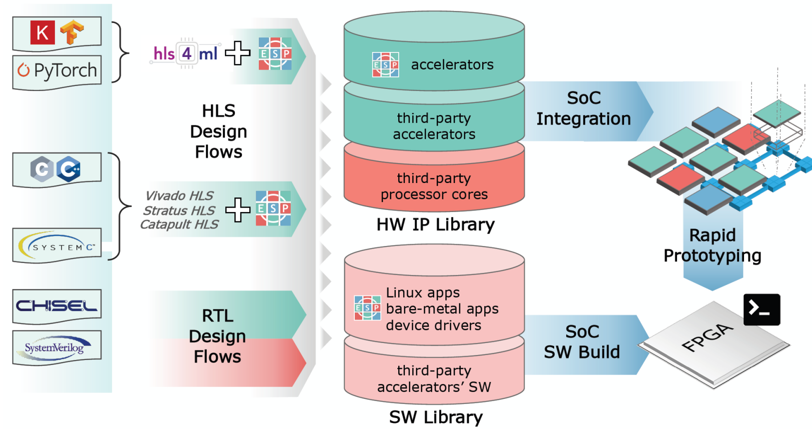


**Figure 3.2:** Agile SoC design and integration flows in ESP [17]

## 3.2 CNN accelerator design using High-Level Synthesis

In HLS-based workflows, ESP simplifies the task for accelerator designers by offering ESP-compatible accelerator templates, skeleton specifications ready for High-Level Synthesis (HLS), numerous examples, and step-by-step tutorials for each workflow. HLS-based workflow using C/C++ has the following advantages:

- It can be found a large codebase of algorithms already written in C/C++.

- Hardware/Software co-design is simplified due to the use of the same language.
- The simulation of the code is so much faster than RTL simulation, making the debugging process faster as well.

However, this workflow also shows some limitations when there is the need to infer concurrency, timing, and communication properties of the hardware systems.
An ESP accelerator design should have a well-organized description that divides the specification into concurrent functional blocks. The objective is to obtain a synthesizable specification that facilitates the exploration of an extensive design space, allowing for the evaluation of numerous micro-architectural and optimization choices. In Figure 3.3, the interconnection between the C/C++ design space and the Register-Transfer Level (RTL) design space is illustrated. HLS tools offer an extensive array of configuration options, referred to as knobs, enabling the generation of diverse RTL implementations. Each of these implementations corresponds to a distinct tradeoff point between cost and performance. The green arrows represent push-button directives within the HLS tool that control these knobs. Additionally, designers have the flexibility to manually transform the specification, as indicated by the orange arrows. This manual transformation allows for exploration of the design space while preserving the functional behavior.

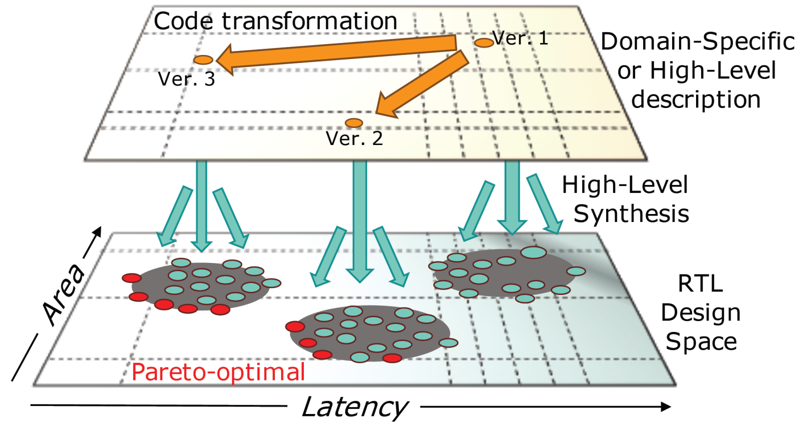


**Figure 3.3:** HLS-based accelerator design in ESP [17]

The HLS design flow using C/C++ involves the following steps:

- **Design Specification:** The functionality and performance requirements of the hardware accelerator are defined using a high-level programming language.
- **C Synthesis:** The high-level code is analyzed and synthesized into an intermediate representation. It captures the behavior of the hardware accelerator taking into account some timing constraints.
- **Optimization:** The intermediate representation is optimized to improve performance, area, and power consumption of the hardware accelerator.
- **RTL Generation:** The optimized intermediate representation is converted into RTL code, which describes the hardware components and their interconnections.
- **Verification:** The RTL code is verified to ensure that it correctly implements the desired functionality and to ensure that it meets the timing constraints.
- **Synthesis and Implementation:** The RTL code is synthesized and implemented into a target technology, such as an FPGA or an ASIC.
- **Testing and Debugging:** The hardware accelerator is tested and debugged to ensure its correct operation.

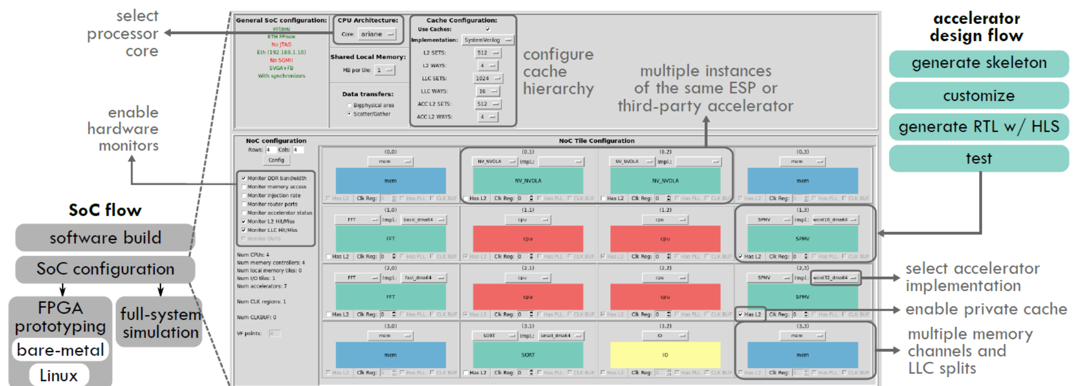


**Figure 3.4:** Overview of the accelerator and SoC design flows with an example of SoC design configuration on the ESP GUI [17]

HLS flow permits designers to rapidly prototype and explore different hardware architectures. Therefore, it significantly reduces the time and effort

required for hardware design. This design flow is part of a larger process intended to integrate the accelerator design into the SoC. The main steps in the ESP accelerator design flow are:

- **Generate Skeleton:** this step involves creating a basic framework or structure for the accelerator design, the overall architecture and organization of the accelerator is defined here. It includes determining the number and type of functional units, the interconnections between them, and the control logic. Then the generated skeleton serves as a starting point for further design and optimization. It also provides a foundation for adding and integrating specific functionality and features into the accelerator design.

- **Customize:** in this step the generated skeleton is tailored depending on specific requirements and performance optimization. During this step, application developers can customize the accelerator design by adding or modifying functional units, adjusting interconnections, and refining the control logic. The customization process aims to optimize the accelerator design for the targeted application or workload, ensuring efficient execution and improved performance. Thanks to the customization the design can be adapted to software requirements too, permitting a further optimization and support for the task performed by the system.

- **Generate RTL with HLS:** as explained before, this step is part of the HLS design flow and consists in generating RTL code using HLS techniques.

- **Testing:** verification and validation of the accelerator's functionality and performance are performed in this step. During this step, various tests and simulations are conducted to ensure that the accelerator operates correctly and meets the desired specifications. Testbenches are created to provide input stimuli and evaluate the output responses of the accelerator design. Functional verification tests check if the accelerator performs the intended operations accurately. Performance testing evaluates the accelerator's speed, latency, throughput, and power consumption. The test step is critical to ensure the reliability and correctness of the accelerator design before proceeding to the next stages of the design flow.

Once the accelerator is tested and verified, it has to be integrated into the SoC, verified together with the other components that interact through the Network on Chip, and tested with the application software. Figure 3.4 shows the different design flows within an example of the use of ESP's configuration GUI. The steps presented in the SoC flow are:

- **Software Build:** the software components that will run on the SoC are compiled and built. It includes tasks such as compiling source code, linking libraries, and generating executable files. The software build process ensures that the software components are ready to be deployed on the SoC.

- **SoC Configuration:** Configuring a SoC encompasses arranging the diverse components and subsystems depending on the intended specifications. This process involves establishing interconnections among different modules, configuring memory interfaces, and setting communication protocols. SoC configuration plays a crucial role in ensuring the seamless integration of hardware components, guaranteeing their proper functionality and readiness for operation.

- **FPGA Prototyping:** it plays an important role in SoC design, performing the implementation of the SoC design on a FPGA device. This phase enables early validation and testing of the SoC design before the actual fabrication process. Through FPGA prototyping, potential design issues and performance bottlenecks can be identified and addressed in time.

- **Full System Simulation:** it covers both hardware and software elements within a virtual environment. This simulation enables thorough testing and evaluation of the SoC's functionality, performance, and interactions among its diverse components. By conducting a full system simulation, potential issues or conflicts can be detected and resolved before the physical implementation of the SoC.

# Chapter 4

# Tiling Architecture for CNN

In this chapter the tiling algorithm used in the proposed work will be explained and analyzed. The software implementation will be also addressed, making special emphasis in the steps needed to obtain the tensor tiles, and finally provide the tile's dimensions and computation order to the accelerator. Then this last one is in charge of loading the tiled data, perform the corresponding operations, and store the results in memory again.

## 4.1 Tiling algorithm

The algorithm proposed in this work takes into consideration several aspects of the accelerator architecture and the topological characteristics, dimensions of the tensors, found in typical CNN layers. In this thesis, the accelerators mentioned in Ch. 2.3 are considered. These considerations are used as input to the tiling algorithm to constraint the tile sizes and ordering. They can be stated as follows:

- The main scope of the tiling algorithm is to fit the input, weight and output data into the accelerator's PLM. To accomplish this requirement, it is needed to know the various sizes of these PLMs, so the algorithm can recursively check if the tile fits these sizes.
- The for loops of the accelerators should have fixed upper bounds. Since each loop works on one dimension of the tensor, this means that each

tensor dimension has a maximum value supported by the accelerators. These maximum values have to be considered when deciding the dimension of a tile.

- The accelerators have several multiply-and-accumulate (MAC) units within the computation unit. Each PE works with a different and independently set of data. In the case of a 2D-convolution (2D-Conv) each PE works on a different output channel, in detphwise convolution (DW-COnv) each PE works on a different input channel, and in fully-connected (FC) operation each PE works on a different output neuron.
- The tiling architecture considers the implementation of a precision-scalable (PS) accelerator based on ST multipliers. Therefore, as explained in Ch. 2, multiple input features and weights values are fetched for the multiplier when using PS accelerators in low-precision configuration. These values are taken from different input channels (up to 4 channels for the lowest precision) in the case of 2D-Conv or different input values (up to 4 for the lowest precision) in the case of DW-COnv or FC algorithms. To leverage the advantages of the PS accelerator, the tiling algorithm should take into account the different values mentioned in the previous bullet points and split the input or weight tensors considering the precision configuration as the limit values. With this consideration, it is always possible to take advantage of the low-precision configuration.
- When the input tensor is tiled in the height or width dimensions, the sliding process used in convolution makes that part of the data has to be replicated in adjacent tiles to guarantee that the convolution operation is perform correctly. Consequently, in this work the tiling of the input tensor in width dimension and the tiling of the weight tensor in the width and height dimension are avoided.

After taking these requirements in consideration, a set of inequalities is derived for each accelerator taking inspiration from Dory [10]. In particular, three memory size conditions have to be met for each PLM used. For the more complex case, 2D-Conv, the inequalities are:

$$PLM_{IN} \geq H_{IN} * W_{IN} * C_{IN} \tag{4.1}$$

$$PLM_{W} \geq H_{kernel} * W_{kernel} * C_{IN} * C_{OUT} \tag{4.2}$$

$$PLM_{OUT} \geq H_{OUT} * W_{OUT} * C_{OUT} \tag{4.3}$$

Inequality 4.1 evaluates if the input PLM size is greater or equal to the multiplication of the input tensor dimensions: height, width and channel dimension. Inequality 4.2 evaluates if the weight PLM size is greater or equal to the multiplication of the weight tensor dimensions: kernel height, kernel width, channel input, and channel output dimension. Inequality 4.3 evaluates if the output PLM size is greater or equal to the multiplication of the output tensor dimensions: height, width and channel dimension. These inequalities have to be checked each time a tile is evaluated. Then if all the conditions are satisfied, the tiling algorithm is able to find a feasible tile which can be executed by the accelerator, otherwise the memory requirements need to be less restrictive.
After this general introduction, the tiling algorithm is now explained in detail for each kind of accelerator considered in this work:

- **2D Convolution steps:**

  1. The tiling starts by dividing the output channel dimension of the weight and output tensors. To obtain the maximum performance of the accelerator, the number of output channels taken by the tile is equal to the number of PEs present in the accelerator, in case there are more PEs than output channels no tiling operation is done. In this way, the accelerator can leverage all its internal resources and can process data in parallel. Figure 4.1 shows a weight tensor with two output channels represented by tow cubes.
  2. The next step is to tile across the input channel dimension of the input and weight tensors. In this step, the tensors are recursively divided by two until the inequalities 4.1 and 4.2 are met or until a minimum channel input value is reached. Depending on the precision configuration of the accelerator, the minimum values change correspondingly to the number of input channels processed in parallel. For instance, when the precision configuration is set to the lowest precision (i.e. 4-bit for activations and weights) the accelerator takes up to 4 different values from different input channels; in this case the minimum channel input value is also set to 4. In the case of medium precision (i.e. 16-bit for activations and weights) the value is 2 and for full precision (i.e. 16-bit for activations and weights) is 1

because the accelerator only takes one value from the input channels at a time. Figure 4.2 shows an input or weight tensor which has been tiled across the input channel dimension dividing the tensor by two.

3. If the tiled tensor still does not fit into the memories, the current step is executed. Here the input and output tensor are tiled across the height dimension. A fixed-height length is taken to simplify the problem of data replication when dealing with adjacent tiles. The length is equal to the height length of the kernel, this choice simplifies the need to calculate a tile dimension compatible with the size of the kernel and the stride used in convolution operation. It also avoids to take a part of the tensor that will not be used due to a mismatch between the tile size and the kernel/stride sizes. Then, the tiling across the width dimension is avoided to prevent data replication for adjacent tiles (which can have some part of the input data in one and the rest of the data in the next tile). Figure 4.3 shows an input or output tensor that has been tiled across height dimension selecting the height size equal to the kernel height size.

4. In case the memory requirements are not satisfied up to this point, this last step is performed. Here the tiling across the output channel dimension is performed again. In this case, performance is resigned and functionality is prioritized. The tile is divided by two, the conditions are checked, and then the cycle is repeated up to the point where just one output channel is obtained.

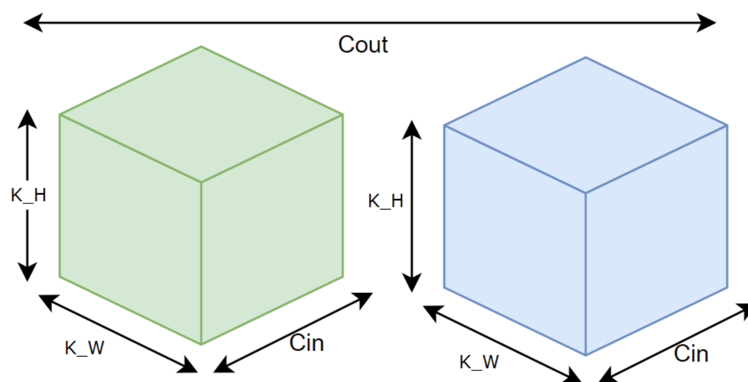


**Figure 4.1:** Tiling of the weight tensor across output channel dimension

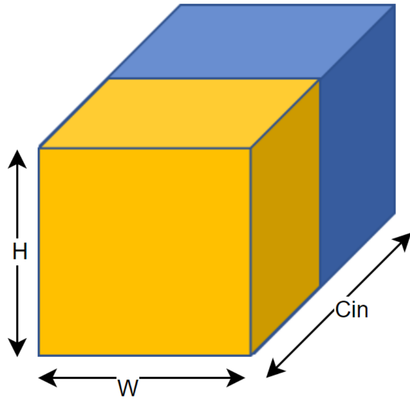


**Figure 4.2:** Tiling of the input/weight tensor across input channel dimension

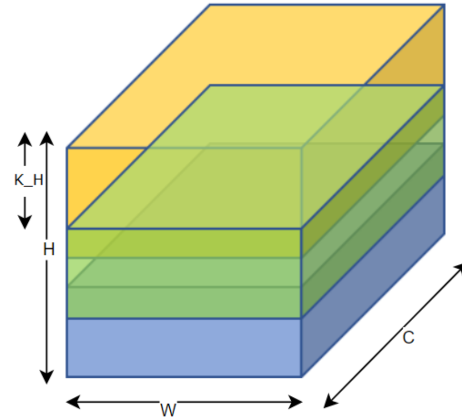


**Figure 4.3:** Tiling of the input tensor across the height dimension

- **Depthwise convolution steps:**

  1. In the case of DW-Conv the process starts with the tiling across the channels. As the case of 2D-Conv, here each PE works in a single channel independently of the other channels. Therefore, the number of channels taken by the tile is equal to the number of PEs present in the accelerator. If the number of channels is less than the number of PEs, no tiling is performed. In this case the three tensors are tiled, input, weight and output tensors.
  2. After tiling across channel dimension, the tiling across height dimension is performed in the same way as the 2D-Conv, obtaining a height size equal to the kernel height size. Here just the input and output tensors are tiled.
  3. Finally, if the tensor does not fit into the PLM requirements; performance is rescinded to achieve a feasible tile. A recursively division by 2 is done in channel dimension until the constraints are met.

- **Fully-connected steps:**

  1. In the FC case there are just two dimensions to tile. First, the tiling is performed in the output activation dimension. In this dimension

the values are independent, therefore each PE can work with one of them. As the previous cases, the number of output activations taken by the tile is equal to the number of PEs. If the number of output activations is less than the number of PEs, no tiling is performed. In this case both weight and output tensors are tiled.

2. The next step is to tile across the input activations dimension both input and weight tensors. In this case, the tensors are recursively divided by two until the memory constraints are met or until a minimum value is achieved. The minimum value depend on the precision configuration that will be used in the computation. For instance, when the precision configuration is set to the lowest precision, the accelerator can take up to 4 different values from different input activations; in this case the minimum value is also set to 4. In the case of medium precision the value is 2 and for full precision is 1 because the accelerator only takes one value from the input activations at a time.

3. In case the memory requirements are not satisfied up to this point, this last step is performed. Here the tiling across the output activations dimension is performed again. In this case, performance is gave up and functionality is prioritized. The tile is recursively divided by two, the conditions are checked each time, and if they are not satisfied the cycle repeats up to the point where just one output channel is obtained.

## 4.2 Software Implementation

The tiling algorithm was implemented in C language and can be executed offline because the size of the accelerator's PLMs are known in advance, i.e. the tile sizes can be calculated before the inference process saving computation time for the processor, following the implementation of each tiling algorithm explained before.

### 4.2.1 2D Convolution Tiling

The method that computes the tiling has the following prototype:

**Listing 4.1:** Tiling method prototype

```
int get_tiling(int Hin, int Win, int Cin, int Cout, int ker, int* Cout_t, int*
     Cin_t, int* Hin_t);
```

It receives the dimensions of the tensors: Hin, Win, Cin, Cout, ker; and then it returns the tiled dimensions with the pointers: `Cout_t, Cin_t, and Hin_t`. Then the memory requirements are calculated with the help of the *mem_update* function. The available and required memories are printed as well.

**Listing 4.2:** Initialization step

```
void mem_update(int Hin, int Win, int Cin, int Cout, int ker, int* mem_in, int
    * mem_w, int* mem_out) {
    *mem_in = Hin * Win * Cin;
    *mem_w = ker * ker * Cin * Cout;
    *mem_out = Hin * Win * Cout;
}
int get_tiling(int Hin, int Win, int Cin, int Cout, int ker, int* Cout_t, int*
     Cin_t, int* Hin_t) {
    int mem_in, mem_w, mem_out;
    mem_update(Hin, Win, Cin, Cout, ker, &mem_in, &mem_w, &mem_out);
    printf("Available local input memory is:\t%d\tMemory needed is:\t%d\n",
    plm_in, mem_in);
    printf("Available local weight memory is:\t%d\tMemory needed is:\t%d\n",
    plm_w, mem_w);
    printf("Available local output memory is:\t%d\tMemory needed is:\t%d\n",
    plm_out, mem_out);
    ...
```

After checking the preliminary conditions, the first step of the tiling is performed. The code checks the memory conditions for the weights and outputs, the maximum number of output channel supported by the accelerator (*FILT_MAX*) is also checked. If the number of output channels are greater than the PEs, the tile is performed and the memory parameters are updated. Otherwise, no tiling is performed.

**Listing 4.3:** First step in the algorithm for conv2d

```
    *Cout_t = Cout;
    if (plm_w < mem_w || plm_out < mem_out || *Cout_t > FILT_MAX) {
        // Tile across Cout dimension with tiles multiple of number of PEs
    (16)
        if (Cout >= PE) {
            *Cout_t = PE;
            mem_update(Hin, Win, Cin, *Cout_t, ker, &mem_in, &mem_w, &mem_out)
    ;
            printf("Available local weight memory is:\t%d\tWeight memory
    needed after tiling Cout is:\t%d\n", plm_w, mem_w);
            printf("Available local output memory is:\t%d\tOutput memory
    needed after tiling Cout is:\t%d\n", plm_out, mem_out);
        } else {
```

```
10              printf("No tiling in Cout was performed\n");
11          }
12      }
```

The next step of the tiling is done in a similar way, in this case the memory requirements are checked recursively in a while loop and if the conditions are satisfied or the minimum values are reached ($N_C_MAX$ is the maximum value of the input channel dimension supported by the accelerator), the loop breaks thanks to the if statements. The code also approximates the tile dimension to a multiple of 2 or 4 in case lower precision configuration are used. Thanks to this approximation the accelerator can leverage the parallel computation for the lower precision configurations. Then the memory conditions are updated and the information of the tiling is printed for debugging purposes.

**Listing 4.4:** Second step in the tiling algorithm for 2D-Conv

```
1       // Tile across Cin dimension dividing by 2 until it fits into memory
        otherwise go to next step
2      *Cin_t = Cin;
3       int i = 1;
4       while ((plm_in < mem_in || plm_w < mem_w || plm_out < mem_out || *Cin_t >
       N_C_MAX)) {
5           if (precision_opt == 0){
6               if (*Cin_t == 1) break;
7           } else if (precision_opt == 1) {
8               if (*Cin_t == 4) break;
9           } else{ // precision_opt == 2 || precision_opt == 3
10              if (*Cin_t == 2) break;
11          }
12          *Cin_t = (*Cin_t / 2) ;
13          // aproximate the tile dimesion to a multiple of 2 or 4 depending on
        precision_opt
14          if (precision_opt == 1){
15              if (*Cin_t % 4 != 0)
16                  *Cin_t += *Cin_t % 4;
17          } else if (precision_opt == 2 || precision_opt == 3){
18              if (*Cin_t % 2 != 0)
19                  *Cin_t += 1;
20          }
21          mem_update(Hin, Win, *Cin_t, *Cout_t, ker, &mem_in, &mem_w, &mem_out);
22          printf("Available input memory is:\t%d\tInput memory needed after
        tiling Cin=%d %d time(s) is:\t%d\n", plm_in, *Cin_t, i, mem_in);
23          printf("Available weight memory is:\t%d\tWeight memory needed after
        tiling Cin=%d %d time(s) is:\t%d\n", plm_w, *Cin_t, i, mem_w);
24          i++;
25      }
```

Afterwards, if memory constraints are still unsatisfied, tiling across height dimension is done. This part of the code checks the memory conditions and also the maximum height dimension supported by the accelerator

($N_H_MAX$). Then assigns the size tile equal to the kernel height size and finally update the memory conditions.

**Listing 4.5:** Third step in the tiling algorithm for 2D-Conv

```
1      // Tile across Hin dimension by stripes with height equal to kernel's
       height
2      *Hin_t = Hin;
3      if (plm_in < mem_in || plm_out < mem_out || *Hin_t > N_H_IN_MAX) {
4          *Hin_t = ker;
5          mem_update(*Hin_t, Win, *Cin_t, *Cout_t, ker, &mem_in, &mem_w, &
       mem_out);
6          printf("Available input memory is:\t%d\tInput memory needed after
       tiling Hin is:\t%d\n", plm_in, mem_in);
7          printf("Available output memory is:\t%d\tOutput memory needed after
       tiling Hin is:\t%d\n", plm_out, mem_out);
8      }
```

Finally, the last part of the algorithm is applied. Memory constraints are checked again and the recursively division by two is done taking into account an approximation to the upper integer in case the division gives an fractional result. This prevents missing data when there is an uneven division. If the conditions are not met, a warning message is printed. Otherwise tiles dimensions are printed, then the total number of tiles are also calculated. This last step will vary depending on the which dimensions were tiled. The *get_h_iterations* function returns the number of iterations that are performed in the convolution when the kernel is slid through the input in the height dimension.

**Listing 4.6:** Fourth step in the tiling algorithm for 2D-Conv

```
1  ...
2      // Tile across Cout again giving up performance
3      while (*Cout_t > 1 && (plm_w < mem_w || plm_out < mem_out)) {
4          //*Cout_t = (int)ceil(*Cout_t / 2.0);
5      *Cout_t = (*Cout_t % 2 == 0)? (*Cout_t / 2) : (*Cout_t / 2 + 1);
6          mem_update(*Hin_t, Win, *Cin_t, *Cout_t, ker, &mem_in, &mem_w, &
       mem_out);
7          printf("Available local weight memory is:\t%d\tWeight memory needed
       after tiling Cout is:\t%d\n", plm_w, mem_w);
8          printf("Available local output memory is:\t%d\tOutput memory needed
       after tiling Cout is:\t%d\n", plm_out, mem_out);
9      }
10     if (plm_in < mem_in || plm_w < mem_w || plm_out < mem_out || Win >
       N_W_IN_MAX || *Hin_t > N_H_IN_MAX ||  *Cin_t > N_C_MAX || *Cout_t >
       FILT_MAX ) {
11         printf("Layer impossible to tile, larger memory needed\n");
12         return -1;
13     } else {
14         printf("Tile dimensions are - Hin x Win x Cin x Cout: %d %d %d %d\n",
       *Hin_t, Win, *Cin_t, *Cout_t);
```

```
15        int cin_it = (Cin % (*Cin_t)==0)? (Cin/ (*Cin_t)) : (Cin/ (*Cin_t) +
    1);
16        int cout_it = (Cout % (*Cout_t)==0)? (Cout/ (*Cout_t)) : (Cout/ (*
    Cout_t) + 1);
17        if (*Hin_t<Hin)
18            printf("Total number of tiles(aprox): %d\n",(get_h_iterations() *
     cin_it * cout_it));
19        else
20            printf("Total number of tiles(aprox): %d\n",(cin_it * cout_it));
21        return 0;
22    }
23 }
```

## 4.2.2 Depthwise Convolution Tiling

The code that implements the tiling for DW-Conv is very similar to the code used in the case of 2D-Conv. The main difference is the absence of output channels in the DW-Conv case, therefore there is just one channel dimension which is tiled according to the number of PEs present in the accelerator. Another major difference is that in this case the precision configuration is not considered due to the absence of tiling in the width dimension, the one from which are taken more input values in case a low precision configuration is used. The code also calculates the number of tiles that should be calculated and it returns the tile's sizes in case of a successful tiling.

**Listing 4.7:** Tiling algorithm DW-Conv

```
1  int get_tiling(int Hin, int Win, int Cin, int ker, int* Cin_t, int* Hin_t) {
2      int mem_in, mem_w, mem_out;
3      mem_update(Hin, Win, Cin, ker, &mem_in, &mem_w, &mem_out);
4      printf("Available local input memory is:\t%d\tMemory needed is:\t%d\n",
    plm_in, mem_in);
5      printf("Available local weight memory is:\t%d\tMemory needed is:\t%d\n",
    plm_w, mem_w);
6      printf("Available local output memory is:\t%d\tMemory needed is:\t%d\n",
    plm_out, mem_out);
7
8      // Tile across Cin dimension dividing by 2 until it fits into memory
    otherwise go to next step
9     *Cin_t = Cin;
10     int i = 1;
11     if ((plm_in < mem_in || plm_w < mem_w || plm_out < mem_out || *Cin_t >
    N_C_MAX)) {
12     if (Cin >= PE) {
13             *Cin_t = PE;
14             mem_update(Hin, Win, *Cin_t, ker, &mem_in, &mem_w, &mem_out);
15             printf("Available local weight memory is:\t%d\tWeight memory
    needed after tiling Cin is:\t%d\n", plm_w, mem_w);
16             printf("Available local output memory is:\t%d\tOutput memory
    needed after tiling Cin is:\t%d\n", plm_out, mem_out);
```

```
17          } else {
18              printf("No tiling in Cin was performed\n");
19          }
20      }
21      // Tile across Hin dimension by stripes with height equal to kernel's
        height
22      *Hin_t = Hin;
23      if (plm_in < mem_in || plm_out < mem_out || *Hin_t > N_H_IN_MAX) {
24          *Hin_t = ker;
25          mem_update(*Hin_t, Win, *Cin_t, ker, &mem_in, &mem_w, &mem_out);
26          printf("Available input memory is:\t%d\tInput memory needed after
        tiling Hin is:\t%d\n", plm_in, mem_in);
27          printf("Available output memory is:\t%d\tOutput memory needed after
        tiling Hin is:\t%d\n", plm_out, mem_out);
28      }
29      // Tile across Cin again giving up performance
30      while (*Cin_t > 1 && (plm_w < mem_w || plm_out < mem_out)) {
31          //*Cout_t = (int)ceil(*Cout_t / 2.0);
32      *Cin_t = (*Cin_t % 2 == 0)? (*Cin_t / 2) : (*Cin_t / 2 + 1);
33          mem_update(*Hin_t, Win, *Cin_t, ker, &mem_in, &mem_w, &mem_out);
34          printf("Available local weight memory is:\t%d\tWeight memory needed
        after tiling Cin is:\t%d\n", plm_w, mem_w);
35          printf("Available local output memory is:\t%d\tOutput memory needed
        after tiling Cin is:\t%d\n", plm_out, mem_out);
36      }
37      if (plm_in < mem_in || plm_w < mem_w || plm_out < mem_out || Win >
       N_W_IN_MAX || *Hin_t > N_H_IN_MAX ||  *Cin_t > N_C_MAX ) {
38          printf("Layer impossible to tile, larger memory needed\n");
39          return -1;
40      } else {
41          printf("Tile dimensions are - Hin x Win x Cin: %d %d %d \n", *Hin_t,
       Win, *Cin_t);
42          int cin_it = (Cin % (*Cin_t)==0)? (Cin/ (*Cin_t)) : (Cin/ (*Cin_t) +
       1);
43          if (*Hin_t<Hin)
44              printf("Total number of tiles(aprox): %d\n",(get_h_iterations() *
         cin_it ));
45          else
46              printf("Total number of tiles(aprox): %d\n",(cin_it));
47          return 0;
48      }
49  }
```

### 4.2.3 Fully-connected Tiling

As the previous cases, the FC Tiling follows the a similar procedure to obtain the tiling and to satisfy the memory constraints. However, in this case the first dimension to be tiled is the output activations dimension taking into consideration the number of PEs present in the accelerator. Afterwards, the tiling across input activations dimension is done considering the memory constraints and the maximum number of input activations supported by

the accelerator ($N_MAX$). An approximation to the upper multiply is also performed in case the division gives as result values different from multiplies of 2 or 4, this last step is done just in the case of using the lower precision configurations.

**Listing 4.8:** Tiling algorithm FC

```
int get_tiling(int N, int M, int* N_t, int* M_t) {
 int mem_in, mem_w, mem_out;
 mem_update(N,M ,&mem_in, &mem_w, &mem_out);
 printf("Available local input memory is:\t%d\tMemory needed is:\t%d\n", plm_in, mem_in);
 printf("Available local weight memory is:\t%d\tMemory needed is:\t%d\n", plm_w, mem_w);
 printf("Available local output memory is:\t%d\tMemory needed is:\t%d\n", plm_out, mem_out);
 *M_t = M;
 if ((plm_in < mem_in || plm_w < mem_w || plm_out < mem_out || *M_t > M_MAX)) {
if (M >= PE) {
        *M_t = PE;
        mem_update( N,*M_t, &mem_in, &mem_w, &mem_out);
        printf("Available local weight memory is:\t%d\tWeight memory needed after tiling M is:\t%d\n", plm_w, mem_w);
        printf("Available local output memory is:\t%d\tOutput memory needed after tiling M is:\t%d\n", plm_out, mem_out);
    } else {
        printf("No tiling in M was performed\n");
    }
 }
 *N_t = N;
int i = 1;
while ((plm_in < mem_in || plm_w < mem_w || plm_out < mem_out || *N_t > N_MAX)) {
    if (precision_opt == 0){
        if (*N_t == 1) break;
    } else if (precision_opt == 1) {
        if (*N_t == 4) break;
    } else{ // precision_opt == 2 || precision_opt == 3
        if (*N_t == 2) break;
    }
    *N_t = (*N_t / 2) ;
    if (precision_opt == 1){
        if (*N_t % 4 != 0)
            *N_t += *N_t % 4;
    } else if (precision_opt == 2 || precision_opt == 3){
        if (*N_t % 2 != 0)
            *N_t += 1;
    }
    mem_update(*N_t, *M_t, &mem_in, &mem_w, &mem_out);
    printf("Available input memory is:\t%d\tInput memory needed after tiling N=%d %d time(s) is:\t%d\n", plm_in, *N_t, i, mem_in);
    printf("Available weight memory is:\t%d\tWeight memory needed after tiling N=%d %d time(s) is:\t%d\n", plm_w, *N_t, i, mem_w);
    i++;
```

```
40      }
41      // Tile across M again giving up performance
42      while (*M_t > 1 && (plm_w < mem_w || plm_out < mem_out)) {
43      *M_t = (*M_t % 2 == 0)? (*M_t / 2) : (*M_t / 2 + 1);
44          mem_update(*N_t, *M_t, &mem_in, &mem_w, &mem_out);
45          printf("Available local weight memory is:\t%d\tWeight memory needed
        after tiling M is:\t%d\n", plm_w, mem_w);
46          printf("Available local output memory is:\t%d\tOutput memory needed
        after tiling M is:\t%d\n", plm_out, mem_out);
47      }
48      if (plm_in < mem_in || plm_w < mem_w || plm_out < mem_out || *N_t > N_MAX
        || *M_t > M_MAX ) {
49          printf("Layer impossible to tile, larger memory needed\n");
50          return -1;
51      } else {
52          printf("Tile dimensions are - N x M : %d %d  \n", *N_t, *M_t);
53          int N_it = (N % (*N_t)==0)? (N/ (*N_t)) : (N/ (*N_t) + 1);
54    int M_it = (M % (*M_t)==0)? (M/ (*M_t)) : (M/ (*M_t) + 1);
55          printf("Total number of tiles(aprox): %d\n",(N_it * M_it ));
56          return 0;
57      }
58  }
```

### 4.2.4 2D Convolution Tiled Process and Loops Order

Once the tile sizes are computed, the next step is to decide the processing order of the tiles to obtain the best performance of the system and try to reduce the memory transfers if partial results are generated. On the other hand, the addressing of the tiles has to be managed to access the data correctly and to store the output data in the correct addresses. Figure 4.4 shows how the tensor data is stored in external memory for the architecture proposed. The picture represents an example of a tensor with the following dimensions: 4x2x2x2 Width - Height - Input channels - Output channels. The values are packed first considering the width dimension or the rows, then each row is packed considering the height dimension. The following dimension to group is the input channel, placing the tensor *Width-Height-Input_Channel* consecutively. Finally, the last packing is done considering the output channels, defining the outermost index as the channel output of the tensor. This is the more complex case and corresponds to a weight tensor; in case of the input and output tensor just the first three dimensions are considered, but the order remains the same.

Figure 4.5 depicts the external memory organization of input, weight and output tensors. To address each tensor three pointers are used, each of them pointing to the beginning of each memory region. Hence these three pointers

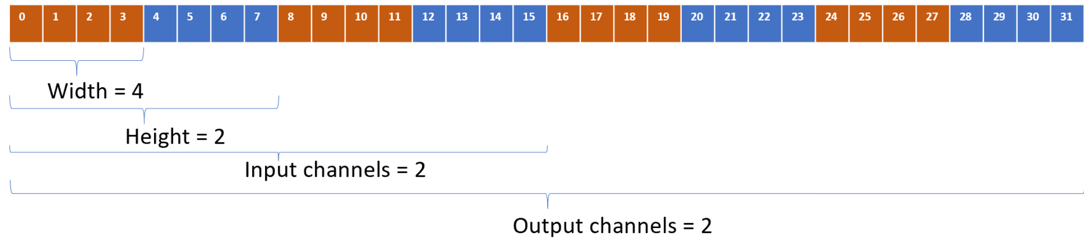


**Figure 4.4:** Tensor data organization in external memory

will serve as the reference when a new tile must be addressed. The software that is in charge of invoking the accelerator to perform the convolution operation should also calculate the offsets needed by the pointers to address the corresponding tiles. Therefore, the three pointers are continuously updated with the offsets. The tile sizes are also checked in case there is present an uneven division in the last iteration.

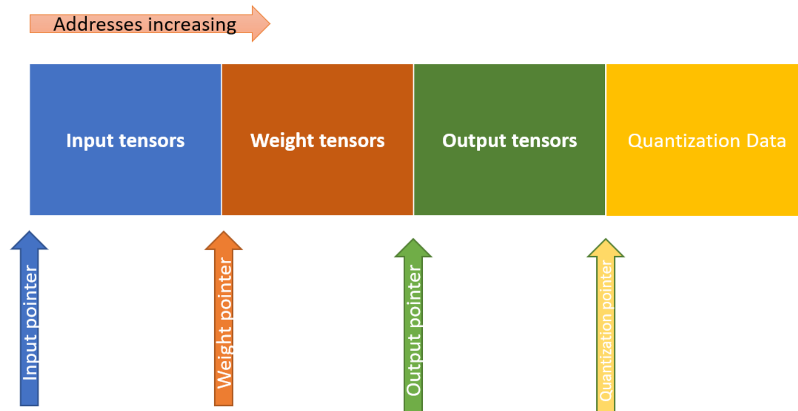


**Figure 4.5:** Organization of the tensors in external memory

The method which controls the convolution tiling organization has the following prototype:

**Listing 4.9:** Prototype 2D-Conv tiled

```
1   int conv2d_tiling (token_t *input,token_t *weight, token_t *output,
    uint32_t offset_data);
```

It receives the three pointers mentioned above: *input, weight,* and *output* and an additionally pointer for the quantization data. The last will be explain

later in the document. The code defines the values and flags that control the processing of the tiles. Table 4.1 explain the function of each variable used in this part of the code. The tiling algorithm is also called, however it can be also performed offline and only the tile sizes must be provided to this part of the code.

**Listing 4.10:** Initialization of parameters for 2D-Conv

```
int conv2d_tiling (token_t *input,token_t *weight, token_t *output, uint32_t
    offset_data ) {
    int Cout_t, Cin_t, Hin_t;
    int Cout_t_aux, Cin_t_aux, Hin_t_aux;
    int Hin_acc = 0;
    int Cin_acc = 0;
    int Cout_acc = 0;
    int offset_PE = 0;
    int offset_PE_out = 0;
    int acc_flag = 0;
    int q_flag = 0;
    int update_reg = 1;
    int offset_read_ci = Hin*Win;
    int tiling = get_tiling(Hin, Win, Cin, Cout, ker, &Cout_t, &Cin_t, &Hin_t)
    ;
    if (tiling != 0)  // If tiling can't be done
        printf ("Tiling infeasible\n");
        return -1;
    Cout_t_aux = Cout_t;
    Cin_t_aux = Cin_t;
    Hin_t_aux = Hin_t;
    token_t* in_p = input;
    token_t* w_p = weight;
    token_t* out_p = output;
    int tile = 0;
    uint8_t pad_type = 0;
    if (PE>1){
        offset_PE = Cin * ker * ker;
        offset_PE_out = Wout*Hout;
    }
    ...

```

| Variable | Description |
|---|---|
| `Cout, Cin, Hin` | This variables store the sizes of the entire tensors for the output channel, input channel and height dimensions respectively |
| `Cout_t, Cin_t, Hin_t` | This variables store the sizes of the tile |

| | |
|---|---|
| Cout_t_aux, Cin_t_aux, Hin_t_aux | Auxiliary variables to restore the size values |
| Hin_acc, Cin_acc, Cout_acc | These variables accumulate the values of tile sizes at each iteration to compare them with the original sizes. If the accumulation value is greater than the original a correction has to be done |
| offset_PE | It stores an offset value used when a tiled weight tensor is read. If the tensor is tiled across input channel and output channel dimensions, the tiled values will be split in different chunks inter-spaced by a size equal to this variable |
| offset_PE_out | It stores the an offset value used when a tiled output tensor is written. If the tensor is tiled in height dimension, the output values will be split in chunks inter-spaced by a size equal to this variable |
| acc_flag | Variable used to enable accumulation, it is needed when processing input channels in different tiles and partial results are generated |
| q_flag | Variable used to activate quantization in the accelerator, it is enabled at the end of the input channel processing, i.e. when no more partial results are generated |
| update_reg | It is used to enable the update of some user configurations registers for the accelerator, not all the registers have to be updated in each iteration |
| offset_read_ci | It is an offset value used when a tiled input tensor is read and the tiling has split the data into chunks inter-spaced by a size equal to this variable. This case happens when a low precision configuration is used |

| | |
|---|---|
| `pad_type` | This variable helps to control the behaviour of the padding in the accelerator depending on how the tile has been split. If the variable is 1, the padding is done on the sides and upper part of the tile. If it is 2, the padding is done on the sides and the lower part of the tile. If it is 3, the padding is done only on the sides of the tile. Finally, if it is 0, the padding is done on all the sides |

**Table 4.1:** Variables definition for 2D-Conv tiled

The first loop to iterate is the one that goes through the output channels. In this case two partial offsets are calculated, one for the weight tensor and another for the output tensor. On the other hand the if condition evaluates if the accumulator value is greater than the original channel output size, if so a correction in the size of the tile is done and the update flag is set to change all the user registers values in the next accelerator invocation.

**Listing 4.11:** Iteration through output channels

```
// Iterate through the number of tiles in Cout channel and calculate the
    address offset for output and weight pointers
    int tempp = (Cout % Cout_t == 0)? (Cout / Cout_t):(Cout / Cout_t +1);
    for (int co = 0 ,Cout_acc=0; co < tempp; co++) {
        token_t* out_p_co = ((co * Cout_t) * Hout * Wout) + output;
        token_t* w_p_co = ((co * Cout_t) * Cin * ker * ker) + weight;
        Cout_acc += Cout_t;
        if (Cout_acc>Cout){
            Cout_t -= (Cout_acc - Cout);
            update_reg = 1;
        }
```

After this point the workflow is divided in two cases: in one case the tiling is done in height and input channel dimensions and in the other the tiling is done just in input channel dimension as seen in Lst. 4.12. Let us see first the case where both dimensions have been tiled.
The next iteration loop is the one that goes through the height dimension. The choice of going first with the height dimension instead of the channel input dimension is taken due to the behaviour of the accelerator. The accelerator used in this thesis has an output stationary dataflow, hence it keeps the output results in memory and permits the reuse of these values in the following iteration. In this context, and taking into account the

computation of the output values in 2D-Conv where the results of the input channels have to be accumulated, it is a reasonable choice to let the channel input loop be the inner part of the code. Therefore, at every new iteration, the partial results computed by the accelerator can be accumulated with the next values in the channel input dimension.
In the height loop, the partial offsets are calculated for the input and output tiles. In the case of the output offset, the partial offset value of the outermost loop is added to get the final offset value for the output tile. Then, the height accumulator (*Hin_acc*) is updated and used to check if the value has over-passed the original height size. If the previous condition is true, the size correction is performed to the tile and the update flag is also set. The size correction has to be done because of an uneven tiling of the tensor can lead to a different size of the tile at the last iteration. In this part the configuration in case of padding is also set. Depending on the iteration, the value of the variable *pad_type* is adjusted. If the first iteration is computed, the variable value is 1 which means to apply the padding to the sides and the upper part. If the last iteration is computed, the variable value is 2 hence the padding is done in the sides and lower part. In the other cases the variable value is 3 hence the padding is done just in the sides of the tensor. The output pointer has to be corrected if padding is used. The correction is done after the first iteration ($h > 0$).
The last iteration loop goes through the channel input dimension. The loop bound is calculated and approximated to the upper integer number. Then, partial offsets are computed and added to the previous partial offsets to generate the final offsets for the input and weight tiles. As the previous cases, a size correction is performed if the accumulator value is greater than the original channel input size of the tensor. In this part the flags to accumulate partial results and to enable quantization are also set. The *acc_flag* is zero, which means that the accelerator must reset the accumulator buffer, when the first iteration is computed and is 1 in all the other iterations, which means that the accelerator must accumulate the partial results. On the other hand, the *q_flag* is set when the accumulator contains the final result, i.e. the last tile iteration is computed, while is zero in all the other iterations. Finally, the code invokes the accelerator to start the convolution operation passing the tile sizes,the stride, the number of padding pixels, the type of padding, the pointers to the tile tensors, the offsets for reading and writing data on memory, and the flags to manage accumulation, precision and quantization.

**Listing 4.12:** Case H and Cin tiled - H and Cin loops iterations

```
    if (Hin_t < Hin) {
        // Iterate through the number of tiles in Hin dimension and calculate the offset for input and output pointers
        for (int h = 0, Hin_acc = Hin_t - stride, Hin_t = Hin_t_aux; h < get_h_iterations(); h++) {
            in_p = input + (h * stride * Win);
            out_p = out_p_co + (h * Wout)  ;
            tile++;
            Hin_acc += stride;
            if (Hin_acc>Hin){
                Hin_t -= (Hin_acc -Hin);
                update_reg = 1;
            }
            if (pad > 0){
                if (h==0)
                    pad_type = 1; // padding sides and upper part
                else if (h==get_h_iterations()-1)
                    pad_type = 2; // padding sides and lower part
                else
                    pad_type = 3; // padding just sides
                if (h > 0){
                    out_p +=  Wout*(pad/stride) ; // offset correction in case of padding
                }
            }
        // Iterate through the number of tiles in Cin channel and calculate the offset for input and weight pointers
        int temp1 = (Cin % Cin_t == 0)? (Cin / Cin_t):(Cin / Cin_t +1);
        for (int ci = 0, Cin_acc=0, Cin_t = Cin_t_aux; ci< temp1; ci++) {
                token_t* in_p_ci = (ci * Cin_t * Hin * Win) + in_p;
                w_p = w_p_co + (ci * Cin_t * ker * ker);
                Cin_acc += Cin_t;
                if (Cin_acc>Cin){
                    Cin_t -= (Cin_acc-Cin);
                    update_reg = 1;
                }
                if(ci == 0)
                    acc_flag = 0;
                else
                    acc_flag = 1;
                if (ci == temp1 -1 && output_q_en)
                    q_flag = 1;
                else
                    q_flag = 0;
                conv2d_hw ((int32_t)(in_p_ci - input), (int32_t)(w_p - input) ,(int32_t) (out_p - input), Win, Hin_t, Cin_t, Cout_t, pad, pad_type,offset_PE,offset_PE_out,precision_opt, acc_flag, q_flag, update_reg, offset_q_data, offset_read_ci);
                printf ("Tile completed\n");
                update_reg = 0;
            }
        }
    }
```

The second case in the workflow is present when the tiling is done only in

the the channel input dimension. This case is similar to the previous scenario with the difference that the loop iteration across the height dimension is omitted. The partial offsets are calculated for the input and weight pointers. The weight partial offset is added with the previous value calculated in the channel output loop to obtain the final pointer for the weight tile, shown in line 6 of Lst 4.13. In the case of the input pointer, it is directly generated in this loop. The correction in case of the tile size and the control of the quantization flag are executed in the same way as the previous case. The configuration of the *pad_type* in this case is set to force the padding around all the tensor because no tiling has been done in the height dimension. Finally, the accelerator is invoked to compute the 2D-Conv with all the parameters previously defined.

**Listing 4.13:** Case only Cin tiled - Cin loop iteration

```
    else{
        // Iterate through the number of tiles in Cin channel and calculate
    the offset for input and weight pointers
    int temp1 = (Cin % Cin_t == 0)? (Cin / Cin_t):(Cin / Cin_t +1);
    for (int ci = 0, Cin_acc=0, Cin_t = Cin_t_aux; ci< temp1; ci++) {
                token_t* in_p_ci = (ci * Cin_t * Hin * Win) + input;
                w_p = w_p_co + (ci * Cin_t * ker * ker);
                Cin_acc += Cin_t;
                if (Cin_acc>Cin){
                    Cin_t -= (Cin_acc-Cin);
                    update_reg = 1;
                }
                tile++;
                if (pad > 0)
                    pad_type = 4; // padding in all sides
                if(ci == 0)
                    acc_flag = 0;
                else
                    acc_flag = 1;
                if (ci == temp1 - 1 && output_q_en)
                    q_flag = 1;
                else
                    q_flag = 0;
                conv2d_hw ((int32_t)(in_p_ci - input), (int32_t)(w_p - input)
    ,(int32_t) (out_p - input), Win, Hin_t, Cin_t, Cout_t, pad, pad_type,
    offset_PE,offset_PE_out,precision_opt, acc_flag, q_flag, update_reg,
    offset_q_data, offset_read_ci  );
                printf ("Tile completed\n");
                update_reg = 0;
            }
        }
```

### 4.2.5 Depthwise Convolution Tiled Process and Loops Order

The DW-Conv tiled process is similar to the 2D-Conv tiled process. However, there are some differences in the use of the variables for managing the offsets, in the computing of the partials offsets, and also in the loop order executed in the process. The table 4.2 defines the variables which change their definitions with respect to 2D-Conv process.

| **Variable** | **Description** |
|---|---|
| `Cin_t, Hin_t` | Same behaviour as previous 2D-Conv case |
| `Cin_t_aux, Hin_t_aux` | Same behaviour as previous 2D-Conv case |
| `Hin_acc, Cin_acc` | Same behaviour as previous 2D-Conv case |
| `offset_PE` | It stores an offset value used when a tiled weight tensor is read. The value differs from the 2D-Conv analogous value because in this case the weight tensor has only three dimensions (height-width-channels), consequently the offset value is smaller, i.e kernel height x kernel width. |
| `offset_PE_out` | Same behaviour as previous 2D-Conv case |
| `acc_flag` | No accumulation is performed in this case. Hence the variable is always 0 |
| `q_flag` | Variable used to activate quantization. Due to the topology of DW-Conv, the variable goes directly to the accelerator invocation. |
| `update_reg` | Same behaviour as previous 2D-Conv case |
| `offset_read_ci` | Same behaviour as previous 2D-Conv case |
| `pad_type` | Same behaviour as previous 2D-Conv case |

**Table 4.2:** Variables definition for DW-Conv tiled

This process begins with the loop that goes through the channels dimension because in this case there is no restriction or constraint for accumulating partial results. Therefore, three partial offsets are calculated, each of them corresponding to the input, weight, and output tensor pointers. The tile size correction is checked and performed in case is necessary. Then, the workflow is also divided depending if the Height tiling is performed or not. In case it is not performed, the code invokes the DW-Conv accelerator passing the partial

offsets calculated in the previous loop and all the parameters related to the tile sizes and operation configuration. On the other hand, when height tiling is done, partial offsets are calculated for the input and output tensor pointers, then they are added with the partial offsets calculated in the previous loop obtaining the final offsets. The padding configuration is also executed in this case and the output offset correction as well. Finally, the DW-Conv accelerator is called to begin the computation of the tile with the parameters defined and calculated before the accelerator's invocation.

**Listing 4.14:** DW-Conv tiled process

```
int depthwise_tiling (token_t *input,token_t *weight, token_t *output,
    uint32_t offset_data ) {
    int Cin_t, Hin_t;
    int Cin_t_aux, Hin_t_aux;
    int Hin_acc = 0;
    int Cin_acc = 0;
    int offset_PE = 0;
    int offset_PE_out = 0;
    int acc_flag = 0;
    int q_flag = 0;
    int update_reg = 1;
    uint32_t offset_q_data = offset_data;
    int32_t offset_read_ci = Hin*Win;
    int tiling = get_tiling(Hin, Win, Cin, ker, &Cin_t, &Hin_t);
    if (tiling != 0)  // If tiling can't be done
        printf ("Tiling infeasible\n");
        return -1;
    Cin_t_aux = Cin_t;
    Hin_t_aux = Hin_t;
    token_t* in_p = input;
    token_t* w_p = weight;
    token_t* out_p = output;
    int tile = 0;
    uint8_t pad_type = 0;
    if (PE>1){
        offset_PE = ker * ker;
        offset_PE_out = Wout*Hout;
    }
    int temp1 = (Cin % Cin_t == 0)? (Cin / Cin_t):(Cin / Cin_t +1);
    for (int ci = 0, Cin_acc=0, Cin_t = Cin_t_aux; ci< temp1; ci++) {
        token_t* in_p_ci = (ci * Cin_t * Hin * Win) + input;
        token_t* w_p_ci = (ci * Cin_t * ker * ker) + weight;
  token_t* out_p_ci = (ci * Cin_t * Hout * Wout) + output;
        Cin_acc += Cin_t;
        if (Cin_acc>Cin){
            Cin_t -= (Cin_acc-Cin);
            update_reg = 1;
        }
        if (Hin_t < Hin) {
            // Iterate through the number of tiles in Hin channel and
    calculate the offset for input and output pointers
```

```
40            for (int h = 0, Hin_acc = Hin_t - stride, Hin_t = Hin_t_aux; h <
     get_h_iterations(); h++) {
41                in_p = in_p_ci + (h * stride * Win);
42                out_p = out_p_ci + (h * Wout)   ;
43                //printf("Tile: %d Cout: %d Cin: %d Hin: %d Input pointer is:
     %d Output pointer is: %d\n", tile, co * Cout_t, ci * Cin_t, h * stride,
     in_p, out_p);
44                tile++;
45                Hin_acc += stride;
46                if (Hin_acc>Hin){
47                    Hin_t -= (Hin_acc -Hin);
48                    update_reg = 1;
49                }
50                //printf ("Tile %d\n", tile);
51                if (pad > 0){
52                    if (h==0)
53                        pad_type = 1; // padding sides and upper part
54                    else if (h==get_h_iterations()-1)
55                        pad_type = 2; // padding sides and lower part
56                    else
57                        pad_type = 3; // padding just sides
58                    if (h > 0){
59                        out_p +=  Wout*(pad/stride) ;
60                    }
61                }
62                    depthwise_hw ((int32_t)(in_p - input), (int32_t)(w_p_ci -
     input) ,(int32_t) (out_p - input), Win, Hin_t, Cin_t, pad, pad_type,
     offset_PE,offset_PE_out,precision_opt, acc_flag, q_flag, update_reg,
     offset_q_data, offset_read_ci  );
63                    printf ("Tile completed\n");
64                    update_reg = 0;
65                }
66        } else{
67            depthwise_hw ((int32_t)(in_p_ci - input), (int32_t)(w_p_ci - input
     ) ,(int32_t) (out_p_ci - input), Win, Hin_t, Cin_t, pad, pad_type,
     offset_PE,offset_PE_out,precision_opt, acc_flag, q_flag, update_reg,
     offset_q_data, offset_read_ci  );
68            printf ("Tile completed\n");
69            update_reg = 0;
70        }
71    }
72    return 0;
73 }
```

### 4.2.6 Fully-connected Tiled Process and Loops Order

The FC tile process is the simplest case of the three because it has less dimensions to be considered. Still, it is worth highlighting the differences with respect to the previous cases. Table 4.3 defines the variables which change their definitions with respect to 2D-Conv process.

| Variable | Description |
|---|---|
| N_t, M_t | Same behaviour as previous 2D-Conv values Cout_t, Cin_t, Hin_t |
| M_t_aux, N_t_aux | Same behaviour as previous 2D-Conv values Cout_t_aux, Cin_t_aux, Hin_t_aux |
| N_acc, M_acc | Same behaviour as previous 2D-Conv values Hin_acc, Cin_acc, Cout_acc |
| offset_PE | It stores an offset value used when a tiled weight tensor is read. The value differs from the 2D-Conv analogous value because in this case the weight tensor has only two dimensions (N-M), consequently the offset value is smaller, i.e just the value of the input activations width. |
| offset_PE_out | This variable is not needed because the output values will not be separated when they will be written on memory |
| acc_flag | Same behaviour as previous 2D-Conv case |
| q_flag | Same behaviour as previous 2D-Conv case |
| update_reg | Same behaviour as previous 2D-Conv case |
| offset_read_ci | This variable is not needed in this case because the input activations have just one dimension |
| pad_type | No padding is perform in this layer |

**Table 4.3:** Variables definition for FC tiled

The process has only two loop iterations to be considered, in this case the order of these loops matters because an accumulation of partial results is performed. Hence the first loop to be executed is the output activations loop. The offset for the output pointer is calculated and the partial offset for the weight pointer is also calculated. If necessary, the correction of the tile size is done for the output activation dimension. At this point, depending if the tiling across the input activations was done, the code invokes the accelerator or continue to set the offsets and parameters.
The second loop to execute is the input activations loop. In this case the input tensor offset is calculated and a partial weight tensor offset is also computed. The latter is added with the value calculated in the previous loop to obtain the final offset value for the weight pointer. If necessary, the correction of the tile size is done for the input activation dimension. The

accumulation and quantization flags are managed in the same way as the 2D-Conv case. Finally, the FC accelerator is invoked to start its computation with the parameters, pointers and flags configured before.

**Listing 4.15:** FC tiled process

```
int fc_tiling (token_t *input,token_t *weight, token_t *output, uint32_t
    offset_data ) {
    int N_t, M_t;
    int M_t_aux, N_t_aux;
    int N_acc = 0;
    int M_acc = 0;
    int offset_PE = 0;
    int acc_flag = 0;
    int q_flag = 0;
    int update_reg = 1;
    uint32_t offset_q_data = offset_data;
    int tiling = get_tiling(N, M, &N_t, &M_t);
    if (tiling != 0)  // If tiling can't be done
        printf ("Tiling infeasible\n");
        return -1;
    M_t_aux = M_t;
    N_t_aux = N_t;
    token_t* in_p = input;
    int tile = 0;
    if (PE>1){
        offset_PE = N;
    }
    int temp1 = (M % M_t == 0)? (M / M_t):(M / M_t +1);
    for (int mi = 0, M_acc=0; mi< temp1; mi++) {
        token_t* w_p_mi = (mi * M_t * N) + weight;
  token_t* out_p = (mi * M_t) + output;
        M_acc += M_t;
        if (M_acc>M){
            M_t -= (M_acc-M);
            update_reg = 1;
        }
        if (N_t < N) {
            int temp2 = (N % N_t == 0)? (N / N_t):(N / N_t +1);
            for (int ni = 0, N_acc = 0 , N_t = N_t_aux; ni < temp2; ni++) {
      in_p = input + ni*N_t;
                token_t* w_p = w_p_mi + ni*N_t   ;
                N_acc += N_t;
                if (N_acc>N){
                    N_t -= (N_acc -N);
                    update_reg = 1;
                }
      if(ni == 0)
                    acc_flag = 0;
                else
                    acc_flag = 1;
                if (ni == temp2 -1 && output_q_en)
                    q_flag = 1;
                else
                    q_flag = 0;
```

```
49                  fc_hw ((int32_t)(in_p - input), (int32_t)(w_p - input) ,(
    int32_t) (out_p - input), N_t, M_t, precision_opt, acc_flag, q_flag,
    update_reg, offset_q_data, offset_PE);
50                  printf ("Tile completed\n");
51                  update_reg = 0;
52              }
53          } else{
54              fc_hw ((int32_t)(in_p - input), (int32_t)(w_p_mi - input) ,(
    int32_t) (out_p - input), N, M_t, precision_opt, 1, output_q_en,
    update_reg, offset_q_data, offset_PE);
55              printf ("Tile completed\n");
56              update_reg = 0;
57          }
58      }
59      return 0;
60  }
```

## 4.3 Hardware Implementation

In this section the hardware implementation of the accelerators used in the tiling architecture are explained with the help of previous works [19] [20]. In these works the workflow followed to implement an CNN accelerator is explained from the very basic steps. It is suggested to check these works before continue reading the following design workflow. This document focuses in explaining the differences from the previous works in order to explain the design flow of the hardware implementation of the CNN accelerators.
The proposed accelerators maintain the interfaces used in previous works. However, the data which is shared through these port interfaces is not the same and will be explained in detail following in this section. Figure 4.6 depicts the ESP accelerator interfaces and signals needed for the integration of the accelerator into the SoC. The accelerators execute the four main phases to perform the complete operation, which are: configuration, load, compute, and store. This work only considers the sequential architecture, but a hierarchical architecture can be implemented as well and will be part of future work. Below each accelerator design flow will be explained:

### 4.3.1 2D Convolution accelerator

The accelerator used in this thesis is the PS accelerator proposed in [16]. As explained at the beginning of Ch. 4, it has multiple PEs, each of them work independently on a different output channel. Each processing element is composed by PS multiply and accumulate units based on Sum-together

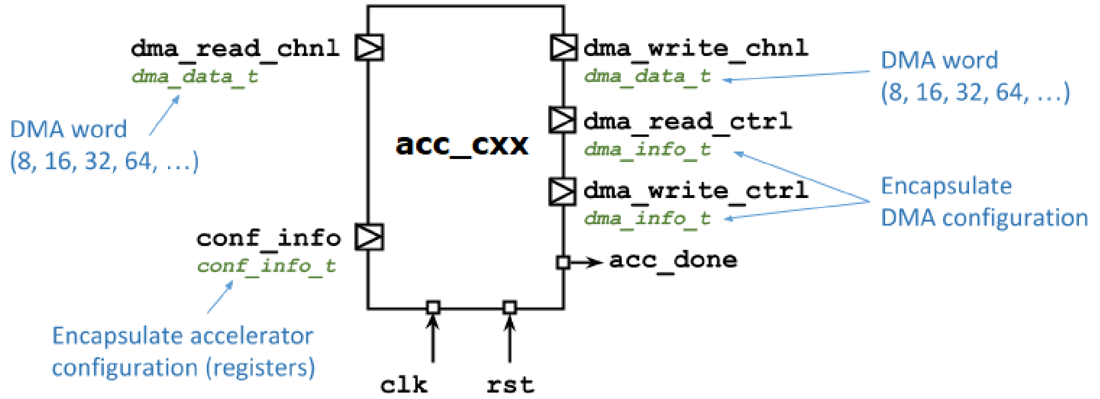


**Figure 4.6:** Interfaces for a generic ESP accelerator

multipliers. This feature permits to have a configurable precision for the computation of the convolution operations. Furthermore, the accelerator also enables the quantization of the output data for three different bitwidths: 4, 8, and 16 bits.

**Configuration Phase** During the configuration phase, the configuration parameters held in the memory-mapped registers and set by the processor are retrieved through the *conf_info* port. Information transmitted through this *ac_channel* is then organized and stored in the appropriate *conf_info_t* structure. This method enables the assignment of each parameter to a local variable, facilitating their utilization in subsequent phases. The maximum number of user-defined registers is 14 [21], which is defined by the ESP accelerator specifications. This suppose a challenge for the architecture proposed due to the need of at least 20 parameters explained in table 4.4. To address this challenge, some parameters have to be packed inside a single register, when written, and then this data has to be unpacked using masks and bitwise or and operations, when read.

| Parameter | Description |
|---|---|
| `in_add` | Pointer or offset to the input tile |
| `w_add` | Pointer or offset to the weight tile |
| `out_add` | Pointer or offset to the output tile |
| `q_flag` | Flag that enables the quantization operation |

| | |
|---|---|
| `n_w` | Width size of the input tile |
| `n_h` | Height size of the input tile |
| `n_c` | Input channel size of the input and weight tile |
| `kern` | Width and Height size of the weight tile. Since most of the times a symetric kernel is used, these dimensions are cosidered the same |
| `filt` | Output channel size of the output and weight tile |
| `pad` | Number of pixels to apply the padding operation |
| `pad_type` | Configuration variable that indicates where to apply padding around the tensor |
| `stride` | Number of positions the kernel slides |
| `CONFIG1` | Configuration variable that specifies the precision (4, 8, or 16 bit) set for the multiply and accumulate operations |
| `offset_PE` | Offset value used to read inter-spaced weight values from different output channels |
| `offset_PE_out` | Offset value used to write inter-spaced output values to different output channels |
| `offset_read_ci` | Offset value used to read inter-spaced input values from different input channels |

**Table 4.4:** List of parameters to configure the 2D-Conv accelerator

In the configuration phase the parameters are unpacked and read as it is shown in the following code:

**Listing 4.16:** Configuration phase for 2D-Conv

```
// Read accelerator configuration
#ifndef __SYNTHESIS__
    while (!conf_info.available(1)) {} // Hardware stalls until data ready
#endif
    params = conf_info.read();
    in_add = params.in_add;
    w_add = params.w_add;
    out_add = params.out_add;
    acc_flag = (params.flags >>1) & 0x00000001;
    q_flag = (params.flags) & 0x00000001;
    relu_flag = (params.flags >>2) & 0x00000001;
    n_w  = params.n_w;
    n_h = params.n_h;
    n_c = params.n_c;
    kern = ((params.pad_stride_kern) & 0x0000F000)>>12;
    filt = params.filt;
```

```
17      pad = ((params.pad_stride_kern) & 0x0000000F);
18      pad_type = ((params.pad_stride_kern) & 0x00000F00)>>8;
19      stride = ((params.pad_stride_kern) & 0x000000F0)>>4;
20      offset_PE_out = params.offset_PE_out;
21      offset_PE = params.offset_PE;
22      CONFIG1 = ((params.options) & 0x0000000F);
23      CONFIG2 = ((params.options) & 0x000000F0) >> 4;
24      offset_q_data = params.offset_q_data;
25      offset_read_ci = params.offset_read_ci;
```

Before passing to the load phase, some auxiliary variables are defined to control the loop boundaries. *n_w_in* and *n_h_in* are the auxiliary variables that keep the width and height dimensions of the input tile after evaluating the type of padding to be applied. Instead, *n_w_out* and *n_h_out* keep the width and height dimensions of the output tile considering the padding value and the stride as well. They are shown in the following code snip:

**Listing 4.17:** Auxiliary variables

```
1          uint16_t n_w_in;
2     uint16_t n_h_in;
3     // Padded Input Dimensions
4     switch (pad_type)
5     {
6     case 0: // no padding
7       n_w_in = n_w ;
8       n_h_in = n_h ;
9       break;
10    case 1: // padding 3 sides
11      n_w_in = n_w + 2 * pad;
12      n_h_in = n_h + pad;
13      break;
14    case 2: // padding 3 sides
15      n_w_in = n_w + 2 * pad;
16      n_h_in = n_h + pad;
17      break;
18    case 3: // padding 2 sides
19      n_w_in = n_w + 2 * pad;
20      n_h_in = n_h;
21      break;
22    case 4: // padding 4 sides
23      n_w_in = n_w + 2 * pad;
24      n_h_in = n_h + 2 * pad;
25      break;
26    default:
27      n_w_in = n_w ;
28      n_h_in = n_h ;
29      break;
30    }
31    // Output Dimensions
32    uint16_t n_w_out;
33    uint16_t n_h_out;
34    if (stride == 1){
35      n_w_out = (n_w_in - kern) + 1;
```

```
36        n_h_out = (n_h_in - kern) + 1;
37      }
38      else{
39        n_w_out = (n_w_in - kern)/2 + 1;
40        n_h_out = (n_h_in - kern)/2 + 1;
41      }
42      // DMA variables
43        uint16_t dma_read_in_data_length = n_w * n_h ;
44        uint16_t dma_read_w_data_length = kern * kern * n_c ;
45        uint16_t dma_write_data_length = n_w_out * n_h_out ;
```

**Load Phase** In this phase the input and weight data of the corresponding tile are transferred from external memory to the private local memories (PLMs) of the accelerator. These data transfers are carried out by the direct-memory access (DMA) engine. As seen in Lst. 4.18 The load phase is composed by two parts, the load of the input tensor and the load of the weight tensor. The first part has three loops iterations, each of them corresponds to one dimension of the tensor. The order of the loops corresponds on how the input data are stored in external memory as explained before, hence the innermost loop iterates through the width dimension. The middle one iterates through the height dimension and the outermost iterates through the input channel dimension. In the outermost loop the DMA configuration is performed, here the offset to the data chunk to read and data length are passed to the DMA controller to start the transaction. In this case the data length is equal to the width size times the height size of the input tile. The input tile pointer, *in_add*, is added to the multiplication of the current input channel and the offset, *offset_read_ci*, which skips the reading to the next input channel of the entire tensor. The result of the previous operation gives the current offset for each iteration, *offset_read*. Afterwards, the hardware stalls until the DMA configuration is correctly written. After the serialization if-condition the other two loops are present. Within the nested loops (iterating over the input height and width), the code applies padding based on the specified padding type (*pad_type*), ensuring that the input data is appropriately padded or not for convolution operations. Depending on the padding type and position within the input matrix, a *pad_control* flag is set to determine whether to include or exclude the data during the loading phase. If the *pad_control* flag is true, indicating that the data from memory needs to be considered, otherwise a zero is read. In case data is read from memory, the code checks the readiness of the DMA read channel (*dma_read_chnl*) to ensure that the data is available before reading it. The

data read is manipulated to fit the bidwidth required by the accelerator, then it is stored in the input data *plm_in* at an appropriate index calculated based on the row, column, and channel. This special index is required to prepare the data in the PLMs in the way the accelerator needs it in the computation phase.

**Listing 4.18:** Input data load and padding control

```
1      for (uint16_t chan = 0; chan < N_C_MAX; chan++){
2          // Configure DMA read channel (CTRL)
3          uint32_t offset_read = in_add + chan*offset_read_ci;
4          dma_read_info = {offset_read, dma_read_in_data_length, DMA_SIZE};
5          bool dma_read_ctrl_done = false;
6          bool pad_control = true;
7      LOAD_CTRL_LOOP:
8          do { dma_read_ctrl_done = dma_read_ctrl.nb_write(dma_read_info); }
       while (!dma_read_ctrl_done);
9      // Force serialization between DMA control and DATA data transfer
10         if (dma_read_ctrl_done) {
11       for (uint16_t row = 0; row < N_H_IN_MAX; row++){
12         for (uint16_t col = 0; col <  N_W_IN_MAX; col++){
13           FPDATA_IN data;
14           switch (pad_type){
15           case 1:
16             if ((row >= pad) && (col >= pad) && (col <  n_w_in - pad))
17               pad_control = true;
18             else
19               pad_control = false;
20             break;
21           case 2:
22             if ((col >= pad) && (col <  n_w_in - pad) && (row < n_h_in - pad))
23               pad_control = true;
24             else
25               pad_control = false;
26             break;
27           case 3:
28             if ((col >= pad) && (col <  n_w_in - pad))
29               pad_control = true;
30             else
31               pad_control = false;
32             break;
33           case 4:
34             if ((row >= pad) && (col >= pad) && (col <  n_w_in - pad) && (row
       < n_h_in - pad))
35               pad_control = true;
36             else
37               pad_control = false;
38             break;
39           default:
40             pad_control = true;
41             break;
42           }
43           if (pad_control)
44           {
45     #ifndef __SYNTHESIS__
```

```
46              while (!dma_read_chnl.available(1)) {}; // Hardware stalls until
        data ready
47      #endif
48              ac_int<DATA_WIDTH, false> data_ac = dma_read_chnl.read().template
        slc<DATA_WIDTH>(0);
49              data.set_slc(0, data_ac);
50            }
51          else{
52              data = 0;
53          }
54                      uint16_t index_in = MAX_INPUT_CHANNELS * (
        MAX_INPUT_WIDTH * row + col) + chan;
55          plm_in.data[index_in] = data;
56          if (col == n_w_in - 1) break;
57        }
58        if (row == n_h_in - 1) break;
59        }
60      }
61        if (chan == n_c - 1 ) break;
62      }
```

Then is the turn of the weight tile to be read. This case has four loop iterations which are also ordered according to the organization of the weight data into the external memory. As the pevious case, here the outermost loop performs the DMA configuration; the offset and data length are passed to the DMA controller to start the transaction. In this case the data length is equal to the width size times the height size times the input channel size of the weight tile. The weight tile pointer, *w_add* , is added to the multiplication of the current output channel and the offset,*offset_PE*, which skips the reading to the next output channel of the entire tensor. The rest of the steps are similar to the input load case, but the internal PLM index has a bit different shape because of the presence of an extra dimension in this tensor. This index is also used to fit the memory organization needed in the computation phase.

**Listing 4.19:** Weight data load

```
1         weight_load_for:
2             for (uint16_t co = 0; co < FILT_MAX; co++){
3                 uint32_t offset_read = w_add + co*offset_PE;
4                 dma_read_info = {offset_read, dma_read_w_data_length, DMA_SIZE
        };
5             bool dma_read_ctrl_done2 = false;
6       LOAD_CTRL_LOOP2:
7                 do { dma_read_ctrl_done2 = dma_read_ctrl.nb_write(
        dma_read_info); } while (!dma_read_ctrl_done2);
8             if (dma_read_ctrl_done2) {
9                     LOAD_LOOP:
10                    for (uint16_t ci = 0; ci < N_C_MAX; ci++){
11                        for (uint16_t j = 0; j < KERN_MAX; j++){
12                            for (uint16_t i = 0; i < KERN_MAX; i++) {
```

```
13                              uint16_t index = MAX_OUTPUT_CHANNELS * (
   MAX_INPUT_CHANNELS * (KERN_MAX * j + i) + ci) + co;
14                          #ifndef __SYNTHESIS__
15                              while (!dma_read_chnl.available(1)) {}; //
   Hardware stalls until data ready
16                          #endif
17                              ac_int<DATA_WIDTH, false> data_ac =
   dma_read_chnl.read().template slc<DATA_WIDTH>(0);
18                              FPDATA_IN data;
19                              data.set_slc(0, data_ac);
20                              plm_f.data[index] = data;
21
22                              if (i == kern - 1) break;
23                          }
24                          if (j == kern - 1) break;
25                      }
26                      if (ci == n_c - 1) break;
27                  }
28          }
29              if (co == filt - 1) break;
30          }
```

The next data to be read are the parameters needed to perform quantization to the results of the convolution operation. In this case the DMA transaction is done just one time with a fixed offset, *offset_q_data*, and a data length equal to the number of output channels times 3 (because three parameters are meant for each output channel) plus 2 extra parameters which are common to all output channels. Each parameter is manipulated to fit the correct bitwidth intended for the efficient computation of the quantization. Finally, they are stored in their respective variables and they are ready to be used for the computation phase.

**Listing 4.20:** Quantization data load

```
1     if (q_flag){
2         uint16_t dma_read_q_data_length = filt * 3 + 2;
3         dma_read_q_info = {offset_q_data, dma_read_q_data_length, DMA_SIZE};
4     bool dma_read_ctrl_done3 = false;
5     LOAD_CTRL_LOOP3:
6         do { dma_read_ctrl_done3 = dma_read_ctrl.nb_write(dma_read_q_info); }
      while (!dma_read_ctrl_done3);
7       if (dma_read_ctrl_done3) {
8             LOAD_Q_LOOP:
9             for (uint16_t i = 0; i < (FILT_MAX*3 + 2); i++) {
10                #ifndef __SYNTHESIS__
11                while (!dma_read_chnl.available(1)) {}; // Hardware stalls
      until data ready
12                #endif
13                ac_int<W_CROSS_BITWIDTH, true> data_WC;
14                ac_int<SF_IN_W_TOT_BITWIDTH, true> data_SF;
15                ac_int<BIASQ_SCALED_TOT_BITWIDTH, true> data_BS;
16                ac_int<SF_OUT_INV_TOT_BITWIDTH, true> data_SFI;
17                ac_int<Z_BITWIDTH, true> data_Z;
```

```
18                  if (i < filt){
19                      data_WC = dma_read_chnl.read().template slc<
    W_CROSS_BITWIDTH>(0);
20                      WEIGHTS_CROSSPRODUCT[i].set_slc(0, data_WC);
21                  } else if (i >= filt && i < (filt*2)){
22                      data_SF = dma_read_chnl.read().template slc<
    SF_IN_W_TOT_BITWIDTH>(0);
23                      SCALING_FACTOR_INPUTS_WEIGHTS[i-filt].set_slc(0, data_SF);
24                  } else if (i >= (filt*2) && i < (filt*3)) {
25                      data_BS = dma_read_chnl.read().template slc<
    BIASQ_SCALED_TOT_BITWIDTH>(0);
26                      BIASQ_SCALED [i-filt*2].set_slc(0, data_BS);
27                  } else if (i == (filt*3)) {
28                      data_SFI = dma_read_chnl.read().template slc<
    SF_OUT_INV_TOT_BITWIDTH>(0);
29                      SCALING_FACTOR_OUT_INVERSE.set_slc(0, data_SFI);
30                  } else {
31                      data_Z = dma_read_chnl.read().template slc<Z_BITWIDTH>(0);
32                      Z_O2.set_slc(0, data_Z);
33                  }
34                  if (i == dma_read_q_data_length - 1) break;
35              }
36          }
37      }
```

Once the data is in the PLMs, it has to be arranged to fit the requirements of the computation unit. This last requires the data to be split in four groups, each of them intended for managing 4 bits of one multiplier-and-accumulate operand. Depending on the precision configuration selected, the operands can be formed in the following ways:

- Taking one 16-bit value from an input channel
- Taking two 8-bit values from two input channels
- Taking four 4-bit values from four input channels

The four groups of buffers are divided by the position that have when are used as operands, i.e. *A/B_reconf_HH* group all the values that contains the 4 MSB of the final operand, *A/B_reconf_HL* group all the values that contains the following 4 bits of the final operand, *A/B_reconf_LH* group all the values that contains the following 4 bits of the final operand, and *A/B_reconf_LL* group all the values that contains the 4 LSB of the final operand. The packing of the data has to take into account the cases where there is no more available data to read from memory, hence the empty positions in the packing should be filled with zeros. This is done inside the loops depending on the precision configuration used for the operations. Two indexes are used for managing the buffers depending on the configuration,

one of them serves as an address for the 4-bit buffers (*idx2*) and the other makes the same for the PLM (*idx*).

**Listing 4.21:** Input data packing

```
for (uint16 h = 0; h < N_H_IN_MAX; h++) {
    for (uint16 w = 0; w < N_W_IN_MAX; w++) {
        for (uint16 ci = 0; ci < N_C_MAX; ci ++) {
            uint16 ci_temp = ci*incr1;
            uint16 idx = (MAX_INPUT_CHANNELS * (MAX_INPUT_WIDTH * h + w) +
    ci_temp).to_int();
            uint16 idx2 = (MAX_INPUT_CHANNELS * (MAX_INPUT_WIDTH * h + w) + ci
    ).to_int();
            if (CONFIG1 == 1) { // 4x
                if (ci == in_ch_temp -1 && n_c % 4 == 1 ){
                    A_reconf_HH [idx2] = (int4) 0;
                    A_reconf_HL [idx2] = (int4) 0;
                    A_reconf_LH [idx2] = (int4) 0;
                    A_reconf_LL [idx2] = (int4) plm_in.data[idx].slc<4>(0);
                } else if (ci == in_ch_temp -1 && n_c % 4 == 2 ) {
                    A_reconf_HH [idx2] = (int4) 0;
                    A_reconf_HL [idx2] = (int4) 0;
                    A_reconf_LH [idx2] = (int4) plm_in.data[idx + 1].slc<4>(0)
    ;
                    A_reconf_LL [idx2] = (int4) plm_in.data[idx].slc<4>(0);
                } else if (ci == in_ch_temp -1 && n_c % 4 == 3 ) {
                    A_reconf_HH [idx2] = (int4) 0;
                    A_reconf_HL [idx2] = (int4) plm_in.data[idx + 2 ].slc
    <4>(0);
                    A_reconf_LH [idx2] = (int4) plm_in.data[idx + 1 ].slc
    <4>(0);
                    A_reconf_LL [idx2] = (int4) plm_in.data[idx].slc<4>(0);
                } else {
                    A_reconf_HH [idx2] = (int4) plm_in.data[idx + 3].slc<4>(0)
    ;
                    A_reconf_HL [idx2] = (int4) plm_in.data[idx + 2].slc<4>(0)
    ;
                    A_reconf_LH [idx2] = (int4) plm_in.data[idx + 1].slc<4>(0)
    ;
                    A_reconf_LL [idx2] = (int4) plm_in.data[idx].slc<4>(0);
                }
            } else if (CONFIG1 == 2 || CONFIG1 == 3) {
                if (ci == in_ch_temp -1 && n_c % 2 == 1 ){
                    A_reconf_HH [idx2] = (int4) 0;
                    A_reconf_HL [idx2] = (int4) 0;
                    A_reconf_LH [idx2] = (int4) plm_in.data[idx].slc<4>(4);
                    A_reconf_LL [idx2] = (int4) plm_in.data[idx].slc<4>(0);
                } else {
                    A_reconf_HH [idx2] = (int4) plm_in.data[idx + 1].slc<4>(4)
    ;
                    A_reconf_HL [idx2] = (int4) plm_in.data[idx + 1].slc<4>(0)
    ;
                    A_reconf_LH [idx2] = (int4) plm_in.data[idx].slc<4>(4);
                    A_reconf_LL [idx2] = (int4) plm_in.data[idx].slc<4>(0);
                }
            } else {
```

```
42                 A_reconf_HH  [idx2] = (int4) plm_in.data[idx].slc<4>(12);
43                 A_reconf_HL  [idx2] = (int4) plm_in.data[idx].slc<4>(8);
44                 A_reconf_LH  [idx2] = (int4) plm_in.data[idx].slc<4>(4);
45                 A_reconf_LL  [idx2] = (int4) plm_in.data[idx].slc<4>(0);
46             }
47             if (ci == in_ch_temp -1) break;
48         }
49         if (w == n_w_in - 1) break;
50     }
51     if (h ==  n_h_in - 1) break;
52 }
```

The packing for the weight data is similar to the previous case, but there is a difference in the order the data are organized inside the four groups. The data belonging to the lowest input channel in the packing goes to the higher positions in the operand, i.e. it goes to the *B_reconf_HH* group. This is done in this way due to the organization of the data in the ST multipliers inside the PEs.

**Listing 4.22:** Weight data packing

```
1  for (uint8 i = 0; i< KERN_MAX; i ++){
2      for (uint8 j = 0; j< KERN_MAX; j ++){
3          for (uint16 ci = 0; ci < N_C_MAX; ci ++) {
4              for (uint8 co = 0; co < FILT_MAX; co++) {
5                  uint16 ci_temp = ci*incr1;
6                  uint16 idx = (MAX_OUTPUT_CHANNELS * (MAX_INPUT_CHANNELS * (
    KERN_MAX * i + j) + ci_temp) + co).to_int();
7                  uint16 idx2 = (MAX_OUTPUT_CHANNELS * (MAX_INPUT_CHANNELS * (
    KERN_MAX * i + j) + ci) + co).to_int();
8                  if (CONFIG1 == 1) { // 4x
9                      if (ci == in_ch_temp -1 && n_c % 4 == 1 ){
10                         B_reconf_HH [idx2] = (int4) plm_f.data[idx].slc<4>(0);
11                         B_reconf_HL [idx2] = (int4) 0;
12                         B_reconf_LH [idx2] = (int4) 0;
13                         B_reconf_LL [idx2] = (int4) 0;
14                     } else if (ci == in_ch_temp -1 && n_c % 4 == 2 ) {
15                         B_reconf_HH [idx2] = (int4) plm_f.data[idx].slc<4>(0);
16                         B_reconf_HL [idx2] = (int4) plm_f.data[idx +
    MAX_OUTPUT_CHANNELS ].slc<4>(0);
17                         B_reconf_LH [idx2] = (int4) 0;
18                         B_reconf_LL [idx2] = (int4) 0;
19                     } else if (ci == in_ch_temp -1 && n_c % 4 == 3 ) {
20                         B_reconf_HH [idx2] = (int4) plm_f.data[idx].slc<4>(0);
21                         B_reconf_HL [idx2] = (int4) plm_f.data[idx +
    MAX_OUTPUT_CHANNELS ].slc<4>(0);
22                         B_reconf_LH [idx2] = (int4) plm_f.data[idx +
    MAX_OUTPUT_CHANNELS *2].slc<4>(0);
23                         B_reconf_LL [idx2] = (int4) 0;
24                     } else {
25                         B_reconf_HH [idx2] = (int4) plm_f.data[idx].slc<4>(0);
26                         B_reconf_HL [idx2] = (int4) plm_f.data[idx +
    MAX_OUTPUT_CHANNELS].slc<4>(0);
```

```
27                          B_reconf_LH [idx2] = (int4) plm_f.data[idx +
    MAX_OUTPUT_CHANNELS*2].slc<4>(0);
28                          B_reconf_LL [idx2] = (int4) plm_f.data[idx +
    MAX_OUTPUT_CHANNELS*3].slc<4>(0);
29                      }
30                  } else if (CONFIG1 == 2 || CONFIG1 == 3) {
31                      if (ci == in_ch_temp -1 && n_c % 2 == 1 ){
32                          B_reconf_HH [idx2] = (int4) plm_f.data[idx].slc<4>(4);
33                          B_reconf_HL [idx2] = (int4) plm_f.data[idx].slc<4>(0);
34                          B_reconf_LH [idx2] = (int4) 0;
35                          B_reconf_LL [idx2] = (int4) 0;
36                      } else {
37                          B_reconf_HH [idx2] = (int4) plm_f.data[idx].slc<4>(4);
38                          B_reconf_HL [idx2] = (int4) plm_f.data[idx].slc<4>(0);
39                          B_reconf_LH [idx2] = (int4) plm_f.data[idx +
    MAX_OUTPUT_CHANNELS].slc<4>(4);
40                          B_reconf_LL [idx2] = (int4) plm_f.data[idx +
    MAX_OUTPUT_CHANNELS].slc<4>(0);
41                      }
42                  } else {
43                      B_reconf_HH [idx2] = (int4) plm_f.data[idx].slc<4>(12);
44                      B_reconf_HL [idx2] = (int4) plm_f.data[idx].slc<4>(8);
45                      B_reconf_LH [idx2] = (int4) plm_f.data[idx].slc<4>(4);
46                      B_reconf_LL [idx2] = (int4) plm_f.data[idx].slc<4>(0);
47                  }
48                  if (co == filt -1) break;
49              }
50              if (ci == in_ch_temp -1) break;
51          }
52          if (j == kern - 1) break;
53      }
54      if (i == kern - 1) break;
55  }
```

At this point all the data is ready for the next phase, the computation phase.

**Computation Phase** This phase is basically a version of the accelerator developed in [16]. For deeper details check the paper to understand the internal composition. The prototype of the functions that implements the accelerator computation is shown as follows:

**Listing 4.23:** Prototype computation phase

```
1 void conv2d_m4_v10_reconf_reducedbitwidth(
2                 int4 INPUT_HH[INPUTS_SIZE_MAX],
3                 int4 INPUT_HL[INPUTS_SIZE_MAX],
4                 int4 INPUT_LH[INPUTS_SIZE_MAX],
5                 int4 INPUT_LL[INPUTS_SIZE_MAX],
6                 int4 WEIGHT_HH[FILTERS_SIZE_MAX],
7                 int4 WEIGHT_HL[FILTERS_SIZE_MAX],
8                 int4 WEIGHT_LH[FILTERS_SIZE_MAX],
9                 int4 WEIGHT_LL[FILTERS_SIZE_MAX],
```

```
10          plm_outputs_t &OUT,
11                  uint8 IN_HEIGHT,
12                  uint8 IN_WIDTH,
13                  uint8 IN_CH,
14                  uint8 K_SIZE,
15                  uint8  STRIDE,
16                  uint8 OUT_HEIGHT,
17                  uint8 OUT_WIDTH,
18                  uint1  RST_OUT_ACC,
19                  uint1  EN_QUANTIZATION,
20                  uint3  CONFIG1,
21                  uint2  CONFIG2,
22                  uint1  EN_RELU,
23                  uint8 OUT_CH,
24                  ac_int<W_CROSS_BITWIDTH, true> WEIGHTS_CROSSPRODUCT[FILT_MAX],
25                  ac_fixed<SF_IN_W_TOT_BITWIDTH, SF_IN_W_INT_BITWIDTH, true>
        SCALING_FACTOR_INPUTS_WEIGHTS[FILT_MAX],
26                  ac_fixed<BIASQ_SCALED_TOT_BITWIDTH, BIASQ_SCALED_INT_BITWIDTH,
         true> BIASQ_SCALED[FILT_MAX],
27                  ac_fixed<SF_OUT_INV_TOT_BITWIDTH, SF_OUT_INV_INT_BITWIDTH,
        true> SCALING_FACTOR_OUT_INVERSE,
28                  ac_int<Z_BITWIDTH, true> Z_O2 );
```

Table 4.5 explains the arguments used in the previous function.

| **Argument** | **Description** |
|---|---|
| `INPUT_HH, INPUT_HL, INPUT_LH, INPUT_LL` | These are the intermediate private local memories which stores the 4-bit values used to store the values that will be assigned to the operand A of the ST multiplier |
| `WEIGHT_HH, WEIGHT_HL, WEIGHT_LH, WEIGHT_LL` | These are the intermediate private local memories which stores the 4-bit values used to store the values that will be assigned to the operand B of the ST multiplier |
| `OUT` | This is the reference to the output PLM |
| `IN_HEIGHT` | The height size of the input tile |
| `IN_WIDTH` | The width size of the input tile |
| `IN_CH` | The input channel size of the input and weight tiles |
| `K_SIZE` | The width and height size of the weight tile |
| `STRIDE` | Number of positions the kernel slides |
| `OUT_HEIGHT` | The height size of the output tile |

| | |
|---|---|
| OUT_WIDTH | The width size of the output tile |
| RST_OUT_ACC | Argument that enables accumulation of partial results, corresponds to *acc_flag* variable |
| EN_QUANTIZATION | Argument that enables quantization of the output results, corresponds to *q_flag* variable |
| CONFIG1 | It specifies the configuration precision of the operations, 4, 8, or 16 bit precision. Consequently it specifies how many input channels values are taken, 1, 2 or 4 values |
| CONFIG2 | It specifies the bitwidth of the final output operation: 4, 8, 16 bit |
| EN_RELU | Argument that enables Relu operation of the output results, corresponds to *relu_flag* variable |
| OUT_CH | The output channel size of the output and weight tiles |
| WEIGHTS_CROSSPRODUCT | Result of a product operation between the quantized weights of a layer and the zero point of the input. Used for quantization and explained in [22] |
| SF_IN_W_INT_BITWIDTH | Result of a product operation between the weight and input scale factors. Used for quantization and explained in [22] |
| BIASQ_SCALED_INT_BITWIDTH | Product of the bias value and its scale factor. Used for quantization and explained in [22] |
| SF_OUT_INV_INT_BITWIDTH | Inverse of the output scale factor. Used for quantization and explained in [22] |
| Z_O2 | Zero point of the output. Used for quantization and explained in [22] |

**Table 4.5:** List of arguments of the function that implements the computation phase

**Store Phase** At this point all the computation values are kept in the output PLM with the addressing followed in the computation phase. This same addressing has to be used to read the values from the output PLM and pass them to the DMA engine, so this last can write the results to the external memory. The output data is passed sequentially to the DMA, hence the loop order determines how the data will be organized into the external memory. Following the memory organization explained at the beginning of this chapter, the outermost loop iterates through the output channels, the next loop iterates through the height dimension and the innermost loop goes across the width dimension. DMA configuration is done in the outermost loop because in each iteration a new transaction is done. The offset given to the DMA configuration is composed by the output tensor pointer passed by the tiling algorithm plus the multiplication of the iteration number in the output channel dimension and the size of one entire output channel in the original tensor, without tiling. Each transaction length size is equal to the multiplication of the height and width of the output tile.

**Listing 4.24:** Store phase

```
1  STORE_CO_LOOP:
2      for (uint16_t co = 0; co < FILT_MAX; co++){
3          uint32_t offset_write = out_add + co*offset_PE_out;
4          dma_write_info = {offset_write, dma_write_data_length, DMA_SIZE};
5          bool dma_write_ctrl_done = false;
6      STORE_CTRL_LOOP:
7          do { dma_write_ctrl_done = dma_write_ctrl.nb_write(dma_write_info); }
       while (!dma_write_ctrl_done);
8          if (dma_write_ctrl_done) { // Force serialization between DMA control
       and DATA data transfer
9  STORE_H_LOOP: for (uint16_t h = 0; h < N_H_OUT_MAX ; h++){
10 STORE_W_LOOP:    for (uint16_t w = 0; w < N_W_OUT_MAX; w++){
11                      uint16_t index = MAX_OUTPUT_CHANNELS * (MAX_OUTPUT_WIDTH *
        h + w) + co;
12                      FPDATA_OUT data = plm_out.data[index];
13                      assert(DMA_WIDTH == 64 && "DMA_WIDTH should be 64 (
       simplicity choice)");
14                      ac_int<DMA_WIDTH, false> data_ac;
15                      ac_int<32, false> DEADBEEF = 0xdeadbeef;
16                  data_ac.set_slc(32, DEADBEEF.template slc<32>(0));
17                      data_ac.set_slc(0, data.template slc<DATA_WIDTH>(0));
18                      dma_write_chnl.write(data_ac);
19                      if (w == n_w_out -1) break;
20                  }
21                  if (h == n_h_out -1) break;
22              }
23          }
24          if (co == filt -1 ) break;
25      }
```

### 4.3.2 Depthwise Convolution accelerator

The DW-Conv accelerator has the same structure and phases as the 2D-Conv accelerator. The main difference is the computation phase which in this case performs a different kind of convolution. Since the computation of this convolution does not consider the output channel dimension, all the parameters related to this dimension are omitted in this case. Other important differences are explained as follows:

- Due to the way how the weight tensor is tiled in this algorithm, all the weight data of the tiles are located together in external memory. Therefore, only one DMA transaction is required for reading these values.

**Listing 4.25:** Weight load in DW-Conv

```
1           dma_read_info = {w_add, dma_read_w_data_length, DMA_SIZE};
2   bool dma_read_ctrl_done2 = false;
3           LOAD_CTRL_LOOP2:
4           do { dma_read_ctrl_done2 = dma_read_ctrl.nb_write(dma_read_info);
      } while (!dma_read_ctrl_done2);
5         if (dma_read_ctrl_done2) {
6               weight_load_for:
7               for (uint16_t ci = 0; ci < N_C_MAX; ci++){
8                   for (uint16_t j = 0; j < KERN_MAX; j++){
9                       for (uint16_t i = 0; i < KERN_MAX; i++) {
10                          uint16_t index = MAX_INPUT_CHANNELS * (KERN_MAX *
      j + i) + ci;
11                         #ifndef __SYNTHESIS__
12                              while (!dma_read_chnl.available(1)) {}; //
     Hardware stalls until data ready
13                         #endif
14                              ac_int<DATA_WIDTH, false> data_ac =
     dma_read_chnl.read().template slc<DATA_WIDTH>(0);
15                              FPDATA_IN data;
16                              data.set_slc(0, data_ac);
17                              plm_f.data[index] = data;
18                         if (i == kern - 1) break;
19                      }
20                      if (j == kern - 1) break;
21                  }
22                  if (ci == n_c - 1) break;
23              }
24        }
25
```

- The packaging done in the load phase arranges the data in the four buffer groups using the same index. Hence a position in this buffers only contains one value no matter the precision configuration used. The values are group together in the computation phase to form the respective operands for the ST multiplier.

**Listing 4.26:** Data packaging in DW-Conv

```
for (uint16 h = 0; h < N_H_IN_MAX; h++) {
    for (uint16 w = 0; w < N_W_IN_MAX; w++) {
        for (uint16 ci = 0; ci < N_C_MAX; ci ++) {
            uint16 idx = (MAX_INPUT_CHANNELS * (MAX_INPUT_WIDTH * h + w)
    + ci).to_int();
            if (CONFIG1 == 1) { // 4x
                a_LL_4b = (int4) plm_in.data[idx].slc<4>(4);
                a_HH_4b = a_LL_4b >> 4;
                a_HL_4b = a_LL_4b >> 4;
                a_LH_4b = a_LL_4b >> 4;
            } else if (CONFIG1 == 2 || CONFIG1 == 3) {
                a_LH_4b = (int4) plm_in.data[idx].slc<4>(4);
                a_LL_4b = (int4) plm_in.data[idx].slc<4>(0);
                a_HH_4b = a_LH_4b >> 4;
                a_HL_4b = a_LH_4b >> 4;
            } else {
                a_HH_4b = (int4) plm_in.data[idx].slc<4>(12);
                a_HL_4b = (int4) plm_in.data[idx].slc<4>(8);
                a_LH_4b = (int4) plm_in.data[idx].slc<4>(4);
                a_LL_4b = (int4) plm_in.data[idx].slc<4>(0);
            }
            A_reconf_HH [idx] = a_HH_4b;
            A_reconf_HL [idx] = a_HL_4b;
            A_reconf_LH [idx] = a_LH_4b;
            A_reconf_LL [idx] = a_LL_4b;
            if (ci == n_c -1) break;
        }
        if (w == n_w_in - 1) break;
    }
    if (h ==  n_h_in - 1) break;
}

```

- The DW-Conv does not need the accumulation of partial results, hence the accumulation flag and its implementation is omitted in this case.
- Each processing element works with values taken from one input channel, which can be computed independently.

### 4.3.3 Fully-connected accelerator

The FC accelerator also follows the same architecture of the previous accelerators. However, it presents less parameters and loops because the tensors used in this cases have less dimensions. The main differences are explained as follows:

- The input values are read using a single loop iteration because the input activations have just one dimension.

- The weight values are read using two loop iterations and an auxiliary offset, *offset_PE*, is also used to read inter-spaced data.

**Listing 4.27:** Weight load in FC accelerator

```
LOOP_LOAD_WEIGHTS:
    for (uint16_t mi = 0; mi < MAX_OUTPUT_NEURONS; mi++){
        uint16_t offset_weight_M = w_add + mi*offset_PE;
        dma_read_info = {offset_weight_M, dma_read_w_data_length,
    DMA_SIZE};
  bool dma_read_ctrl_done2 = false;
        LOAD_CTRL_LOOP2:
        do { dma_read_ctrl_done2 = dma_read_ctrl.nb_write(dma_read_info);
     } while (!dma_read_ctrl_done2);
        if (dma_read_ctrl_done2) {
            weight_load_for:
            for (uint16_t ni = 0; ni < MAX_INPUT_ACTIVATIONS; ni++) {
                uint16_t index = MAX_OUTPUT_NEURONS * ni + mi;
            #ifndef __SYNTHESIS__
                while (!dma_read_chnl.available(1)) {}; // Hardware
    stalls until data ready
            #endif
                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[index] = data;
                if (ni == N - 1) break;
            }
        }
        if (mi == M - 1) break;
    }

```

- The architecture of this accelerator considers two ST multipliers for each processing element [16]. This decision implies that the packaging should consider twice the values compared to the previous cases. Therefore, there are 8 different intermediate buffers which store parts of two different operands intended for the two ST multipliers. The packaging part has to fill the empty spaces in case no available data is present for the feeding the multipliers.

**Listing 4.28:** Packaging data in FC accelerator

```
for (uint16 c = 0; c < MAX_INPUT_ACTIVATIONS; c++) {
    uint16 c_temp = c*incr1_reconf;
    if (CONFIG1 == 1) { // 4x
        if (c == new_N -1 && N % 8 == 1 ){
            a1_HH_4b = (int4) plm_in.data[(c_temp+3).to_int()].slc<4>(0);
            a1_HL_4b = (int4) plm_in.data[(c_temp+2).to_int()].slc<4>(0);
            a1_LH_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(0);
            a1_LL_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(0);
```

```
9              a2_HH_4b = (int4) 0;
10             a2_HL_4b = (int4) plm_in.data[(c_temp+6).to_int()].slc<4>(0);
11             a2_LH_4b = (int4) plm_in.data[(c_temp+5).to_int()].slc<4>(0);
12             a2_LL_4b = (int4) plm_in.data[(c_temp+4).to_int()].slc<4>(0);
13         } else if (c == new_N -1 && N % 8 == 2 ){
14             a1_HH_4b = (int4) plm_in.data[(c_temp+3).to_int()].slc<4>(0);
15             a1_HL_4b = (int4) plm_in.data[(c_temp+2).to_int()].slc<4>(0);
16             a1_LH_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(0);
17             a1_LL_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(0);
18             a2_HH_4b = (int4) 0;
19             a2_HL_4b = (int4) 0;
20             a2_LH_4b = (int4) plm_in.data[(c_temp+5).to_int()].slc<4>(0);
21             a2_LL_4b = (int4) plm_in.data[(c_temp+4).to_int()].slc<4>(0);
22         } else if (c == new_N -1 && N % 8 == 3 ){
23             a1_HH_4b = (int4) plm_in.data[(c_temp+3).to_int()].slc<4>(0);
24             a1_HL_4b = (int4) plm_in.data[(c_temp+2).to_int()].slc<4>(0);
25             a1_LH_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(0);
26             a1_LL_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(0);
27             a2_HH_4b = (int4) 0;
28             a2_HL_4b = (int4) 0;
29             a2_LH_4b = (int4) 0;
30             a2_LL_4b = (int4) plm_in.data[(c_temp+4).to_int()].slc<4>(0);
31         } else if (c == new_N -1 && N % 8 == 4 ){
32             a1_HH_4b = (int4) plm_in.data[(c_temp+3).to_int()].slc<4>(0);
33             a1_HL_4b = (int4) plm_in.data[(c_temp+2).to_int()].slc<4>(0);
34             a1_LH_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(0);
35             a1_LL_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(0);
36             a2_HH_4b = (int4) 0;
37             a2_HL_4b = (int4) 0;
38             a2_LH_4b = (int4) 0;
39             a2_LL_4b = (int4) 0;
40         } else if (c == new_N -1 && N % 8 == 5 ){
41             a1_HH_4b = (int4) 0;
42             a1_HL_4b = (int4) plm_in.data[(c_temp+2).to_int()].slc<4>(0);
43             a1_LH_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(0);
44             a1_LL_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(0);
45             a2_HH_4b = (int4) 0;
46             a2_HL_4b = (int4) 0;
47             a2_LH_4b = (int4) 0;
48             a2_LL_4b = (int4) 0;
49         } else if (c == new_N -1 && N % 8 == 6 ){
50             a1_HH_4b = (int4) 0;
51             a1_HL_4b = (int4) 0;
52             a1_LH_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(0);
53             a1_LL_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(0);
54             a2_HH_4b = (int4) 0;
55             a2_HL_4b = (int4) 0;
56             a2_LH_4b = (int4) 0;
57             a2_LL_4b = (int4) 0;
58         } else if (c == new_N -1 && N % 8 == 7 ){
59             a1_HH_4b = (int4) 0;
60             a1_HL_4b = (int4) 0;
61             a1_LH_4b = (int4) 0;
62             a1_LL_4b = (int4) plm_in.data[(c_temp).to_int()].slc<4>(0);
63             a2_HH_4b = (int4) 0;
64             a2_HL_4b = (int4) 0;
```

```
65                  a2_LH_4b = (int4) 0;
66                  a2_LL_4b = (int4) 0;
67              } else{
68                  a1_HH_4b = (int4) plm_in.data[(c_temp+3).to_int()].slc<4>(0);
69                  a1_HL_4b = (int4) plm_in.data[(c_temp+2).to_int()].slc<4>(0);
70                  a1_LH_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(0);
71                  a1_LL_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(0);
72                  a2_HH_4b = (int4) plm_in.data[(c_temp+7).to_int()].slc<4>(0);
73                  a2_HL_4b = (int4) plm_in.data[(c_temp+6).to_int()].slc<4>(0);
74                  a2_LH_4b = (int4) plm_in.data[(c_temp+5).to_int()].slc<4>(0);
75                  a2_LL_4b = (int4) plm_in.data[(c_temp+4).to_int()].slc<4>(0);
76              }
77          } else if (CONFIG1 == 2 || CONFIG1 == 3) { // 8x
78              if (c == new_N -1 && N % 4 == 1 ){
79                  a1_HH_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(4);
80                  a1_HL_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(0);
81                  a1_LH_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(4);
82                  a1_LL_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(0);
83                  a2_HH_4b = (int4) 0;
84                  a2_HL_4b = (int4) 0;
85                  a2_LH_4b = (int4) plm_in.data[(c_temp+2).to_int()].slc<4>(4);
86                  a2_LL_4b = (int4) plm_in.data[(c_temp+2).to_int()].slc<4>(0);
87              } else if (c == new_N -1 && N % 4 == 2 ){
88                  a1_HH_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(4);
89                  a1_HL_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(0);
90                  a1_LH_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(4);
91                  a1_LL_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(0);
92                  a2_HH_4b = (int4) 0;
93                  a2_HL_4b = (int4) 0;
94                  a2_LH_4b = (int4) 0;
95                  a2_LL_4b = (int4) 0;
96              } else if (c == new_N -1 && N % 4 == 3 ){
97                  a1_HH_4b = (int4) 0;
98                  a1_HL_4b = (int4) 0;
99                  a1_LH_4b = (int4) plm_in.data[(c_temp).to_int()].slc<4>(4);
100                 a1_LL_4b = (int4) plm_in.data[(c_temp).to_int()].slc<4>(0);
101                 a2_HH_4b = (int4) 0;
102                 a2_HL_4b = (int4) 0;
103                 a2_LH_4b = (int4) 0;
104                 a2_LL_4b = (int4) 0;
105             } else{
106                 a1_HH_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(4);
107                 a1_HL_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(0);
108                 a1_LH_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(4);
109                 a1_LL_4b = (int4) plm_in.data[(c_temp  ).to_int()].slc<4>(0);
110                 a2_HH_4b = (int4) plm_in.data[(c_temp+3).to_int()].slc<4>(4);
111                 a2_HL_4b = (int4) plm_in.data[(c_temp+3).to_int()].slc<4>(0);
112                 a2_LH_4b = (int4) plm_in.data[(c_temp+2).to_int()].slc<4>(4);
113                 a2_LL_4b = (int4) plm_in.data[(c_temp+2).to_int()].slc<4>(0);
114             }
115         } else { // if (CONFIG == 0 || CONFIG == 4) { // 16x
116             if (c == new_N -1 && N % 2 == 1 ){
117                 a1_HH_4b = (int4) plm_in.data[(c_temp).to_int()].slc<4>(12);
118                 a1_HL_4b = (int4) plm_in.data[(c_temp).to_int()].slc<4>(8);
119                 a1_LH_4b = (int4) plm_in.data[(c_temp).to_int()].slc<4>(4);
120                 a1_LL_4b = (int4) plm_in.data[(c_temp).to_int()].slc<4>(0);
```

```
121             a2_HH_4b = (int4) 0;
122             a2_HL_4b = (int4) 0;
123             a2_LH_4b = (int4) 0;
124             a2_LL_4b = (int4) 0;
125         } else {
126             a1_HH_4b = (int4) plm_in.data[(c_temp).to_int()].slc<4>(12);
127             a1_HL_4b = (int4) plm_in.data[(c_temp).to_int()].slc<4>(8);
128             a1_LH_4b = (int4) plm_in.data[(c_temp).to_int()].slc<4>(4);
129             a1_LL_4b = (int4) plm_in.data[(c_temp).to_int()].slc<4>(0);
130             a2_HH_4b =(int4) plm_in.data[(c_temp+1).to_int()].slc<4>(12);
131             a2_HL_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(8);
132             a2_LH_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(4);
133             a2_LL_4b = (int4) plm_in.data[(c_temp+1).to_int()].slc<4>(0);
134         }
135     }
136     A1_reconf_HH[c] = a1_HH_4b;
137     A1_reconf_HL[c] = a1_HL_4b;
138     A1_reconf_LH[c] = a1_LH_4b;
139     A1_reconf_LL[c] = a1_LL_4b;
140     A2_reconf_HH[c] = a2_HH_4b;
141     A2_reconf_HL[c] = a2_HL_4b;
142     A2_reconf_LH[c] = a2_LH_4b;
143     A2_reconf_LL[c] = a2_LL_4b;
144     if (c == new_N -1) break;
145 } // c
146
```

- When the input activations are tiled, this accelerator also performs the accumulation of partial results to calculate the output activations.
- Each processing element works on a different output activation.
- The store phase only has one loop iteration because the output has only one dimension.

# Chapter 5

# Simulation, FPGA implementation and Results

In this chapter the correctness of the tiling algorithm presented above will be tested thought C and RTL simulations, and then further evaluated on a FPGA exploiting the ESP framework. The basic steps to set up all the framework and environment variables, in order to reproduce the tests just mentioned, are presented in these thesis works [19], [20]. They can be taken as a reference guide to replicate the workflow from scratch. Accordingly, the following chapter will focus on explaining the differences in the workflow and some considerations that have been taken to validate the tiling architectures. The test results confirms the correct behaviour of the tiling algorithm and highlights the conditions in which the tiling architecture performs at its best. It also compares the performance using the hardware accelerators with the tiling algorithm and the performance of the RISC-V processor to compute convolution operations.

## 5.1 C/C++ simulation

The first step to validate the correct operation of the tiling algorithm of each accelerator is to perform a C simulation by considering the accelerators isolated from the rest of the SoC. Hence, the testbench has to simulate the operations of the DMA and the processor, i.e. set the tile parameters and

configuration for the computation of the convolution, feed the DMA channels with random data to be used by the accelerator as follows:

**Listing 5.1:** Input and weight data transfer to accelerator in TB

```
// Pass inputs to the accelerator
    for (unsigned i = 0; i < input_size; i++) {
  //generate random input
        FPDATA_IN data_fp = (i % 200) * 0.25 - 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);
    }
    for (unsigned i = 0; i < weight_size; i++) {
        FPDATA_IN data_fp = (i % 200) * 0.25 - 25;
        weights[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, weights[i].template slc<DATA_WIDTH>(0));
        dma_read_chnl.write(data_ac);
    }
```

After passing the data, the accelerator is called and performs the convolution. Then, the output data is fetched from the accelerator's output memory and compared to the golden results computed by the testbench code. Figure 5.1 shows the test's results considering a tile with the following dimensions and parameters:

- Width = 7
- Height = 7
- Input channels = 7
- Kernel width and height = 3
- Ouptut channels = 10
- Stride = 1
- Paddding = valid

```
# ============================================
# Simulating design
# cd ../..; ./conv2d_cxx_catapult_basic_fx32_dma64/conv2d_cxx_catapult.v1/scverify/orig_cxx_osci/scverify_top
# Info: main(): --------------------------------
# Info: main(): ESP - Conv2D [Catapult HLS C++]
# Info: main():       Single block
# Info: main(): --------------------------------
# Info: main(): Configuration:
# Info: main():   - n_w: 7
# Info: main():   - n_h: 7
# Info: main():   - n_c: 7
# Info: main():   - kern: 3
# Info: main():   - filt: 10
# Info: main():   - stride: 1
# Info: main():   - acc_flag: 0
# Info: main(): Other info:
# Info: main():   - DMA width: 64
# Info: main():   - DMA size [2 = 32b, 3 = 64b]: 3
# Info: main():   - DATA width: 32
# Info: main():   - Input and Weight size: 973
# Info: main():   - memory in (words): 343
# Info: main():   - memory out (words): 250
# Info: main(): --------------------------------
# Info: main(): Validation: PASS
# Info: main():   - errors 0 / total 250
# Info: main(): --------------------------------
# make[2]: Leaving directory `/home/bueno3/esp/accelerators/catapult_hls/conv2d_cxx_catapult/hw/hls-work-virte
x7/conv2d_cxx_catapult_basic_fx32_dma64/conv2d_cxx_catapult.v1'
# Saving project file '/home/bueno3/esp/accelerators/catapult_hls/conv2d_cxx_catapult/hw/hls-work-virtex7/conv
2d_cxx_catapult_basic_fx32_dma64.ccs'. (PRJ-5)
# ***************************************************************
# uArch: Single block
# ***************************************************************
# Done!
```

**Figure 5.1:** C Testbench simulation result of a 2D Convolution operation

## 5.2 High Level Synthesis and RTL simulation

At this point the behaviour of the accelerator is checked at higher level of abstraction. Now the high-level synthesis has to be performed to obtain the RTL code and to integrate the accelerator into the SoC. The HLS tool used in this work is Siemens Catapult HLS.
The synthesis workflow is similar to the previous works [19], [20], with the difference that in this architecture an accelerator with multiple PEs has to be synthesized, therefore loop unrolling directives need to be added to the synthesis script. The adoption of multiple PEs requires to access more information from the PLMs at the same time. Since the memories have only one read port and one write port, the interleave directive is needed to infer multiple smaller memory banks from the original PLMs for those that require multiple data accesses in parallel. This directive specifies that the data should be organized or accessed in an interleaved manner. This means that the tool will attempt to interleave the memory accesses for the specified variables, enabling concurrent or grouped access to multiple memory locations. Furthermore, the largest arrays of the accelerator's architecture

are mapped to memories and others are mapped to registers. The following directives are added to the base synthesis script of [19]:

**Listing 5.2:** Additional directives for systhesis script [19]

```
1  directive set /$ACCELERATOR/core/plm_in.data:rsc -MAP_TO_MODULE
       Xilinx_RAMS.BLOCK_1R1W_RBW
2  directive set /$ACCELERATOR/core/plm_f.data:rsc -MAP_TO_MODULE
       Xilinx_RAMS.BLOCK_1R1W_RBW
3  directive set /$ACCELERATOR/core/plm_out.data:rsc -MAP_TO_MODULE
       Xilinx_RAMS.BLOCK_1R1W_RBW
4  directive set /$ACCELERATOR/core/plm_out.data:rsc -INTERLEAVE
       $MAX_OUTPUT_CHANNELS
5  directive set /$ACCELERATOR/core/buf_acc.data:rsc -INTERLEAVE
       $MAX_OUTPUT_CHANNELS
6  directive set /$ACCELERATOR/core/B_reconf_HH:rsc -INTERLEAVE
       $MAX_OUTPUT_CHANNELS
7  directive set /$ACCELERATOR/core/B_reconf_HL:rsc -INTERLEAVE
       $MAX_OUTPUT_CHANNELS
8  directive set /$ACCELERATOR/core/B_reconf_LH:rsc -INTERLEAVE
       $MAX_OUTPUT_CHANNELS
9  directive set /$ACCELERATOR/core/B_reconf_LL:rsc -INTERLEAVE
       $MAX_OUTPUT_CHANNELS
10 directive set /$ACCELERATOR/core/output_acc:rsc -MAP_TO_MODULE {[Register]}
11 directive set /$ACCELERATOR/core/outputq_quantized_4x:rsc  -MAP_TO_MODULE {[
       Register]}
12 directive set /$ACCELERATOR/core/outputq_quantized_8x:rsc  -MAP_TO_MODULE {[
       Register]}
13 directive set /$ACCELERATOR/core/outputq_quantized_16x:rsc -MAP_TO_MODULE {[
       Register]}
14 directive set /$ACCELERATOR/core/in_h_for  -PIPELINE_INIT_INTERVAL 1
15 directive set /$ACCELERATOR/core/in_w_for  -PIPELINE_INIT_INTERVAL 1
16 directive set /$ACCELERATOR/core/k_h_for   -PIPELINE_INIT_INTERVAL 1
17 directive set /$ACCELERATOR/core/k_w_for   -PIPELINE_INIT_INTERVAL 1
18 directive set /$ACCELERATOR/core/ci_for    -PIPELINE_INIT_INTERVAL 1
19 directive set /$ACCELERATOR/core/co_for    -UNROLL yes
```

Figure 5.2 shows the list of loops considered in the accelerator, the loop where loop unrolling was applied (*co_for, wb_for, q_for*) to enable 16 PEs and the rest of them which pipelining was applied. Figure 5.3 shows the scheduled operations of all the loops that compose the 2D convolution accelerator.

After performing the synthesis, the tool gives different results from the expected ones with respect to the number of multipliers synthesised in the accelerator. As it can be deduced, if there are 16 PEs, 16 are also the number of multipliers expected after the synthesis. However, when the target technology is FPGA-oriented, the ST multipliers are synthesized with multiple multipliers instead of a single instance, as in the FPGA fabric the types of resources are limited. Therefore, the synthesis tool, which in this work is Xilinx Vivado, decided to emulate the behaviour of the ST multiplier

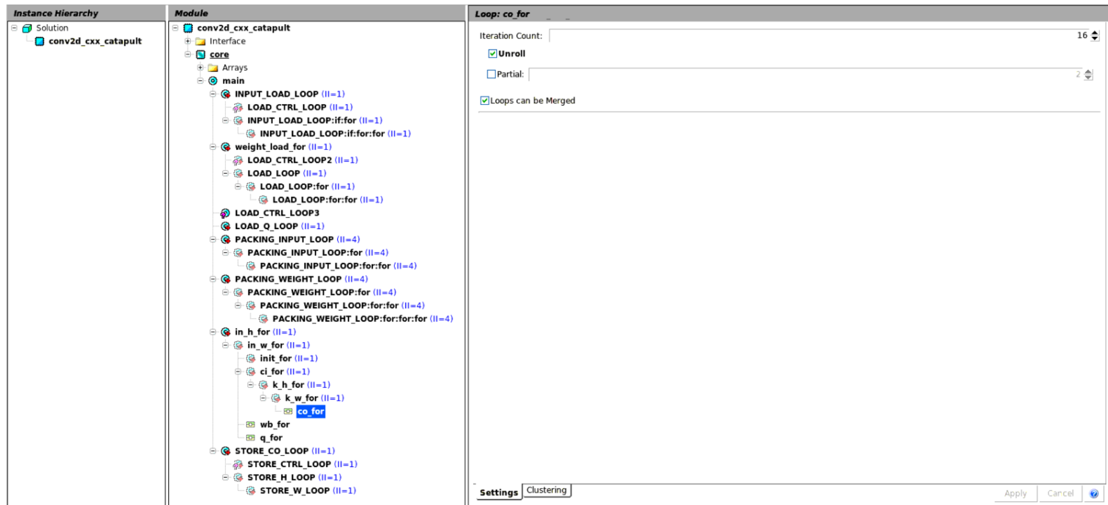


**Figure 5.2:** Loop list and configurations

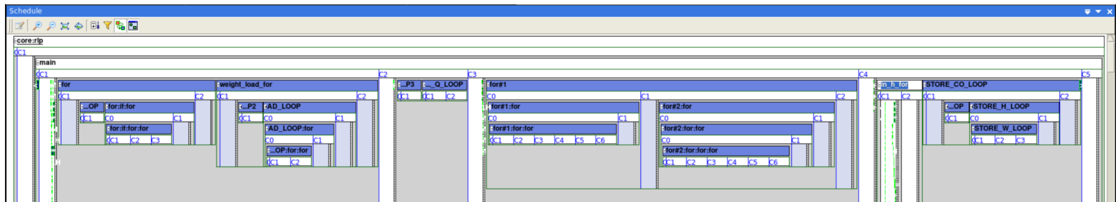


**Figure 5.3:** 2D-Conv loop's schedule

with extra multipliers of different operators bitwidths (8, 16 bits). Hence, instead of having 16 ST multipliers, the final design has 64 multipliers that together behave as 16 ST ones. Figure 5.4 shows that more multipliers are synthesised when the target is a FPGA technology. On the other hand, Figure 5.5 shows that exactly 16 multipliers are synthesised when the target technology is ASIC oriented. This experiment double-checked that the unrolling and memory partitioning was performed correctly by the HLS tool.

The next step is to simulate the operation of all the SoC in RTL. Thanks to the baremetal it is possible to simulate how the processor can handle the tiling algorithm and subsequently call the accelerator to process each tile in a specific order. Figure 5.6 shows the accelerator's waveforms captured from the simulation tool. When it is high, the signal *acc_done* tells that the accelerator has finished a tile computation, therefore four tiles are computed in Figure 5.6. Also in this case the convolution results are compared to the golden results, computed in software considering the entire tensors and the

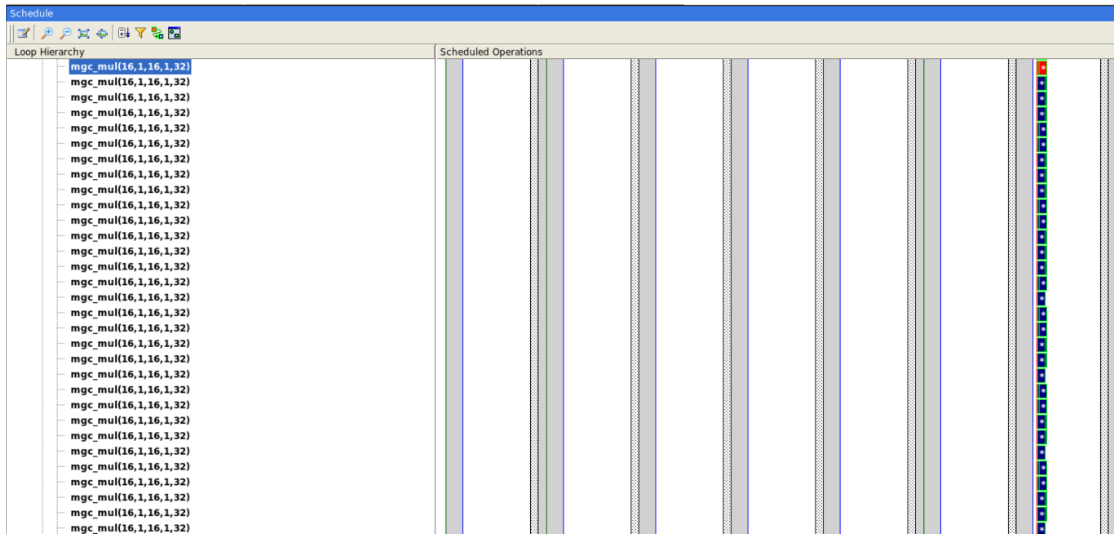


**Figure 5.4:** Multipliers synthesised for FPGA technology

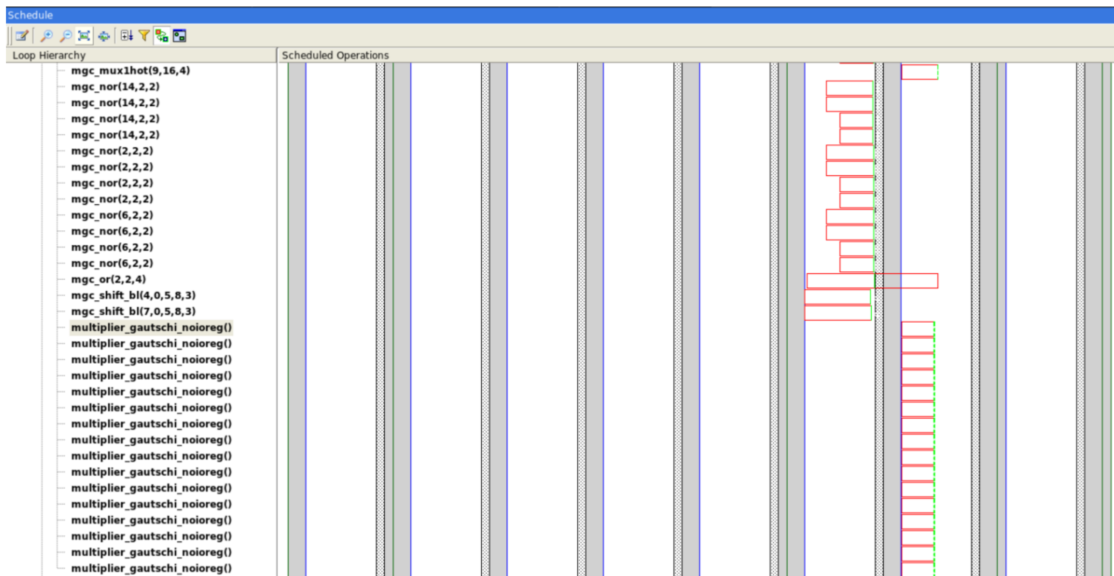


**Figure 5.5:** Multipliers synthesised for ASIC technology

results match.

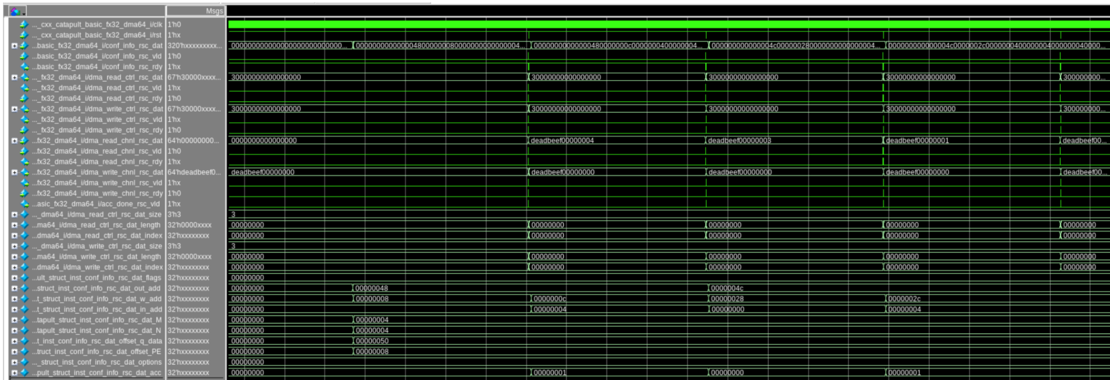

**Figure 5.6:** RTL Simulation of Tiled convolution

## 5.3 FPGA Prototyping and validation results

Finally, the tiling architecture is tested on the FPGA, in this case the board *ProFPGA XC7V2000T*. The RTL code is synthesised considering the FPGA resources and constraints, then the generated bitstream is upload to the FPGA to program it. After this process, the FPGA is ready to execute the baremetal code that implements the tiling algorithm. As expected, the tests performed in the FPGA are orders of magnitude faster than the RTL simulation, hence larger tensors can be used when testing the tiling process. To validate the results and correct behaviour of the system, Tensorflow framework is used to obtain the golden values of a convolution layer. The code developed in previous works used Tensorflow to generate the results of a convolution operation using quantization as well. This code also provides the parameters needed to apply quantization to the final results, all the values are written in text files. However, the tensor values (input, weight, output) have a different organization and order inside the text files with respect to the order consider for the external memory organization and explained in the chapter 4. Therefore, a reorder of the data is needed to correctly feed the accelerator with the input and weight values. The golden results are reordered as well to compare them against the system's ones. The following pyhton script was developed to take the values and parameters from the text files, reorder the data and write them into arrays inside header files that will be imported by the baremetal C code. The script shows the dimensions of the tensor used in the test as well.

**Listing 5.3:** Data reorder and headers generator script

```python
import struct
import numpy as np
Win = 8
Hin = 8
Hout = 6
Wout = 6
Cin = 16
Cout = 4
ker = 3
def read_file_write_header (input_file, header_file,array_name, array_type,
    reorder, input_file2=''):
    # Read numbers from the input file
    with open(input_file, "r") as file:
        lines = file.read().splitlines()
    if (reorder == 2):
        with open(input_file2, "r") as file2:
            lines2 = file2.read().splitlines()
    def_name = str(array_name).upper()
    # Create the C header file
    with open(header_file, "w") as header:
        header.write("#ifndef "+def_name+"_H\n")
        header.write("#define "+def_name+"_H\n\n")
        header.write(f"#define {array_name}_size {len(lines)}\n")
        header.write(f"{array_type} {array_name}[{array_name}_size] = {{\n")
        if (reorder==1):
            temp = []
            temp_ordered = []
            for i,line in enumerate(lines):
                temp.append(float(line))
            if (array_name == 'input'):
                channel1 = Cin
                channel2 = 1
                Height = Hin
                Width = Win
            elif((array_name == 'output')):
                channel1 = Cout
                channel2 = 1
                Height = Hout
                Width = Wout
            elif((array_name == 'weight')):
                channel1 = Cin
                channel2 = Cout
                Height = ker
                Width = ker
            for c2 in range (channel2):
                for c1 in range (channel1):
                    for h in range (Height):
                        for w in range (Width):
                            temp_ordered.append(temp [channel2*(channel1*(
    Width*h + w)+c1)+c2])
            for i in range(len(temp_ordered)):
                header.write(f"    {temp_ordered[i]}")
                if i < len(temp_ordered) - 1:
                    header.write(",")
                header.write("\n")
            header.write("};\n\n")
```

```
55              header.write("#endif\n")
56          elif (reorder == 2):
57              temp1 = []
58              temp2 = []
59              for i,line in enumerate(lines):
60                  temp1.append((np.float128(line)))
61              for i,line in enumerate(lines2):
62                  temp2.append((np.float128(line)))
63              for i in range(len(temp1)):
64                  header.write(f"    {str(temp1[i]*temp2[0])}")
65                  if i < len(lines) - 1:
66                      header.write(",")
67                  header.write("\n")
68              header.write("};\n\n")
69              header.write("#endif\n")
70          else:
71              for i, line in enumerate(lines):
72                  if (reorder == 3):
73                      header.write(f"    {1/float(line)}")
74                  else:
75                      header.write(f"    {float(line)}")
76                  if i < len(lines) - 1:
77                      header.write(",")
78                  header.write("\n")
79              header.write("};\n\n")
80              header.write("#endif\n")
81      print(f"Generated {header_file} with {len(lines)} elements.")
82  read_file_write_header("qconv2d/0_ofq_in_out.txt","input.h", "input", "double"
        , 1)
83  read_file_write_header("qconv2d/0_wq.txt","weight.h", "weight", "double" , 1)
84  read_file_write_header("qconv2d/0_ofq_out.txt","output.h", "output", "double"
        ,1 )
85  read_file_write_header("qconv2d/0_subq1.txt","w_cross.h", "w_cross", "double",
         0 )
86  read_file_write_header("qconv2d/0_bq.txt","biasq_scaled.h", "biasq_scaled", "
        double", 2 , "qconv2d/0_b_s.txt")
87  read_file_write_header("qconv2d/0_w_s.txt","scaling_factor_iw.h", "
        scaling_factor_iw", "double", 2 , "qconv2d/0_of_s_in.txt")
88  read_file_write_header("qconv2d/0_of_z_in.txt","z_o.h", "z_o", "double", 0 )
89  read_file_write_header("qconv2d/0_of_s_out.txt","scaling_factor_oi.h", "
        scaling_factor_oi", "double", 3 )
```

Figure 5.7 shows the results of the 2D convolution implemented and validated in the FPGA. There is only one error detected due to the approximations done for the quantization which is totally normal and do not affect the accuracy of the results.

## 5.4 Performance results

The 2D-Conv accelerator and the tiling algorithm are tested with three different layers that have different dimensions: the first and the last layer

**Figure 5.7:** 2D-Conv results in FPGA

of MobileNet, and a common layer taken from [23]. The dimensions of the layers are shown in table 5.1, the stride is 1 and no padding is applied in all the layers. The test also considers different PLMs sizes, two values for the number of PEs (4 and 16, which corresponds to the Cout size of the PLMs), and the 3 kind of precision configurations (16, 8, or 4 bits). The tensor dimensions are represented in the following order: Height (or kernel height) x Width (or kernel width) x Input Channels x Output Channels.
Table 5.1 presents the results of the tests. The first column shows the time, in clock cycles, that takes to the accelerator to compute the layer. The second column shows the time the processor takes to compute the layer in software, considering the tiling algorithm as well. The third column shows the time that takes to the processor to compute the entire layer without the tiling algorithm. The fourth column shows the speedup of the hardware accelerator approach (HW) in comparison with the processor approach (SW) with tiling,

a negative value means that the accelerator takes less time to perform the computation. The fifth column shows the speedup of the hardware accelerator approach (HW) in comparison with the processor approach (SW) without tiling. The rest of the columns provides information about the layer that has been processed and the parameters given by the tiling algorithm.
The results demonstrate that the accelerator performs the best when the number of tiles is low and when the tile sizes are large. In these cases the accelerator better exploits each DMA transaction, moving more data in each transaction and spending less time configuring the accelerator and the DMA transactions. The results suggest that the largest PLMs are preferred because they permit to have less number of tiles and tiles with bigger dimensions. The tiling and DMA transactions overhead can become more significant when the number of tiles are bigger and the size of the tiles are smaller, impacting the overall performance of the accelerator and the tiling algorithm. Figure 5.8 shows the RTL simulation waveforms of the worst case experiment of table 5.1 (10th row) and Figure 5.9 shows the RTL simulation waveforms of the best case experiment of table 5.1 (15th row). In the worst case the percentages of time with respect to the total time are the following: the load phase 32%, the computation phase 56%, and the store phase 12%. Instead in the best case the percentages of time with respect to the total time are: 34%, 44%, and 22% respectively. This proves that in approximately half of the total time, the accelerator performs data movements for reading and writing. To improve the performance, a hierarchical design (with load, compute, and store phases pipelined with a dataflow directive), and double buffers for the accelerator's PLMs would hide the data transfer times. These improvements are left as future work.

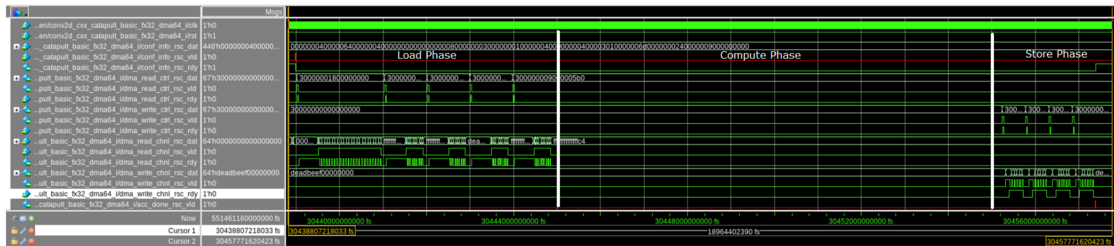


**Figure 5.8:** RTL simulation of the 2D-Conv computation for the worst case scenario

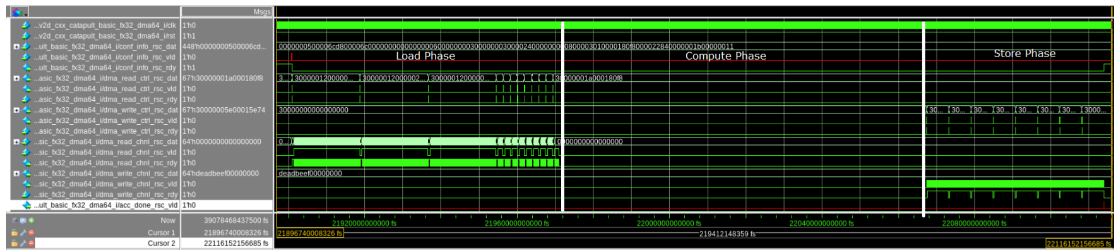


**Figure 5.9:** RTL simulation of the 2D-Conv computation for the best case scenario

| HW Time (CLK) | SW tiled Time (CLK) | SW Time (CLK) | HW/SW tiled speedup (%) | HW/SW speedup (%) | Layer and dimensions (HxWxCinx-CoutxK) | PLM IN - OUT / WEIGHT dimensions (HxWxCinx-Cout) (KxKx-CinxCout) | Number of Tiles, Dimensions (HxWxCinxCout), Input (HxWxCin) and Weight (KxKxCinxCout) Tile sizes , and Precision |
|---|---|---|---|---|---|---|---|
| 201612 | 491439 | 539587 | -59.0 | -62.64 | Average Layer, 8x8x16x4x3 | 18x18x16x16, 7x7x16x16 | 1, 8x8x16x4, 1024 - 576, 16 bit |
| 193670 | 492609 | 540793 | -60.7 | -64.19 | Average Layer, 8x8x16x4x3 | 18x18x16x16, 7x7x16x16 | 1, 8x8x16x4, 1024 - 576, 8 bit |
| 192872 | 492617 | 540847 | -60.8 | -64.34 | Average Layer, 8x8x16x4x3 | 18x18x16x16, 7x7x16x16 | 1, 8x8x16x4, 1024 - 576, 4 bit |
| 385840 | 525075 | 541069 | -26.5 | -28.69 | Average Layer, 8x8x16x4x3 | 7x7x16x16, 7x7x16x16 | 2, 8x8x8x4, 512 - 288, 16 bit |
| 375124 | 523193 | 539747 | -28.3 | -30.50 | Average Layer, 8x8x16x4x3 | 7x7x16x16, 7x7x16x16 | 2, 8x8x8x4, 512 - 288, 8 bit |
| 373262 | 523287 | 539487 | -28.7 | -30.81 | Average Layer, 8x8x16x4x3 | 7x7x16x16, 7x7x16x16 | 2, 8x8x8x4, 512 - 288, 4 bit |
| 1491223 | 700194 | 540908 | 113.0 | 175.69 | Average Layer, 8x8x16x4x3 | 7x7x4x16, 7x7x4x16 | 8, 8x8x2x4, 182 - 72, 16 bit |
| 1485316 | 694232 | 539689 | 114.0 | 175.22 | Average Layer, 8x8x16x4x3 | 7x7x4x16, 7x7x4x16 | 8, 8x8x2x4, 182 - 72, 8 bit |
| 4439336 | 3463583 | 539464 | 28.2 | 2972.70 | Average Layer, 8x8x16x4x3 | 7x7x4x16, 7x7x4x16 | 24, 3x8x4x4, 96 - 144, 4 bit |
| 17464261 | 4794601 | 540984 | 264.2 | 3128.24 | Average Layer, 8x8x16x4x3 | 7x7x4x4, 7x7x4x4 | 96, 3x8x1x4, 24 - 36, 16 bit |
| 8742729 | 4458131 | 539773 | 96.1 | 1519.70 | Average Layer, 8x8x16x4x3 | 7x7x4x4, 7x7x4x4 | 48, 3x8x2x4, 48 - 72, 8 bit |
| 4438198 | 3463673 | 539621 | 28.1 | 722.47 | Average Layer, 8x8x16x4x3 | 7x7x4x4, 7x7x4x4 | 24, 3x8x4x4, 96 - 144, 4 bit |

| 76913244 | 7039674661 | 57406251 | -99.3 | -8.80 | First layer MobileNet, 96x96x3x8x3 | 18x18x16x16, 7x7x16x16 | 282, 3x96x1x8, 288 - 72, 16 bit |
|---|---|---|---|---|---|---|---|
| 35403581 | 6654219550 | 56909831 | -99.5 | -37.79 | First layer MobilNnet, 96x96x3x8x3 | 18x18x16x16, 7x7x16x16 | 188, 3x96x2x8, 576 - 144, 8 bit |
| 17692218 | 5395975638 | 57401763 | -99.7 | -69.18 | First layer MobileNet, 96x96x3x8x3 | 18x18x16x16, 7x7x16x16 | 94, 3x96x3x8, 864 - 216, 4 bit |
| 104644738 | 7180285575 | 57406595 | -98.5 | 82.29 | First layer MobileNet, 96x96x3x8x3 | 18x18x16x4, 7x7x16x4 | 564, 3x96x1x4, 288 - 36, 16 bit |
| 69765942 | 6785121501 | 56906888 | -99.0 | 22.60 | First layer MobileNet, 96x96x3x8x3 | 18x18x16x4, 7x7x16x4 | 376, 3x96x2x4, 576 - 72, 8 bit |
| 35216349 | 5475623532 | 56915317 | -99.4 | -38.13 | First layer MobileNet, 96x96x3x8x3 | 18x18x16x4, 7x7x16x4 | 188, 3x96x3x4, 864 - 108, 4 bit |
| 208236894 | 7450825997 | 57406873 | -97.2 | 262.74 | First layer MobileNet, 96x96x3x8x3 | 16x16x4x4, 7x7x4x4 | 1128, 3x96x1x2, 288 - 18, 16 bit |
| 140916813 | 7044438935 | 56906939 | -98.0 | 147.63 | First layer MobileNet, 96x96x3x8x3 | 16x16x4x4, 7x7x4x4 | 752, 3x96x2x2, 576 - 36, 8 bit |
| 69986943 | 5728349776 | 56909180 | -98.8 | 22.98 | First layer MobileNet, 96x96x3x8x3 | 16x16x4x4, 7x7x4x4 | 376, 3x96x3x2, 864 - 54, 4 bit |
| 47893013 | 17234456 | 14320720 | 177.9 | 234.43 | Last layer MobileNet, 3x3x256x256x1 | 18x18x16x16, 7x7x16x16 | 256, 3x3x16x16, 144 - 256, 16 bit |

| 47642604 | 17230042 | 14327886 | 176.5 | 232.52 | Last layer MobileNet, 3x3x256x256x1 | 18x18x16x16, 7x7x16x16 | 256, 3x3x16x16, 144 - 256, 8 bit |
|---|---|---|---|---|---|---|---|
| 47519053 | 17237134 | 14327862 | 175.7 | 231.65 | Last layer MobileNet, 3x3x256x256x1 | 18x18x16x16, 7x7x16x16 | 256, 3x3x16x16, 144 - 256, 4 bit |
| 187780605 | 21357954 | 14324971 | 779.2 | 1210.86 | Last layer MobileNet, 3x3x256x256x1 | 18x18x16x4, 7x7x16x4 | 1024, 3x3x16x4, 144 - 64, 16 bit |
| 189312157 | 21326110 | 14327630 | 787.7 | 1221.31 | Last layer MobileNet, 3x3x256x256x1 | 18x18x16x4, 7x7x16x4 | 1024, 3x3x16x4, 144 - 64, 8 bit |
| 189011585 | 21329425 | 14328101 | 786.2 | 1219.17 | Last layer MobileNet, 3x3x256x256x1 | 18x18x16x4, 7x7x16x4 | 1024, 3x3x16x4, 144 - 64, 4 bit |

**Table 5.1:** 2D-Conv Performance results

# Chapter 6

# Conclusions

Convolutional Neural Networks (CNNs) have become pivotal in numerous computer vision applications, enabling advancements across diverse domains like image classification, object detection, and medical imaging. The escalating demand for these applications has underscored the critical need for more efficient, scalable solutions that can meet the computational demands of cutting-edge CNN models.
This thesis addresses these challenges by presenting an innovative tiling architecture tailored for large-scale CNN inference. The emphasis on High-Level Design and the Embedded Scalable Platform (ESP) underscores a paradigm shift in designing hardware accelerators and optimizing tensor partitioning for memory-constrained edge devices. The thesis work has demonstrated how is possible to execute larger CNN layers in a memory constrained device thanks to the tiling method proposed within the architecture. However, the results have demonstrated that in order to achieve the best performance of the accelerators and tiling algorithm, the PLMs have to be large enough to avoid getting a high number of small tiles after applying the tiling algorithm. Moreover, the utilization of High-Level Design enables a higher-level representation of hardware, yielding more concise, debug-friendly C/C++ code. ESP complements this approach by seamlessly integrating the architecture into a System-on-Chip (SoC), facilitating straightforward integration and testing of baremetal software applications that test the proposed algorithm. Rigorous testing through RTL simulation and FPGA deployment has validated the feasibility and real-world effectiveness of the proposed tiling algorithm.

**Future work:**

- Exploring dynamic tiling strategies that adapt to other types of dataflows and memory constraints in real-time scenarios.
- Exploring the management of the tiles with a hardware approach instead of the software approach used in this work.
- Evaluating the tiling algorithm on a complete quantized DNN with mixed-precision to validate the ST-based accelerators and the tiling overhead.

In conclusion, this thesis highlights the fundamental role of innovative tiling architectures and High-Level Design in enhancing CNN inference on edge devices. By addressing critical challenges, this research lays a robust foundation for future advancements in efficient CNN deployment.

# Bibliography


[1] Dashanka Nadeeshan. *Advancements of Deep Learning 1: Introduction to Convolutional neural networks and Main operations*. May 2018. URL: https://medium.com/coinmonks/advancements-of-convolutional-neural-networks-part-1-introduction-and-main-operations-5d12f35b28d4 (cit. on p. 2).

[2] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. Dec. 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (cit. on p. 4).

[3] Atul Pandey. *Depth-wise Convolution and Depth-wise Separable Convolution*. Sept. 2018. URL: https://medium.com/@zurister/depth-wise-convolution-and-depth-wise-separable-convolution-37346565d4ec (cit. on pp. 6, 9).

[4] Kh Shahriya Zaman, Mamun Bin Ibne Reaz, Sawal Hamid Md Ali, Ahmad Ashrif A Bakar, and Muhammad Enamul Hoque Chowdhury. «Custom Hardware Architectures for Deep Learning on Portable Devices: A Review». In: *IEEE Transactions on Neural Networks and Learning Systems* 33.11 (2022), pp. 6068–6088. DOI: 10.1109/TNNLS.2021.3082304 (cit. on p. 12).

[5] Yunji Chen et al. «DaDianNao: A Machine-Learning Supercomputer». In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 2014, pp. 609–622. DOI: 10.1109/MICRO.2014.58 (cit. on p. 12).

[6] Google. *Edge TPU*. Accessed: 2023-11-25. URL: https://cloud.google.com/edge-tpu?hl=es (cit. on p. 12).

[7] Robert David et al. *TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems*. 2021. arXiv: `2010.08678 [cs.LG]` (cit. on p. 12).

[8] Gang Li, Zejian Liu, Fanrong Li, and Jian Cheng. *Block Convolution: Towards Memory-Efficient Inference of Large-Scale CNNs on FPGA*. 2021. arXiv: `2105.08937 [cs.AR]` (cit. on pp. 12, 13).

[9] Nicolas Tollenaere, Auguste Olivry, Guillaume Iooss, Hugo Brunie, Albert Cohen, et al. «Efficient convolution optimisation by composing micro-kernels». In: (2021) (cit. on pp. 12, 15).

[10] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. «DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs». In: *IEEE Transactions on Computers* 70.8 (Aug. 2021), pp. 1253–1268. DOI: `10.1109/tc.2021.3066883`. URL: `https://doi.org/10.1109%2Ftc.2021.3066883` (cit. on pp. 12, 13, 15, 36).

[11] Angelo Garofalo, Gianmarco Ottavi, Francesco Conti, Geethan Karunaratne, Irem Boybat, Luca Benini, and Davide Rossi. *A Heterogeneous In-Memory Computing Cluster For Flexible End-to-End Inference of Real-World Deep Neural Networks*. 2022. arXiv: `2201.01089 [cs.AR]` (cit. on pp. 16, 17).

[12] Yu-Sheng Lin, Hung-Chang Lu, Yang-Bin Tsao, Yi-Min Chih, Wei-Chao Chen, and Shao-Yi Chien. «GrateTile: Efficient Sparse Tensor Tiling for CNN Processing». In: *2020 IEEE Workshop on Signal Processing Systems (SiPS)*. 2020, pp. 1–6. DOI: `10.1109/SiPS50750.2020.9195243` (cit. on pp. 17, 18).

[13] Angshuman Parashar et al. «Timeloop: A Systematic Approach to DNN Accelerator Evaluation». In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2019, pp. 304–315. DOI: `10.1109/ISPASS.2019.00042` (cit. on p. 18).

[14] Chan Park, Sungkyung Park, and Chester Sungchung Park. «Roofline-Model-Based Design Space Exploration for Dataflow Techniques of CNN Accelerators». In: *IEEE Access* 8 (2020), pp. 172509–172523. DOI: `10.1109/ACCESS.2020.3025550` (cit. on pp. 19–22).

[15] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. «Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks». In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 367–379. DOI: 10.1109/ISCA.2016.40 (cit. on p. 20).

[16] Luca Urbinati and Mario R. Casu. «Design-Space Exploration of Mixed-precision DNN Accelerators based on Sum-Together Multipliers». In: *2023 18th Conference on Ph.D Research in Microelectronics and Electronics (PRIME)*. 2023, pp. 377–380. DOI: 10.1109/PRIME58259.2023.10161835 (cit. on pp. 23, 25, 60, 71, 77).

[17] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. «Agile SoC development with open ESP». In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ACM, Nov. 2020. DOI: 10.1145/3400302.3415753. URL: https://doi.org/10.1145%2F3400302.3415753 (cit. on pp. 27, 30–32).

[18] Maico Cassel Dos Santos et al. «A Scalable Methodology for Agile Chip Development with Open-Source Hardware Components : (Invited Paper)». In: *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2022, pp. 1–9 (cit. on pp. 27, 28).

[19] Federico Perenno. «High-Level Design of 2D-Convolution Accelerators for AI Leveraging Embedded Scalable Platform (ESP)». MA thesis. Torino, Italy: Politecnico di Torino, 2022 (cit. on pp. 60, 81, 83, 84).

[20] Riccardo Capodicasa. «High-level design of a Depthwise Convolution accelerator and SoC integration using ESP». MA thesis. Torino, Italy: Politecnico di Torino, 2022 (cit. on pp. 60, 81, 83).

[21] Columbia University - System Level Design Group. *ESP Accelerator Specifications*. Accessed: 2023-11-30. URL: https://www.esp.cs.columbia.edu/docs/specs/esp_accelerator_specification.pdf (cit. on p. 61).

[22] Marco Alessio Terlizzi. «Mixed-Precision Quantization and Inference of MLPerf Tiny DNNs on Precision-Scalable Hardware Accelerators». MA thesis. Torino, Italy: Politecnico di Torino, 2023 (cit. on p. 73).

[23] Luca Urbinati and Mario R. Casu. «A Reconfigurable 2D-Convolution Accelerator for DNNs Quantized with Mixed-Precision». In: *Applications in Electronics Pervading Industry, Environment and Society*. Ed. by Riccardo Berta and Alessandro De Gloria. Cham: Springer Nature Switzerland, 2023, pp. 210–215. ISBN: 978-3-031-30333-3 (cit. on p. 90).